Technische Universität Braunschweig

Institut für Nachrichtentechnik

Bachelor Thesis

Extension of an Open Source Tool for Manipulating CAN Bus Data

Fynn Denecke
Matrikelnummer: 5001452

22.09.2023

Technische Universität Braunschweig
Institute for Communications Technology
Schleinitzstraße 22 · 38106 Braunschweig

Prüfer: Prof. Dr.-Ing. Ulrich Reimers, Prof. Dr. Michael Terörde
Betreuer: Dr.-Ing. Oliver Hartkopp, Jonas von Beöczy, M.Sc.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Bachelor Thesis mit dem Titel

„Extension of an Open Source Tool for Manipulating CAN Bus Data“

selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Braunschweig, 22.09.2023

Fynn Denecke

Preface

Thanks to the Volkswagen AG for allowing me to pursue this bachelor thesis with their assistance. Thanks also to all the colleagues in the ESEC/3 subdivision, especially to Dr. Oliver Hartkopp, who helped greatly in the realization of this project with his expertise and professional attitude.

Abstract

The Controller Area Network (CAN) bus system and its successor CAN FD are the most widespread bus systems in the automotive industry.

CAN has such a long history, that attacks have been developed that can be programmed to fit onto just a small microcontroller. These attacks are collected in the CANHack toolkit by Dr. Ken Tindell. The attacks in this program go as far as being able to shut down targeted CAN nodes completely or overwrite sent datagrams with customized data.

However, nowadays Controller Area Network Flexible Data-Rate (CAN FD) is the far more used and attractive protocol, since it is both faster and contains larger amounts of data. So far, the CANHack toolkit is unable to target frames sent using this new standard.

This is where this bachelor thesis sets in by expanding CANHacks functionality to create the first open-source CAN FD hacking tool. On the journey to this goal the toolkit will not only undergo additional changes to its interface and software building process, but also the inner workings of both protocols and the attacks offered by the program will be explored. The result of this project and the purpose of the improved CANHack toolkit is to allow for more stress testing methods in automotive products and for them to become more secure in the process.

Contents

1 Introduction

The Controller Area Network (CAN) has been a part of the automotive industry ever since the standards first introduction, thanks to its reliable and secure data transmission.

The role of testing setups to simulate attacks on CAN-Bus also cannot be underestimated, since especially in this market the error safety is a big factor in the success of new products. One of the tools designed to purposefully break CAN-Bus' security is the CANHack toolkit by Dr. Ken Tindell, Chief Executive Officer (CEO) of Canis Automotive Labs, which this thesis will examine to gain a deeper understanding of CAN-Bus' weaknesses.

Of course, CAN has not stagnated facing the recent advancement of automotive technology, instead it has been updated to the new Controller Area Network Flexible Data-Rate (CAN FD) standard in order to address the new challenges and possibilities posed by these developments. CAN FD is essential to almost any newly produced vehicle nowadays and offers fast and reliable transmission of large amounts of data in interconnected systems.

However, the amount of low-level testing soft- and hardware for CAN FD is limited to say the least. While the aforementioned CANHack toolkit can fill this gap for classical CAN, as of now the program does not support the CAN FD protocol, either. After having a look at the differences between the two protocols, the thesis will try to establish a way of incorporating the new CAN FD standard into the already existing toolkit. Additionally other optimizations of the currently offered tools and the setup of the program in general will be evaluated. Naturally, the user experience of CANHack shall remain the same, if not giving more options for inputs and settings regarding the attacks launched.

The final goal is to create an advanced CAN FD hacking toolkit that offers at least as many features as the base version without compromising on simplicity and the base functionality.

2 CAN Bus

The Controller Area Network, mostly referred to as CAN-Bus, is a serial bus system released by Bosch in 1986 [IMP15]. It is standardized by the International Organization of Standardization (ISO) through ISO11898 since 1993 [OS01], with the latest changes dating back to December of 2015. The ISO standard has multiple parts, however this project will mostly focus on part one, which describes the data link layer and the physical signaling of CAN-Bus [ISO21].

2.1 Connection

The CAN-Bus uses two cables for data transmission, CAN_H (high) and CAN_L (low). There may be an additional cable for each CAN_GND (ground) and a 5V power connection. In CAN systems a bit set to 0 is considered dominant, while one set to 1 is considered recessive [LXQ12]. In a recessive state, both CAN_H and CAN_L sit at an equal amount of voltage, which is usually 2.5V. To transmit a dominant bit, the level of CAN_Hs voltage rises to at least 3.5V, while CAN_Ls is lowered down to a maximum of 1.5V, so the average voltage of the bus remains the same. This process can be observed in figure 2.1. The state of the bus is determined by the difference in voltage V_{diff}, which is 0V for recessive and +2V for dominant bits. [HPL02].

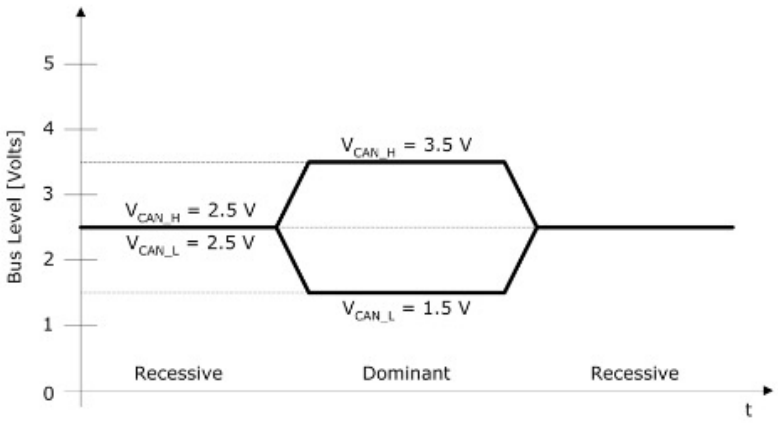

Figure 2.1: Voltage levels on CAN Bus [Vos19]

2.2 Frame Types

Frames sent through CAN Bus are divided into four different kinds. The data frames are used to transmit data between nodes. Remote frames can be used to request data frames from other nodes, these data frames will then be sent using the same identifier as the remote frame. Both data and remote frames use bit-stuffing to prevent and detect errors. After every fifth consecutive bit of one kind there is a stuff bit of the opposite value added. This technique is used up until and not including the Acknowledgement (ACK) bit. [AMB20]

Error frames can be thrown by any node connected to the bus that detects an error. They only consist of 6 consecutive bits, either dominant or recessive, depending on the current state of the controller. The three states a node can assume are error-active, error-passive and bus-off. An error-active node sends an error frame consisting of six dominant bits, while error-passive nodes send six recessive bits. Error-passive nodes additionally can only write onto the bus if no error-active node tries to do so. A node in bus-off mode is basically switched off. Nodes assume these states automatically through counting sent and received error frames, as seen in figure 2.2. [OS01]

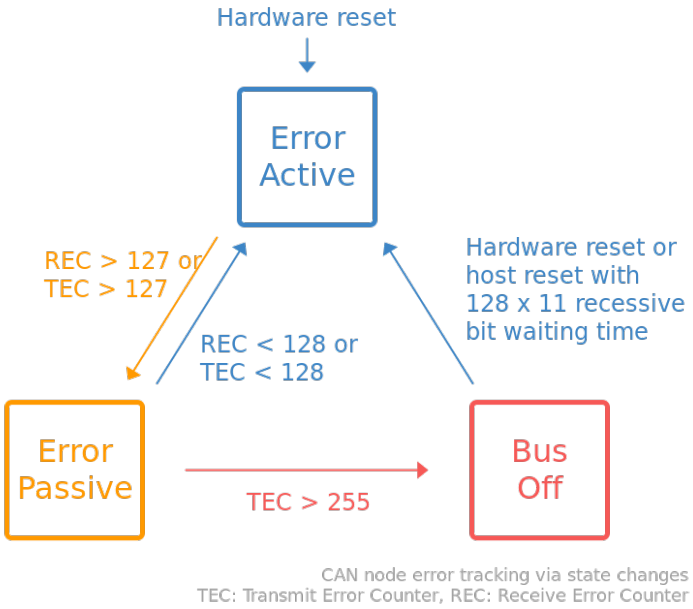

Figure 2.2: CAN Error States [Ele22]

Any transmitted error frame will increase the Transmit Error Counter (TEC) by 8, while succesfully transmitted frames will decrease it by 1. The Receive Error Counter (REC) is raised by 8 for the first error frame received and by 1 for every subsequent one. Receiving a valid frame will lower the REC by 1. [Blo21]

The last kind of frames are overload frames, which function as placeholders between data or remote frames. They contain the overload flag, echo and delimiter. The overload flag are 6 consecutive dominant bits, with the overload echo extending this series by up to 6 more dominant bits. Closing out the frame is the overload delimiter consisting of 8 dominant bits. [CF07]

2.3 Data And Remote Frame Format

All information in this section is based on the CAN ISO11898-1:2015 standard [Int15].

Figure 2.3: (a) Normal CAN Frame (b) Extended CAN Frame

A CAN data frame begins with one dominant bit, the Start Of Frame (SOF). After that comes the 11 bit identifier, which can be extended up to 29 bit by setting the Identifier Extension (IDE) bit to recessive. The Remote Transmission Request (RTR) bit is used to indicate a remote frame, in extended formats the original RTR bit is replaced by the Substitute Remote Request (SRR) bit. This whole section of the frame is called the arbitration field. Arbitration in CAN-Bus works through priority, as all CAN nodes function as masters and all frames are broadcast. Whenever two frames are sent through the bus at the same time, the one with a lower identifier will win arbitration. Additionally data frames will win over a corresponding remote frame. The process happens without loss of data or time, the losing frame will be re-transmitted as soon as possible. A CAN frame contains no information as to its destination and origin, the receivers accept frames solely based on their relevance, this is called acceptance filtering.

After the arbitration field comes the control field. It contains the r0 and, depending on whether the frame is extended, the IDE or r1 bit and the Data Length Code (DLC). The r-bits, r standing for reserved, merely serve as placeholders. In data frames the DLC indicates the size of the following data field. Remote frames use the DLC to specify the amount of data requested from their destination. The DLC is 4 bits long and contains a binary value, with the excpetion of 8, which is set for any value where DLC3 = 1.

Table 2.1: DLC values

Data Bytes	DLC3	DLC2	DLC1	DLC0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	X	X	X

Next is the data field of up to 8 bytes. It only exists in data frames and carries the actual payload of the CAN frame. Its length is specified in the DLC and always has to be that exact size, longer contents need to be cut down and shorter ones extended with 0s. If the amount of bytes does not match up with the DLC, an error frame will be thrown.

Every non-stuff bit up until the end of the data field is used to calculate the 15-bit long Cyclic Redundancy Check (CRC). It uses a generator polynomial for division, which in the case of CAN is set as the following: $x^{15}+x^{14}+x^{10}+x^{8}+x^{7}+x^{4}+x^{3}+1$. CANs CRC offers a Hamming distance of 6, meaning that 5 individual bit failures can be detected, additionally the polynomial allows for burst-error-detection of up to 15 bit. Unfortunately the variable data length and stuff bits impact these numbers, so that for some patterns the Hamming distance is reduced to 2. If the CRC sent in the frame does not match up with the one calculated by the receiving nodes, the other nodes may throw an error frame or correct the mistake. The CRC field ends with the CRC Delimiter, a single recessive bit. This is also the point where CAN stops employing bit-stuffing.

The Acknowledgement (ACK) bit is sent out as recessive but can be overwritten by any node connected to the bus if the frame is recognized as valid. A dominant ACK bit indicates a successful transmission, however it is possible that recessive ACKs are overwritten in networks with multiple CAN nodes. In this case an error frame can be thrown in the following, always recessive, ACK Delimiter.

Finally the last bits are taken up by the End Of Frame (EOF) and the Intermission Frame Space (IFS). They consist of 10 combined recessive bits, 7 in the EOF and 3 in the IFS. While the EOF marks the actual last bits of the frame, the IFS makes sure the minimum distance of 3 bits is kept between transmissions.

3 CANHack Toolkit

The CANHack Toolkit is an open-source project created by Dr. Ken Tindell, the CEO of Canis Automotive Labs. It runs using custom modifications for MicroPython and offers support for both the Raspberry Pico and Pico W. While originally intended to be used on the Canis Labs CANPico board, CANHack board and the Car Hacking Village DEF CON 30 badge, it also works on basic versions of the Pico, as long as it has a CAN-Bus transceiver connected to it.

3.1 Set Up

This section discusses the setup of both the soft- and hardware of the CANHack project as it has been executed for this thesis. There may be some variation based on the parts used.

3.1.1 Hardware

For this project, a standard Raspberry Pico has been extended with an MCP2512FD CAN transceiver and a male SUB-D9 plug. The MCP2512FD was soldered onto an RE899 circuit board and serves as a middleman between the Pico and the plug, processing CAN data in and out.

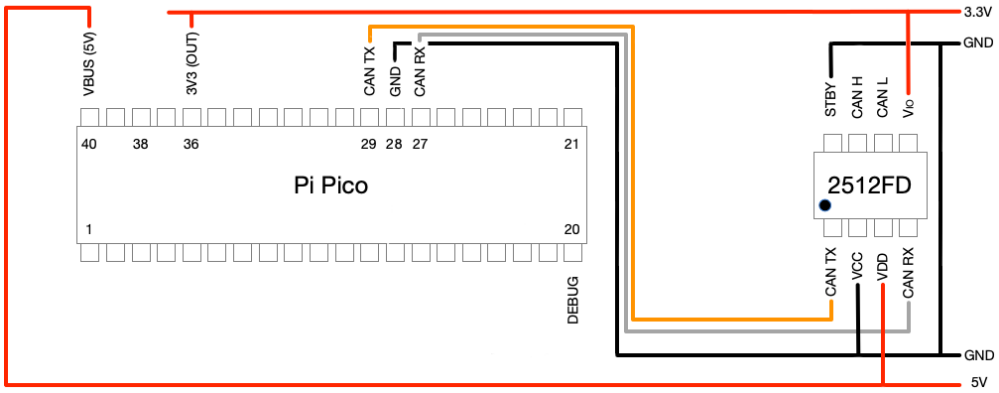

Figure 3.1: Breadboard diagram for CAN on Pico

As seen in figure 3.1, the MCP2512FD requires two power connections, one 5V and one 3V3. The 5V outlet is connected from pin 40 to the VIO and 3V3 from pin 36 to VCC. The CAN Receive (RX) and Transmit (TX) pins are connected to pins 27 and 29 respectively. The 28th pin of the Pico is used for both ground connections on the MCP2512FD, Standby (STBY) and VDD.

In the upcoming figures 3.2 and 3.3 the actual physical connection used for this project is displayed. The first one also makes clear why the Ground (GND) pins have been changed from the recommended setup: The pin 28 offers itself nicely because it sits right in the middle of the CAN RX and TX pin. The two power connections, 3V3 color-coded to magenta instead of red here, are located on the bottom of the Pico, as well.

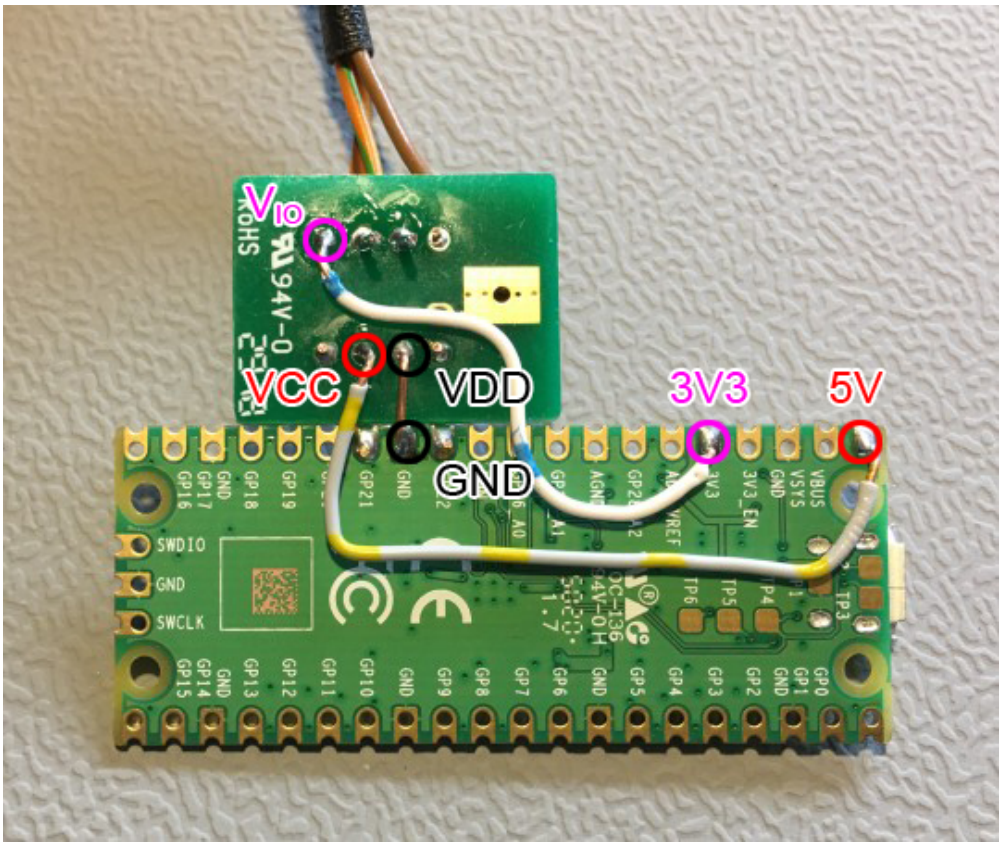

Figure 3.2: Power and GND connections (bottom side)

The pins 27 and 29, in the figure 3.3 described by their software identifiers General Purpose (Input/Output) (GP) 21 and 22, serve as the in- and output of the Raspberry Pico, so data can be easily sent to and read from the CAN transceiver. Their cable also happens to support the construction physically, as thanks to them and the GND connection from figure 3.2 make it so that no additional hardware for stabilizing is needed. Another thing that figure 3.3 demonstrates is the connection between the GND or VDD of the MCP2512FD to its own STBY pin. Since both of these pins just need to be grounded, there is no need for another GND pin of the Pico to be accessed. The last part of the transceivers pinout are the CAN_H and CAN_L lines, whose functionality for CAN-Bus has been explained in section 2.1.

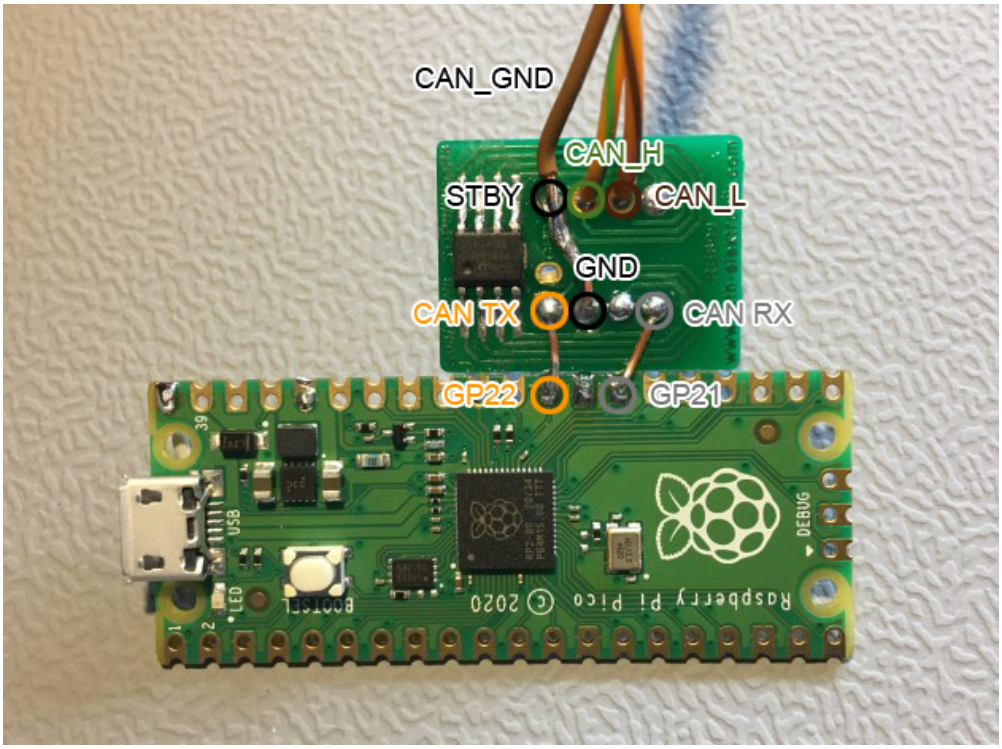

Figure 3.3: CAN plug connections (top side)

Additionally there are some other devices aiding in this project, mostly for testing. The PCAN-Diag FD offers an oscilloscope specifically designed for CAN and several other features regarding diagnosis of CAN communication and is heavily used in this project for checking the effects of the bit-banging performed by the base and modified version of the CANHack toolkit. In order to test out attacks, multiple USB CAN adapters are connected to the same Linux computer as the Pico and communicated with via SocketCAN. All these devices, including the CANHack setup, are connected to the same CAN-Bus with two 120Ω termination resistors. A diagram of this setup can be found on next pages figure 3.4, as well as a picture taken in person of the actual setup in figure 3.5.

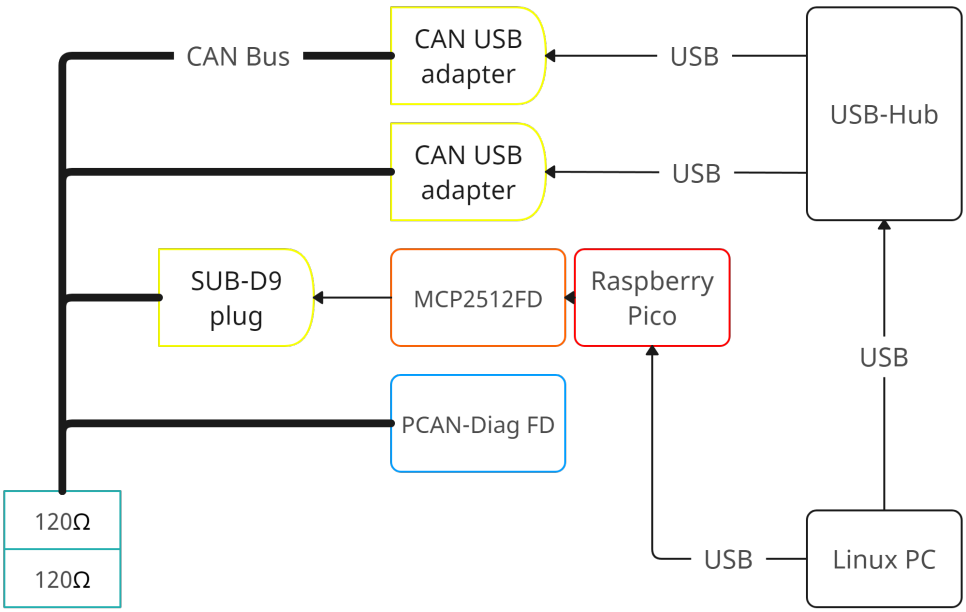

Figure 3.4: Full setup diagram

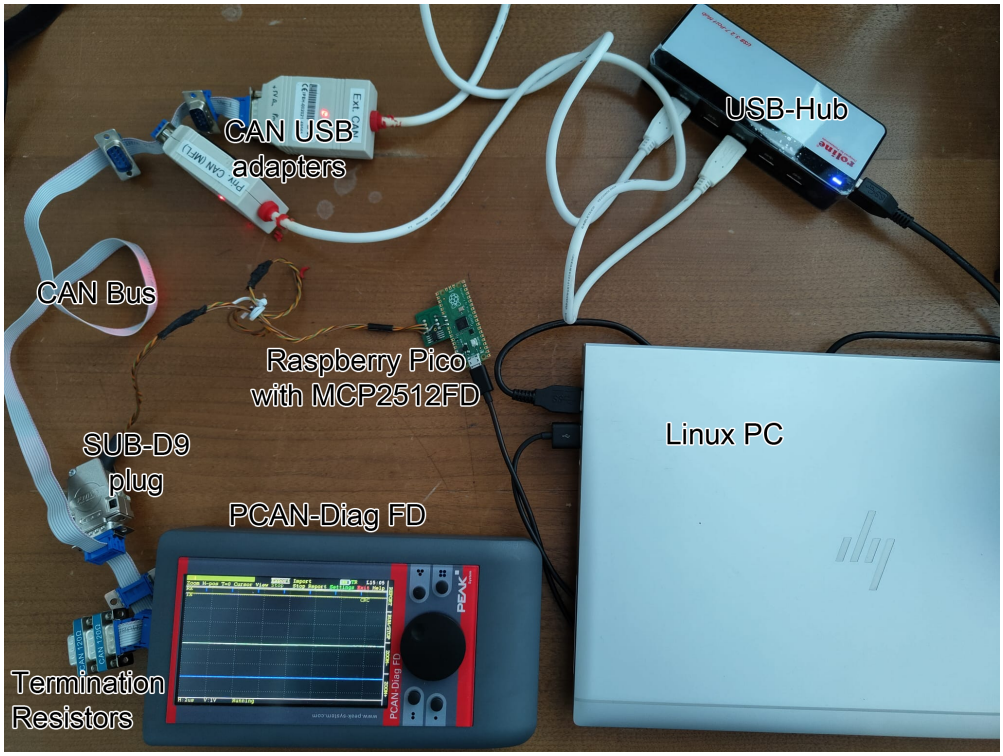

Figure 3.5: Picture of actual setup

3.1.2 Software

The GitHub project offers a already pre-compiled firmware.uf2 that includes not only the CANHack Toolkit, but also the Canis Labs CAN Software Development Kit (SDK) and support for the MCP2517FD external CAN transceiver, as well as Microcontroller Interconnect Network (MIN), another project by Dr. Tindell, which aims to directly connect microcontrollers.

To build the firmware from the ground up, the file structure from figure 3.6 should be constructed. This process is also described in more detail and with links to the necessary git repositories in the README.txt inside canhack/pico/micropython, except for the required downgrade of the CAN SDK to its first release. After establishing the structure, the firmware can be built in the micropython/ports/rp2 directory by using cmake with the options mentioned in the README.txt and make. The instructions as contained in the README.txt can be found in the appendix A.1.

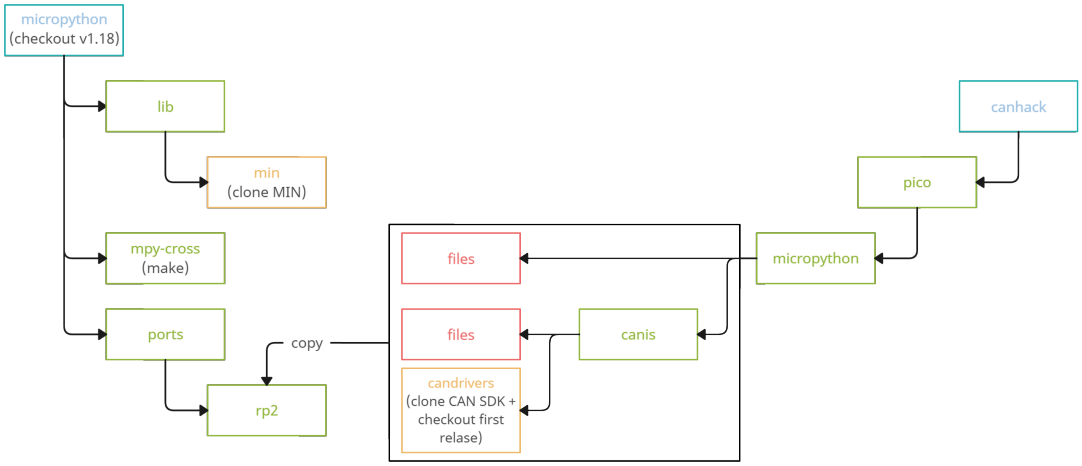

Figure 3.6: Project structure for building CANHack firmware

Whether or not the firmware is built from the ground up or pulled from the repository, it can be flashed onto the Pico by holding the boot button while connecting it to the PC via USB and copying the .uf2-file into the Pico that shows up as a removable disk.

3.2 Current Functionality

This section will discuss the functionalities of the CANHack toolkit only, since the other installed MicroPython mods run independently of it and are not considered in the further development of this project.

Going forward, CANHacks Application Programming Interface (API) functionalities written in Python will be referred to as methods, while the corresponding ones written in C will be referred to as functions.

3.2.1 General

The main.c, which is executed after flashing, is very similar to the one for original MicroPython and allows for the typical communication types such as Bluetooth or Universal Serial Bus (USB). It also supports MIN, which is mentioned in the comments to be 'conditionally included' in the future, instead of hard coded as it is right now. The prompts can be written to the Pico in regular Python fashion and are then routed through modrp2.c using MP_QSTR macros to end up being processed in rp2_canhack.c inside the canis directory. This file bridges the gap between the API and the canhack.c that contains most of the actual functionality in the form of functions.

The bit-banging is performed by using pulse width modulation (PWM), the free-running timer increases by 1 with every system clock cycle. The Raspberry Picos standard frequency sits at 125 megahertz (MHz), so the bit time has been set to 249. While 250 clock cycles would hit exactly 500 kilobits per second (kbps) mathematically, one cycle has been subtracted to account for the delay when setting the CAN TX pin.

After establishing a connection with the flashed Pico, the CANHack toolkit has to be initialized like a normal Python class to access its methods.

```
ch = rp2.CANHack(500)
```

It only has one argument to be given: the desired bit rate. It can be set to either 500 (standard), 250 or 125 kbps. This is achieved by adding a prescaler to the PWM. After Initialization the class is then able to execute the following methods:

set_frame() Constructs a valid CAN frame and saves it in one of two slots

get_frame() Returns a set frame as tuple

send_frame() Sends a frame on the CAN Bus

send_janus_frame() Performs a Janus Attack by sending a Janus frame

spoof_frame() Targets a frame and sends a spoof version

error_attack() Destroys a targeted frame with error frames

double_receive_attack() Causes a targeted frame to be received twice

freeze_doom_loop_attack() Causes the bus to freeze after a targeted frame has been transmitted

set_can_tx() Sets CAN TX pin

square_wave() Drives CAN TX for 160 bits with a 50:50 duty cycle square wave at half the CAN bit rate

loopback() Drives Trigger (TRIG) pin to the same value as CAN RX pin for 160 bit times after falling edge

get_clock() Returns the current clock time

reset_clock() Resets the free-running counter to zero

send_raw() Sends the raw bitstream of a frame on the CAN TX pin

print_frame() Prints out frame with color-coded fields

stop() Stops a currently executed attack (only in multi-threaded environments)

3.2.2 Set Frame

Before using most of the CANHack functions a CAN frame needs to be generated. To create this frame, the method set_frame() has to be called. It has seven possible arguments, with the CAN identifier (ID) as a requirement and the other six being optional.

can_id (int) The 11 or 29 bit ID of the CAN frame

remote (bool) True if the frame is a remote frame

extended (bool) True if the frame is extended (29 bit ID)

data (int) Up to 8 bytes of data for the frame to carry

set_dlc (bool) True if dlc should be set to a predefined value

dlc (int) Value the DLC shall be set to if set_dlc = True

second (bool) True if setting the shadow/secondary frame (for a Janus Attack)

Example call for set_frame():

```
ch.set_frame(can_id=0x123, data=bytes([0x01, 0x02, 0x03]))
```

This call sets up a non-extended data frame with a DLC of 3, the payload '01 02 03' and the hexadecimal identifier 123_{16}. All booleans are set to False if not otherwise specified.

There is not an error thrown if, for example, the can_id argument implies a 29 bit identifier while extended is set to negative. Instead the ID will simply be cut down to fit into 11 bits. The same goes for data if the set dlc is too short. The dlc argument will also be ignored if set_dlc is not True. There will only be a ValueError thrown if dlc is above 15, the payload is more than 8 bytes or both remote and data are set.

Those arguments are processed and then passed onto the canhack.c function canhack_set_frame(). It uses helper functions called do_crc() and add_bit(), the latter of which uses another helper function called add_raw_bit(), to construct a can_frame_t, which is defined using the struct command in the header file of canhack.c. The can_frame_t not only consists of a bitstream, it also tracks the total amount of bits, the position of stuff bits, the length of the arbitration field, the CRC value and the last bits of all CAN frame fields.

3.2.3 Send Frame

Frames are put onto the bus using the send_bits() function from canhack.c. It uses bit-banging on the predefined CAN TX pin. The toolkit does not employ a Interrupt Request (IRQ) for timing, as mentioned before, the internal pulse width modulation is used. The length of bits is predefined as an amount of clock ticks inside the rp2_canhack.h, which is set to 249 by standard. The offset from the bit start for sampling is also defined in the same header file, as sampling enables the function to recognize falling edges and use them to recalibrate its timing. The sample offset is set to 150. With the clock rate of the Pico sitting at 125 MHz and the added delay through the bit-setting itself, the speed of the send_bits() function comes out to 500 kbps. As mentioned earlier, it can be reduced by adding a prescaler at the Initialization of the CANHack class.

Send_bits() can not be called directly from the API, as it is merely a helping function for most of the other functions in canhack.c. To send the previously set frame without any specific intention one has to call the send_frame() method in the API.

timeout (int) Timeout for the process to stop (standard value: 50000000 -> ≈ 17 seconds on Pico)

second (bool) True if shadow frame should be sent

retries (int) Number of retries after arbitration loss or error

Example call for send_frame() using standard values:

```
ch.send_frame(second=False, retries=2)
```

A ValueError is thrown if the selected frame has not been set yet.

This method is linked to the canhack.c function canhack_send_frame(). Like the other C functions that use send_bits(), canhack_send_frame() pre-processes most of the arguments before passing them into the actual sending function. There is no checking of arbitration inside send_bits(), that is all done in canhack_send_frame() beforehand.

3.2.4 Janus Attack

A Janus Attack is performed by sending out a single frame with two different payloads. This can be achieved by switching the state of the bus before the end of the current bit time. It can cause different nodes to read separate values off the bus, since the sample points may differ between devices.

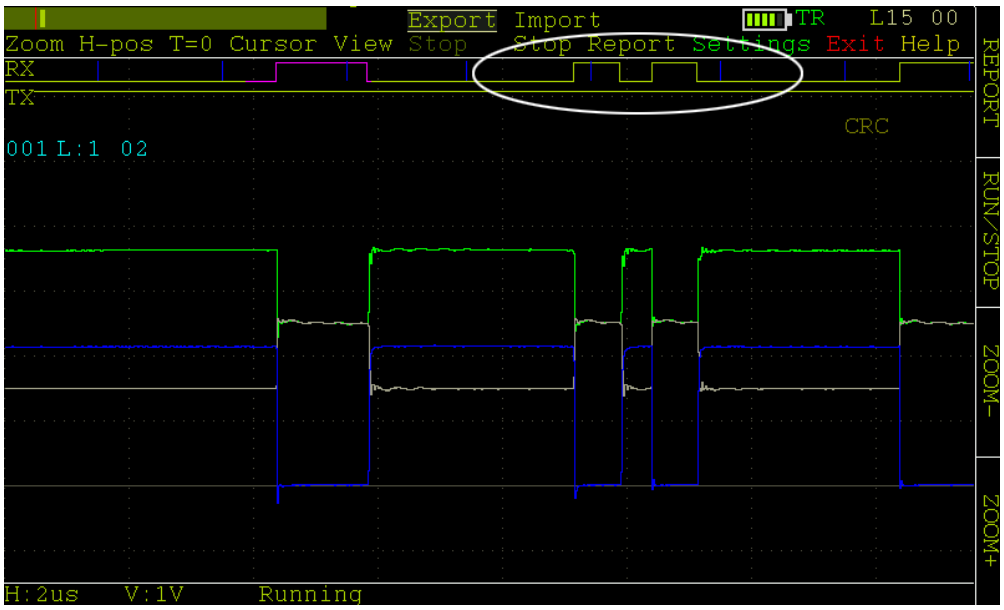

Figure 3.7: Janus Attack as seen on PCAN-Diag FD

More information on the PCANs scope can be found in the appendix A.2.

In the example in figure 3.7 the sample point of the PCAN-Diag FD is set at 80% of the frame, this causes the device to read the frames value as '02' (as seen in the top left), '10' in binary. If the sample point was at 60%, it would shift the blue line indicating the sample point to the left and read the value as '01' in both binary and decimal. To achieve this effect, the normal bit timing is extended by two more points of interest, the sync and the split time. At the beginning of a Janus Attack bit the bus is set dominant, trying to create a falling edge, if the previous bit was recessive, in order to achieve a resynchronization of the CAN controllers. Then comes the sync time, which marks the potential end of this dominant state, setting the bus to whatever value is set in the main frame. Once the split time is reached, the bus flips to the shadow frame bit.

To iniciate a Janus Attack via the CANHack toolkit, both the main and shadow frame have to be set using the set_frame() method. For a successful transmission it is required that both frames consist of the same total amount of bits, including stuff bits. Once both frames are set, the send_janus_frame() method can be called from the API.

timeout (int) Timeout for the process to stop (standard value: 50000000)

sync_time (int) Number of clock ticks to wait for sync between controllers

split_time (int) Number of clock ticks from start of bit until bit value switch

retries (int) Number of retries after arbitration loss or error

Example call for send_janus_frame that starts a Janus Attack with basic settings:

```
ch.send_janus_frame()
```

All arguments are optional, the standard sync and split times are set to 50 and 155, which is 25% and 62.5% of the total bit time, and no retries are performed by default. A ValueError is thrown if any of the frames have not been set.

Send_janus_frame() redirects to a canhack.c function of the same name, which uses the function send_janus_bits() to put the frame onto the bus after winning arbitration. Send_janus_bits() works similar to send_bits(), adding the ability to switch bit values during single bit times.

3.2.5 Error Attack

The Error Attack allows for a CAN node to be pushed into error-passive or even bus-off mode. The method needed to act out this kind of attack is simply called error_attack() and allows for the following arguments:

timeout (int) Timeout for the process to stop (standard value: 50000000)

repeat (int) Number of times to repeat the attack

Example call for an Error Attack that should drive a CAN node to error-passive ($16 * 8 = 128$ on the REC):

```
ch.error_attack(repeat=16)
```

The function returns a boolean of whether the timeout has been hit. If the main frame has not been set before executing, a ValueError is thrown.

The corresponding C function works by sampling the bus until the latest bits of the bitstream match up with the previously set frames identifier. Before starting this process, the identifier is saved as an attack parameter inside the CANHack class/structure. However, the attack parameters do not just contain the sole identifier, but add the minimum amount of recessive bits between data transmissions in front of it. There is also an attack mask being set, which is to be ANDed with the bitstream read from the bus. Once the identifier has been recognized, the bus is then bombarded with error-active error frames to the specified amount.

3.2.6 Spoof a frame

Spoofing a frame is the practice of basically mimicking a frames identifier and switching up the data contained inside of it. In this toolkits case, the ID being copied and the data replacement are pulled from the main frame set via the set_frame() method. The effects of this differ depending on whether the original sender of the frame is in error-passive or -active mode. In error-active, the original frame will always arrive ahead of the spoof frame, while in error-passive can be overwritten. To achieve the second kind of effect the error_attack() method may need to be called before using the spoof_frame() method.

timeout (int) Timeout for the process to stop (standard value: 50000000)

overwrite (bool) True if targeted frame is to be overwritten

second (bool) True if spoof frame is supposed to be janus frame

sync_time (int) Number of clock ticks to wait for sync between controllers

split_time (int) Number of clock ticks from start of bit until bit value switch

retries (int) Number of retries after arbitration loss or error

loopback_offset Number of clock ticks to shift the spoof frame (standard value: 93)

Example call for a spoofing attack utilizing the advantages of a previous Error Attack:

```
ch.spoof_frame(overwrite=True)
```

By default, spoof_frame() does not overwrite, retry or send a janus frame. A ValueError is thrown if the main frame (or shadow frame in case of second = True) has not been set previously.

The method leads to two different functions inside canhack.c, depending on whether overwrite is set or not. In the case of a non-overwrite spoofing, the redirection goes to the canhack_spoof_frame() function, which recognizes a frames identifier in the same way the canhack_error_attack() function does it. It then simply calls the canhack_send_frame() or canhack_send_janus_frame function. These will then wait for an arbitration win to send out the frame.

If overwriting the targeted frame is desired, janus frames can not be used, the original sender needs to be in error-passive mode and the overwrite argument has to be set. The method then redirects towards the canhack_spoof_frame_error_passive() function, which also employs the same kind of recognition technique as mentioned before. However instead of sending out the frame, the loopback_offset value is used to shift the spoof frame in a way to aign with the original frame before passing it to the send_bits() function that, in contrary to canhack_send_frame(), does not wait for arbitration. Since the original sender is in error-passive, it will always lose arbitration and so the spoof frame will be the first to be transmitted, while the original one has to wait for another chance at winning arbitraion.

3.2.7 Double Receive Attack

The double_receive_attack() method only has 2 arguments, timeout and repeat. They are set to the same standard values as in the error_attack() method, including the return value concerning the timeout. The corresponding C function is the same as for the Error Attack as well, changes in the inner arguments passed to canhack.c make it so that instead of immediately sending out error frames it waits for the last bit of the EOF. By doing so, the receiving nodes have already verified the frame, but the transmitting node has not marked it as sent. This causes the frame to be sent again and all other nodes receiving it twice or however many times has been specified in the repeat argument.

Example call for Double Receive Attack with 3 additional frames sent:

```
ch.double_receive_attack(repeat=3)
```

3.2.8 Freeze Doom Loop

Very similar to the double receive attack is the freeze_doom_loop() method as it also utilizes the canhack_error_attack() C function with different arguments. This time it sends overload frames instead of error frames and does not return a feedback as to whether the timeout has been reached. The controllers stay in overload recovery mode, which is like error recovery mode, but without an increase of the error counters. This results in the bus basically freezing, since no frames can be sent out while the bus is still filled with overload frames.

Example call for a Freeze Doom Loop attack that stops the bus for the duration of 10 overload frames:

```
ch.freeze_doom_loop(repeat=10)
```

3.2.9 Other non-attacking methods

Get_frame(second) returns a tuple containing all the information inside the can_frame_t specified through the only argument, which will cause the shadow frame to be returned when set to True. It is mostly used inside of other functions to retain information for further processing it. To actually read out a set frame, the method print_frame(second) is more suitable. It prints out the full frame with the fields being color-coded and the stuff bits being printed in red.

Example calls for both methods (main and shadow frame):

```
ch.get_frame()
ch.print_frame(second=True)
```

There is also some hardware diagnostic methods included in the toolkit. The set_can_tx(recessive) method sets the CAN TX pin according to its argument, recessive equaling True and dominant False. Square_wave() works similarly, but alternating the setting of the pin for 160 bits on a 50:50 duty cycle. Using loopback() the CAN RX input can be driven onto the TRIG pin for 160 bit times, after recognizing a falling edge to indicate a transmission start.

Example calls for both methods:

```
ch.get_frame()
ch.print_frame(second=True)
```

Clock related methods include get_clock() and reset_clock(), which work as their name implies, returning the current clock time or setting it to zero respectively.

Example calls for both methods:

```
ch.get_clock()
ch.reset_clock()
```

Another sending method is send_raw(), which outputs the raw bitstream of a set CAN frame onto the TX pin, without waiting for arbitration or other CAN specific functionalities.

Example call for send_raw():

```
ch.send_raw()
```

Finally the last method included is stop(). It is only relevant in multi-threaded environments, as it allows for an attack to be stopped remotely. In this project this method will not be of further importance.

Example call for stop():

```
ch.stop()
```

The CANHack toolkit does not have any methods for decoding received CAN frames or specific fields of those, as this is not necessary for the current functionality. They are however included in the additional CAN SDK software, which is automatically installed when following the standard procedures.

4 Project Goals

After closely analyzing the CANHack toolkit, there are some aspects that may be able to be extended or improved on.

The first is the build process. While it offers a lot of different programs to be added into the finished firmware, it could be more flexible in what to include and by that also faster if one decides to reduce the chosen amount of optional features. In some comments inside the code for example, Dr. Tindell mentions making the MIN tool optional. This idea will be further expanded upon concerning other non-crucial contents.

The CANHack toolkit currently only supports the base version of CAN at a maximum bit rate of 500 kilobits per second. However more commonly used is the updated CAN FD standard, FD standing for Flexible Data-Rate. This protocol can change its transmission speed in the data section to up to 10 megabits per second (Mbps), but most real world applications set the speed to 2 Mbps in the data field and remain at 500 kbps during arbitration. It also allows for data fields of up to 64 bytes. One goal of the project should be to allow the CANHack toolkit to also target frames sent via CAN FD using these bit rate settings. After the expansion most, if not all, attacks should also be able to achieve the same results on CAN FD setups, while keeping their original functionality.

Another possible change would be to investigate the possibility of changing the timing inside C functions like send_bits() to use interrupts instead of pulse width modulation. This might increase the performance of the program and also help with achieving the CAN FD expansion.

Additionally changes will have to be made to the API to accommodate the new CAN FD functionalities. This includes not only specification as to which kind of frame is being targeted, but settings regarding details of the set frames contents as well.

5 CAN FD

The Controller Area Network Flexible Data-Rate (CAN FD) was released in 2012 as a successor to the classical CAN standard, with the goal of allowing for more data inside the frame and faster transmission rates. [AJ17] It keeps the properties of classical CAN in regards to voltage levels and arbitration. This chapter will deal with the new functionalities offered by CAN FD and their addition to the already existing CANHack toolkit.

5.1 Differences To Classical CAN

All information in this section is based on the CAN ISO11898-1:2015 standard [Int15].

5.1.1 Frame Format

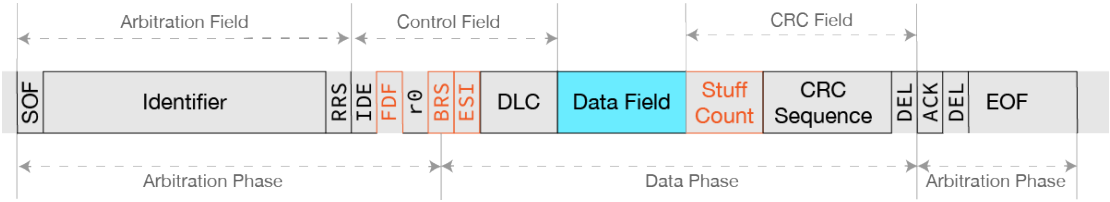

Figure 5.1: CAN FD Frame Base Format [Mic23]

While error and overload frames and their handling has remained the same, the structure of data frames has been subject to many changes, as displayed in figure 5.1. Remote frames have been removed from the CAN FD format, most CAN FD transceivers, however, can still handle classical CAN frames and with that remote frames as well.

The Arbitration field has largely remained the same, with the only changes being the always dominant Remote Request Substitute (RRS) permanently replacing the RTR bit, since remote frames are no longer supported. In extended frames the IDE is once again included in the arbitration field, the RRS is replaced by the SRR bit and moved to the end of the arbitration field. Just like in classical CAN, the SRR is recessive.

Quite some changes happened in the control field, which has been lengthened from a definitive 6 to 8 or 9 bits. It contains 8 bits in extended and 9 bits in non-extended frames, as those include the IDE bit in the control field. Right after it comes the new Flexible Data-Rate Format (FDF) bit. It replaces the r1 in extended and r0 in non-extended formats. Since both of these reserved bits are usually dominant, the recessive FDF allows for a clear indication that the sent frame is one of the CAN FD format. After the FDF, the extended and non-extended bitstreams line up again, starting with r0. In the ISO standard, r0 is now only referred to as the reserved (res) bit, because there are no other reserved bits left. Next is the Bit Rate Switch (BRS). If this bit is set to recessive in a CAN FD frame, starting from it the bit rate of the transmission will be increased by a preset multiplier, 4 in most cases. This will be explained in further detail in the upcoming section 5.1.2. Last before the DLC comes the Error State Indicator (ESI), which has been added to include the current error state of the sender for other nodes on the bus to see. A dominant ESI implies an error-active controller and a recessive one implies the sender to be in error-passive mode. The DLC has received no changes if set to 8 and lower, but the DLC values higher than 8 will now follow a new pattern to allow for much larger values:

Table 5.1: DLC values for CAN FD

Data Bytes	DLC3	DLC2	DLC1	DLC0
0 - 7	Same as CAN (2.1)			
8	1	0	0	0
12	1	0	0	1
16	1	0	1	0
20	1	0	1	1
24	1	1	0	0
32	1	1	0	1
48	1	1	1	0
64	1	1	1	1

As seen in table 5.1 above, the data field now allows for data of up to 64 bytes to be transmitted using a single CAN FD frame. If the DLC value happens to overshoot the actual payload of the data field, it will be filled up in the same way as before. In CAN FD this is a more common occurrence, considering that the DLC values now make large jumps in their implied byte sizes.

Up until the CRC field, which now includes a Stuff Bit Counter (SBC) ahead of the checksum, the stuffing in CAN FD is the same as in regular CAN. The first bit of the CRC field, however, is going to be a so-called fixed stuff bit (FSB) and so will be every fifth bit in the whole field. If the last bit of the data section happens to be a stuff bit, there will not be an additional FSB, but rather it will replace the regular stuff bit. The stuff count will not take these FSBs into consideration. It only has a length of 4 bits, with the first 3 being the gray coded value of the total stuff bits modulo 8 and the last one being a parity bit. This is visualized in table 5.2 down below.

Table 5.2: Gray code for SBC

Stuff Bits % 8	SBC3	SBC2	SBC1	SBC0 (parity)
0	0	0	0	0
1	0	0	1	1
2	0	1	1	0
3	0	1	0	1
4	1	1	0	0
5	1	1	1	1
6	1	0	1	0
7	1	0	0	1

The CRC sequence itself has also received an update, it is now 17 bits long for DLC values of 16 or less and 21 bits for longer data sections. The new polynomials are set as follows:

CRC-17: $x^{17} + x^{16} + x^{14} + x^{13} + x^{11} + x^6 + x^4 + x + 1$

CRC-21: $x^{21} + x^{20} + x^{13} + x^{11} + x^7 + x^4 + x^3 + 1$

Nothing after the CRC Delimiter has changed bitstream-wise, but it also serves as the last bit transmitted at the increased speed that CAN FD offers. Again this will be discussed further in the upcoming speed section.

Overall the frame buildup has undergone quite a few positive changes, like extended data sections and added security through fixed stuff bits and longer CRC sequences. To see how much the protocol benefits from the longer data field, the ratio of data bits to total bits can be calculated for optimal conditions. These conditions are:

- The frames identifier is not extended.
- The frame contains 8 and 64 bytes of data respectively.
- There is no stuff bits (except for the 6 FSBs for CRC-21).

Table 5.3: Optimal frame sizes for CAN and CAN FD

Frame field	Bit amount CAN	Bit amount CAN FD
Arbitration	13	13
Control	6	9
Data	64	512
CRC	16	32
ACK	2	2
EOF	7	7
IFS	3	3
Total:	111	578

Pulling the values for the total and data bits from table 5.3, the following ratios can be calculated:

Classical CAN: $64/111 = 57.\overline{657}\%$

CAN FD: $512/578 \approx 88.581\%$

These calculations show that even though the security and informational value of CAN FD frames has been improved through additional bits in the CRC and control field, the benefits of the lengthened data field still allows for a much better ratio in terms of data contained inside the frame.

5.1.2 Bit Rate

What the previous calculation does not take into account, is that both the data and CRC field are transmitted at much faster bit rates than before. In fact, a CAN FD frame of the same conditions running at a common 4 times speed increase after the BRS would only take 59 classical CAN bit times longer to be transmitted, all while carrying 8 times as much data.

Classical CAN: 111 bit times

CAN FD: $544/4 + 34 = 170$ bit times

This bit rate switch occurs if the BRS bit is set to be recessive, in case of a dominant BRS bit the bus will remain at the arbitration fields bit rate. The increase happens after the BRS bit has been read at the normal sample point. This means the bit time for BRS is different than for both fast and regular bit rate, since the distance between the start of the bit and the sample point has to remain the same, while the time until the ESI bit needs to be cut short. Same goes for the CRC Delimiter, which ends the increase: The sample has to be taken while the bus is still fast, but the time until switching to the ACK bit has to be elongated to fit the regular bit rate.

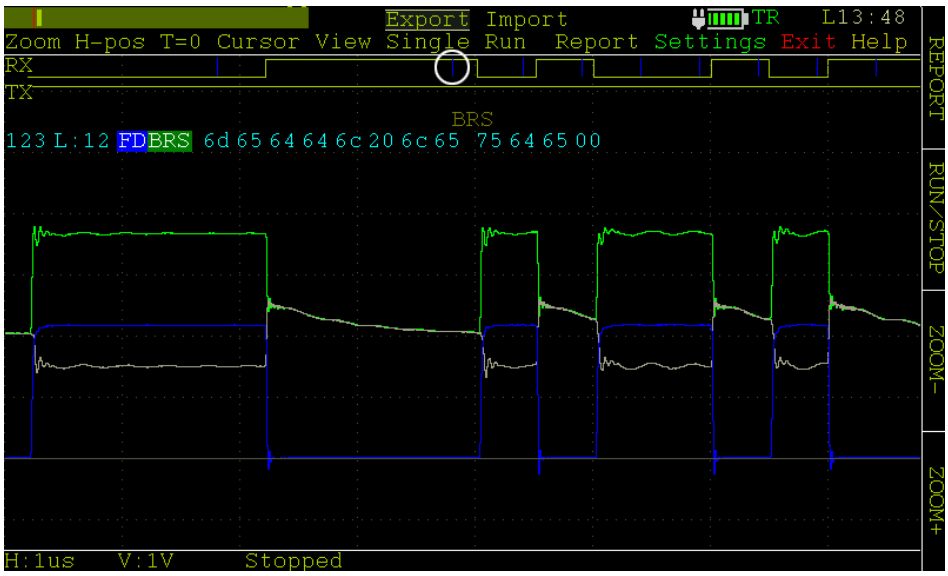

Figure 5.2: Bit Rate Switch as seen on PCAN-Diag FD

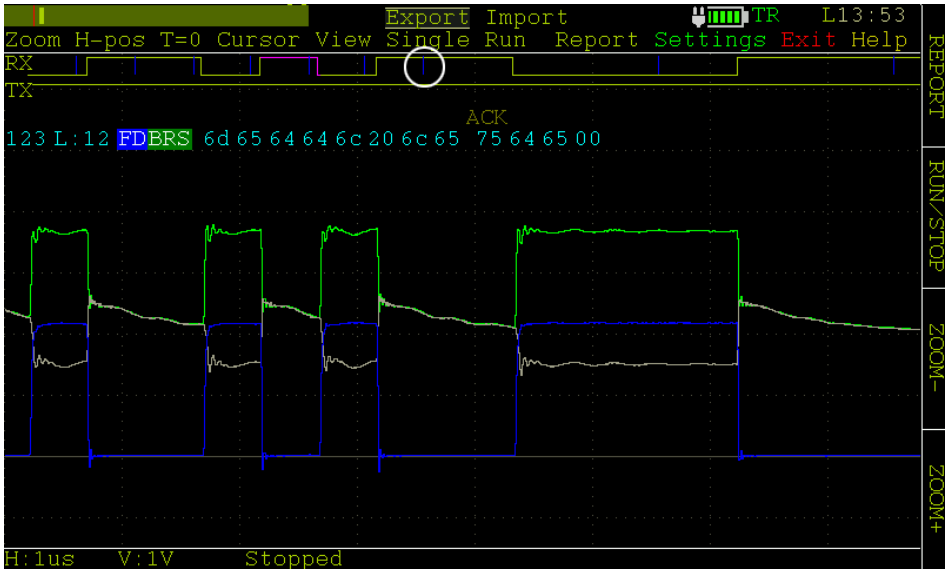

Figure 5.3: CRC Delimiter (right before ACK) as seen on PCAN-Diag FD

The two screenshots 5.2 and 5.3 were captured on the PCAN-Diag FD and show the bit rate switch on a CAN-Bus in action. As previously explained, the BRS and CRC Delimiter have differing lengths from the surrounding bits. Details about the PCANs scope can be found in the appendix A.2.

5.2 Implementation

All changes are being done to a fork of the CANHack project, called canhack_f, publicly available under `https://github.com/dene67/canhack_f`. In this section the basic realization of CAN FD frame support and an increased bit rate will be discussed. To achieve a successful implementation, the functionality of the frame builder and the bus sending operations in the CANHack toolkit will have to be adjusted.

5.2.1 CAN FD Frame Builder

The first and most important piece to implement is of course the CAN FD frame format itself. As a start, the canhack.h header is subject to a few changes, such as increasing the frame length supported by the program.

- Change CANHACK_MAX_BITS from 160 to 700 (should always cover the entire length of a CAN FD frame)
- Change total bit counter tx_bits from unsigned 8 bit to 32 bit integer (max value of uint8_t is 255 -> not enough for CAN FD frames)
- Add boolean fd (True = CAN FD frame) to can_frame_t
- Add unsigned 8 bit integer stuff_count (stores amount of stuff bits, exception: fixed stuff bits) to can_frame_t

The newly added fd variable is to be defined via user input when setting a new frame, so to give the user access to this feature, changes have to be made in both canhack.c and rp2_canhack.c.

canhack.c:

- Add 'bool fd' as argument for canhack_set_frame()
- Store fd argument into can_frame_t

rp2_canhack.c:

- Add 'MP_QSTR_fd' as optional argument in rp2_canhack_set_frame(), affects set_frame() API method
- Parse argument fd
- Add error handling in case both fd and rtr = True
- Pass fd to canhack_set_frame()

Up next are changes to canhack_set_frame(), which is the main function for frame building in canhack.c. New bits have to be added after being introduced to CAN FD.

- Add RRS as replacement for RTR in CAN FD frames
- Add FDF (replaces r1 in extended frames)
- Comment reserved bit 0 (r0) to be res in CAN FD frames
- Add BRS bit
- Add ESI bit

BRS and ESI are also implemented as optional arguments for the API method set_frame() in the same way that the fd argument has been earlier.

The DLC requires no changes in the actual putting of the transmitted value, but rather the calculation of it, since 8 is no longer the maximum. Instead, in rp2_canhack.c the data size needs to be calculated or read from the user input and then encoded with the correct bit pattern according to table 5.1.

- Initialize data array with a length of 64 for CAN FD (instead of 8)
- Fill data array using copy_mp_bytes(), returns length of actual payload
- Adjust error handling to allow for longer data array
- Calculate binary DLC from length of actual payload (stored and calculated using decimal values)

The data field only requires very minor adjustments to be equipped to handle CAN FD frames. The DLC unfortunately can not be used as a length descriptor for its loop anymore. This forces a few changes in the canhack_set_frame() function.

- Recalculate length from passed dlc argument and store in unsigned 8 bit integer len (also account for classical CAN frames)
- Replace dlc with len inside for-loop that puts data into the frame

After the data field comes the CRC field, which now includes the new Stuff Bit Counter and has different polynomials and CRC sequence size. To accommodate for this, the CRC fields part for CAN FD will be completely rewritten and separated from the classical CANs by means of an if-clause. The first component, the SBC, requires some code additions in the earlier stages of the function.

- Initialize stuff_count as 0

- Increment stuff_count when the passed bit is a stuff bit in add_raw_bit() (function adds bits to the bitstream and stores stuff bit positions)

Before setting the SBC, the first fixed stuff bit needs to be added. Some preparation is needed for this, as in case of the last data bit being a stuff bit, it needs to be replaced. Since FSBs are not reflected in the CRC calculation and normal stuff bits are automatically used in the CRC, the replacement has to take place before the start of the CRC field in canhack_set_frame().

- Deactivate stuffing for last data bit
- Add first FSB
- Adjust last_data_bit variable if it would have been a stuff bit
- Decrement stuff_count by 1 (first FSB does not count either way)

Now come the calculation and putting of the Stuff Bit Counters gray code and the second fixed stuff bit, which always comes after the 4 bit long SBC.

- Calculate stuff_count modulo 8
- Create switch-case to set 3 bit gray code
- Get parity by masking least significant bit of stuff_count
- Add into frame using for-loop
- Add second FSB

Prior to adding the CRC to the frame, it of course needs to be calculated. Since there are two new sizes and polynomials, two new functions will be introduced. Also, when extending the frame with the CRC, the rest of the FSBs need to be kept in mind. They can not be hard coded the same way the first two are, because, while they are consistent in their spacing, the amount varies based on the CRCs length.

- Calculate unsigned 8 bit integer crc_len based on dlc
- Create functions do_crc17() and do_crc21()
- Add 'uint8-t dlc' as argument of add_bit() (function checks for necessity of CRC and stuffing, then redirects to do_crc() and add_raw_bit())
- Implement do_crc17() and do_crc21() into add_bit()
- Use dlc and fd variables to redirect towards the correct CRC function
- Add into frame using for-loop with length crc_len
- Add FSB at every fourth loop

After the CRC field, the classical CAN and CAN FD frames show no difference, so everything inside canhack_set_frame() beyond this point remains the same.

5.2.2 Send CAN FD Frames

The next due changes when implementing CAN FD regard the new increased bit rate. Currently the maximum data transmission rate of the CANHack toolkit sits at 500 kbps. So, to increase it to 2 Mbps, the bit time of about 250 would have to be divided by 4, which is not evenly possible and since there are no half cycles, the clock frequency will have to be changed.

During development multiple frequencies have been tested. The highest possible frequency without overclocking is 133 MHz, but the highest practical one is 130 due to calculation of bit times. This frequency had proven to be too slow, since the bit time of 65 cycles in the fast data section of CAN FD is not long enough to execute the code contained in the send_bits() function. So the next step was to overclock the device and testing the limits of this technique. Frequencies higher than 300 MHz caused the Pico to not boot correctly, so the one that was finally settled on was 250 MHz. It is fast enough to support the send_bits() functionality at CAN FD speeds and also allows for easy calculations, as it is just double the regular clock frequency.

- Overclock device to 250 MHz in the main.c (executed at boot)
- Calculate new timings and add them to rp2_canhack.h
  - Change BIT_TIME from 249 to 499
  - Add BIT_TIME_FD as 125
  - Change SAMPLE_POINT_OFFSET from 150 to 300
  - Add SAMPLE_POINT_OFFSET_FD as 75
  - Change DEFAULT_LOOPBACK_OFFSET from 93 to 186
  - Add SAMPLE_TO_BIT_FD as BIT_TIME_FD - SAMPLE_POINT_OFFSET_FD (50)
  - Change FALLING_EDGE_RECALIBRATE from 31 to 62
- Add boolean brs (True = bit rate increased) to can_frame_t, so send_bits() knows whether to switch bit rates
- Add unsigned 32 bit integer brs_bit (stores position of the brs bit) to can_frame_t, so send_bits() knows when to switch bit rates

After preparing everything, now comes the adjustment of send_bits(). There was multiple ideas regarding the realization of the bit rate switch. The method that had proven to b ethe most time-effective and least complicated was including two code blocks, which would turn fast data mode on and off respectively. Since the bit time in send_bits() is constant right now and the counter for the bit setting is too small, a few changes need to be made to the arguments and variables.

- Add unsigned 16 bit integer cur_bit_time and initialize as BIT_TIME
- Change tx_index argument from uint8_t to uint16_t (uint8_t max value is 255, CAN FD frames can be over 600 bits long, though)
- Change ADVANCE() arguments from BIT_TIME to cur_bit_time

Now that the bit time is a variable, the process for changing it can be implemented. To ensure the BRS bit and CRC delimiter are sent before the switch, the code blocks are placed behind the SET_CAN_TX() command. Also the tx_index is always 1 position higher than the current tx value to be put on the bus, since the variable is incremented right after reading off the bitstream. This needs to be kept in mind when deciding on the correct bit to switch the bit rate on. First up is the increase, which requires the bit time to be reduced and the upcoming bit end and sample point to be adjusted after hitting the set BRS bit.

- Add if-clause thats executed when tx_index is at brs + 1 and the current tx is 1 (=> current bit is BRS and it is set)
- Set cur_bit_time to BIT_TIME_FD
- Reduce bit_end by SAMPLE_TO_BIT_END_FD
- Set sample_point to bit_end - SAMPLE_TO_BIT_END_FD

The reason for reducing bit_end by the value SAMPLE_TO_BIT_END_FD is explained through the timing of the sample points. The function always samples at 60% of the total bit time, so at 300 clock cycles if the bit start is assumed to be 0. Since the frame is transmitted at a much higher bit rate after the BRS, the distance from sample_point to bit_end is reduced from 200 to just 50 (SAMPLE_TO_BIT_END -> SAMPLE_TO_BIT_END_FD), which would put bit_end at 350 cycles. Most CAN transceivers, however, use 80% of the total bit time, equaling to 400 clock cycles. If bit_end is set earlier than that, some controllers might miss the BRS bit. To counteract this, bit_end is set to 450 clock cycles after start by reducing the variable by 50. The sample_point variable is then also adjusted to the distance it uses in fast data mode.

The code block for setting the bit rate back to the original state has very similar requirements. The speed reduction takes place when the CRC delimiter is hit, however this time it does not depend on the state of the bus, but rather on whether the BRS bit had been set previously. This can be read out of the used frames properties.

- Add if-clause thats executed when tx_index is at brs + 1 and the current tx is 1 (=> current bit is BRS and it is set)
- Set cur_bit_time to BIT_TIME
- Extend bit_end by (SAMPLE_TO_BIT_END - SAMPLE_TO_BIT_END_FD)
- Set sample_point to bit_end - SAMPLE_TO_BIT_END

Realignment of bit_end and sample_point in this block has the same reasons as the previous. It just reverses the effect by adding to the bit_end variable and then once more subtracting the desired distance from sample_point.

The rest of the function can be left alone as it only checks for termination requirements. To fully support sending CAN FD frames, however, there is one small change to canhack_send_frame() left, the function that is typically used to call send_bits(). In there, the tx_bits variable also has to be made into a 16 bit integer to match up with the argument passed to send_bits().

5.3 Restoring Original Functionality

Now that the toolkit is able to both create and send viable CAN FD frames, it is time to adjust the rest of the API methods and their corresponding C functions in a way that allows them to support CAN FD frames as well. The goal is also to keep the impact on the original functionality with classical CAN frames to an absolute minimum.

5.3.1 No Change Required

There is some functions that do not require changes or automatically work through previous actions, like changing CANHACK_MAX_BITS. These are mainly hardware diagnostic and printing functions, a list of them is found below:

- get_frame()
- set_can_tx()
- square_wave()
- get_clock()
- reset_clock()
- send_raw()
- print_frame()

- stop()

Of course the different bit rate settings when initializing the CANHack class in MicroPython are also still available, as they only affect the PWM used for timing and not the constants regarding bit times etc. Their respective bit rate increases are:

500: 2 megabits per second (standard)

250: 1 megabit per second

125: 500 kilobits per second

5.3.2 Janus Attack

The Janus Attack function send_janus_bits() inside canhack.c is subject to very similar changes as the send_bits() function, with the exception of two new variables having to be established to allow for the correct timings. The regular sync and split time are passed to the function as an argument that is ultimately entered by the user, an option that should be kept for these new times set for CAN FD.

canhack.c:

- Add 'ctr_t sync_time_fd' as argument in send_janus_bits() and canhack_send_janus_frame()
- Add 'ctr_t split_time_fd' as argument in send_janus_bits() and canhack_send_janus_frame()
- Add both arguments to the send_janus_bits() call in canhack_send_janus_frame()

The crt_t type is predefined in the rp2_canhack.h as a uint16_t and serves as the timer size.

rp2_canhack.c:

- Add 'MP_QSTR_sync_time_fd' as optional argument in rp2_canhack_send_janus_frame()
- Add 'MP_QSTR_split_time_fd' as optional argument in rp2_canhack_send_janus_frame()
- Parse arguments sync_time_fd and split_time_fd
- Pass both to canhack_send_janus_frame()

In addition to the new variables, it is also necessity to expand the error handling of rp2_canhack_send_janus_frame. A CAN FD Janus Attack cannot succeed, when the BRS bits of both frames do not perfectly align in both position and value. The main and shadow frame need to be transmitted at the same bit rate at all times, due to the makeup of the send_janus_bits() function.

- Add error handling in case of different brs in frame1 and frame2
- Add error handling in case of different brs_bit in frame1 and frame2

Now it is time to implement similar code blocks as in send_bits(). This time, however, two types of code blocks need to be implemented. The first one in the sync phase of the Janus Attack switches the current bit time and the bit end in the same way as known from section 5.2.2, with the sample points adjustment replaced by one regarding the sync time.

- Add code blocks from send_bits() in sync phase
- Delete sample point adjustment
- **BRS:** Set sync_end to bit_end + sync_time_fd
- **CRC Delimiter:** Set sync_end to bit_end + sync_time

Lastly another type of code block is added to the split phase of the Janus Attack. This one only switches the split end under the same conditions as the first two code blocks. The switch of the split_end value is done in separates code block, because the split end of the BRS bit needs to be hit at the regular bit rate. A similar reason as to the one given for the bit end adjustment when implementing CAN FD into send_bits().

- **BRS:** Set split_end to bit_end + split_time_fd
- **CRC Delimiter:** Set split_end to bit_end + split_time

Although succesfully implemented, the send_janus_frame() API call might still produce mixed results. As mentioned in the beginning of the section 5.2.2 the short bit time of 65 clock ticks when using 130 MHz on the Pico was not fast enough to reliably handle send_bits() functionality. The time between a sync and split end is now only 47 cycles with base settings. This, of course, caused inconsistency with the send_janus_bits() function that may cause some Janus frames to be flagged with error frames by other nodes, because the timing does not match up and causes things like a faulty CRC or stuff rule violations.

5.3.3 Loopback

The loopback() method passes back the CAN RX value to the TRIG pin. While it does use the BIT_TIME constant for timing, it does not use it to read out specific singular values, but rather to time the end of the function. This works by counting down from 160 to 0, decrementing by 1 every BIT_TIME. To support CAN FD, all that needs to be done is extend the duration of this cycle in case a CAN FD frame is to be read. The length should account for the worst case, which is the maximum frame length combined with no bit rate switching.

canhack.c:

- Add 'bool fd' as argument in canhack_loopback() (True = CAN FD frame)
- Add if-clause to set uint i = 700 when fd = true

rp2_canhack.c:

- Add 'MP_QSTR_fd' as optional argument in rp2_canhack_loopback()
- Parse argument fd
- Pass fd to canhack_loopback()

5.3.4 Attacks

The most important piece of the attacking functions is the attack_parameters struct contained inside the canhack struct. They are made up of the following variables:

- uint64_t bitstream_mask
- uint64_t bitstream_match
- uint32_t n_frame_match_bits
- uint32_t n_frame_match_bits_cntdn
- uint32_t attack_cntdn
- uint32_t dominant_bit_cntdn

The important ones are the first 3 variables. They are set in a separate function named canhack_set_attack_masks() at the bottom of canhack.c, which is usually called by the rp2_canhack.c functions of the attacks before executing them. It has no argument, but uses the values stored in the canhack structs frames for calculation. The attack_parameters are set like this:

- Calculate n_frame_match_bits as the length of the arbitration field (last_arbitration_bit+1)

- Set bitstream_mask as binary number with the least significant $n_{frame\_match\_bits}$+10 bits as 1
- Set bitstream_match as arbitration field with 10 leading 1s (minimum distance between frames)

This works for classical frames, but causes the problem of no possible distinguishing between them and CAN FD arbitration phases. Luckily the FDF bit, which indicates whether a frame is a CAN FD frame, sits just outside the arbitration field. So by increasing the length of n_frame_match_bits by 1, the attack parameters can now definitively tell the difference between a classical and CAN FD frame.

For the spoof_frame() method this is all that needs to be implemented. The LOOPBACK_OFFSET can remain at the original value for 250 MHz and does not have to be adjusted to faster bit rates, because the adjustment of the spoof frames timing to fit the original frame still takes place before the BRS bit. Other than that, the canhack_spoof_frame() and canhack_spoof_frame_error_passive() C functions, that the API method leads to, use the already implemented send_bits() and send_janus_bits functions to put the spoof frame onto the bus.

The canhack_error_attack() function; final destination of the error_attack(), freeze_doom_loop() and double_receive_attack() API methods; has to be adjusted regarding the latter two attack types. The Error Attack only affects the arbitration and then targets the error delimiter of the error frame, both of these are running at a non-increased bit rate.

The Freeze Doom Loop and Double Receive Attack, however, target the EOF. While the EOF is also back to regular speed, the space inbetween it and the control field is not. The canhack_error_attack() in this regard works by sampling values every BIT_TIME, waiting for at least 6 consecutive equal bit values to indicate the EOF. But what if, by chance, inside the fast data section every fourth bit happens to be equal? This would cause the function to prematurely send out error or overload frames. Though the brs_bit value stored in the frame passed to the function does give information about the start of the increased bit rate, there is no method in CANHack for decoding CAN frames and so no way exists to check for the end of it. To solve this issue, the bit rate is switched as normally once the BRS bit hits and the amount of consecutive bit values of the same type is multiplied by 4. A regular bit sampled at 4 times the bit rate should produce 4 equal values, so, for example, the EOF of 7 recessive bits will now be 28 recessive 'bits'. This workaround allows for both Freeze Doom Loop and Double Receive Attack to be performed on CAN FD frames with BRS enabled.

6 Other Changes

Besides the changes in functionality, the CANHack toolkit has also been altered in its API and building process. While changes in the API go hand in hand with those in functionality, the goal of altering the build process is to allow for more flexibility and faster building times when choosing less optional features.

6.1 API

Some methods that can be called in the API have received new optional arguments related to CAN FD.

set_frame():

fd (bool) True if CAN FD frame

brs (bool) True if 4x bit rate is applied for data and CRC field

esi (bool) True if frame indicates node to be in error-active mode (False = error-passive)

The fd argument is set to False if not otherwise specified, while the other two are True by default, as most CAN FD frames do utilize the increased bit rate and there is no real advantage to sending an error-passive frame on purpose.

Example call:

```
ch.set_frame(0x123, data=bytes([0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
0x06, 0x07, 0x08, 0x09]), fd=True, esi=False, brs=False)
```

This command sets an error-passive CAN FD frame with the identifier 123_{16}, the data of numbers 0_{16} through 9_{16} and a DLC of 12. It does not switch its bit rate, because the BRS bit will be set to dominant.

loopback():

fd (bool) True if reading CAN FD frame

The standard value of this fd argument is False.

Example call:

```
ch.loopback(fd=True)
```

This command drives the TRIG pin to the same value as the CAN RX pin for 700 bit times.

send_janus_frame():

sync_time_fd (int) Number of clock ticks to wait for sync between controllers after BRS

split_time_fd (int) Number of clock ticks from start of bit until bit value switch after BRS

Like outside of the fast bit rate section of the frame, the split and sync times sit at about 25% and 62.5% of the current bit time if not otherwise specified. This means they are set to the rounded values 31 and 78 by default.

Example call:

```
ch.send_janus_frame(sync_time_fd=25, split_time_fd=100)
```

This command starts a Janus Attack using the previously set frames, changing the sync and split time after a potential Bit Rate Switch to 20% and 80% of the frame.

spoof_frame():

sync_time_fd (int) Number of clock ticks to wait for sync between controllers after BRS

split_time_fd (int) Number of clock ticks from start of bit until bit value switch after BRS

The spoof_frame() method also offers to send Janus frames as spoof frames when overwrite is not set to True, so the timing arguments from send_janus_frame() also have to be added here. The standard values remain the same.

Example call:

```
ch.spoof_frame(sync_time_fd=25, split_time_fd=100, second=True)
```

This command spoofs a frame with the previosly set identfier using a Janus frame compromised of the two frames stored in CANHack, with its sync and split times changed to 20% and 80% after a Bit Rate Switch.

6.2 Build Process

The aim of modifying the CANHack toolkits build process is to offer more flexibility and speed by making features optional that were previously required. This is mainly affects the two repositories that have to be cloned into the MicroPython and CANHack folders, MIN and candrivers (Canis CAN SDK). Leaving out these parts of the program will not diminish the functionality of the CANHack methods, but speed up the downloading and building process by a lot.

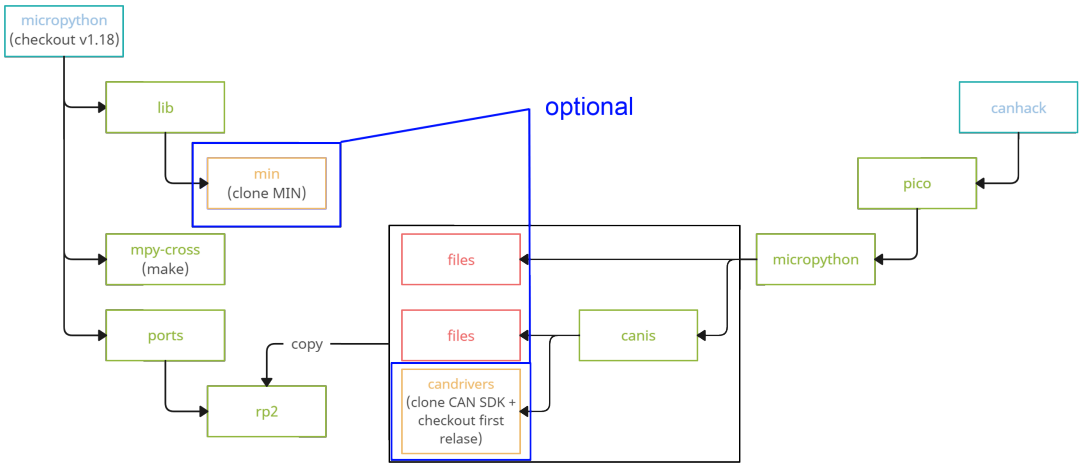

Figure 6.1: Project structure after modification

Most changes that were made to achieve a successful build under these new conditions were made in the CMakeLists.txt. While the original compiler options were all kept, ther was two new additions. One of them is the CANHACK option, which already existed inside the CMakeLists.txt, but now it stands alone and can be entered by the user. That means the cloning of the Canis CAN SDK repository is no longer necessary to run CANHack.

The other new compiler option added in RP2MIN, which adds Dr. Ken Tindells MIN project to the installation, if it is downloaded as in the previous building guide. This new RP2MIN option also solves the issue of MIN being unconditionally included: Without setting -DRP2MIN=1 it will not be added to the firmware.uf2. All changes to the CMakeLists.txt can be found in colored-diff in the appendix A.3.

Regarding the new conditional MIN support, modrp2.c and main.c also needed to be modified. The adjustments here are limited to #ifdef-blocks placed around any mention of MIN-related functions or imports.

6.3 Interrupts

Another potential change to be investigated is the employment of hardware interrupts to perform more efficient bit-banging. The Raspberry Pico offers multiple different kinds of IRQs that could be used for bit-banging purposes. However, in the Raspberry Pi Pico C/C++ SDK documentation it says the following on pages 30 and 31 [Ras23]:

> [...] for slower protocols you might be able to use an IRQ to wake up the processor from what it was doing fast enough (though latency here is a concern) to send the next bit(s). [...] Above certain speeds — say a factor of 1000 below the processor clock speed — IRQs become impractical, in part due to the timing uncertainty of actually entering an interrupt handler. The alternative when ”bit-banging” is to sit the processor in a carefully timed loop, [...]

According to this excerpt, Interrupt Requests are a less precise method of bit-banging for our purposes, even recommending a method similar to the currently employed. This is because one thousandth of the overclocked Pico frequency 250 MHz would only come out to 250 kilohertz (kHz), far too slow for the CAN Bus implementation, which runs at a bit rate of up to 500 kbps or even 2 **mbps!** (**mbps!**) in case of CAN FD. In conclusion, interrupts might free up some more processor time when working with slower protocols, but they are not suited to replace the timing methods used in the CANHack toolkit.

7 Summary and Outlook

The thesis started out with lots of research about CAN Bus and the CANHack toolkit. Before even thinking about editing anything, SocketCAN had to be learned to enable for testing. After setting up both software and hardware according to the documentation, the first CANHack attacks were performed. They helped a lot in gaining understanding of CAN-specific functionalities, such as the error counters or what makes and breaks arbitration.

Once all features of CANHack were discovered it was time to dive into the inner workings of the program. To understand how the MicroPython usermod system works, the first change made to the toolkit was a testing function with little functionality, but it was a great way of becoming aware of how the API methods and their arguments were routed through the different files and ended up being processed in C. This, however, was only the beginning. Next was analyzing the different C functions and how exactly they affect the bus and its nodes, and with that came the realization that the implementation of the newer CAN FD standard would be possible.

Arriving at the biggest part of the thesis, it started by taking an even closer look at the ISO publication and the two standards differences. Many new bit types and fields had to be calculated and added to the frame building process, expanding the length of the code dedicated to this task from a little under 200 to more than 400 lines. Adding this much code to an already existing program of course came with some challenges. The calculation of DLC values was one of those. Since the intention was to make the code for this as short as possible and avoid a switch-case, mathematical solutions had to be come up with. Another challenge was the overrunning of variables. After successfully implementing all the new frame contents, large data sizes suddenly caused several informational pieces of the can_frame_t structure to change into unrealistic values. This was solved by adding 'poison' variables inbetween each value inside the struct. Running the frame builder and checking the values of the 'poisons' allowed for identification of the source of the issues.

When building frames was thought to no longer a problem, up came the realization of the bit rate increase, warranting a deeper look into the capabilities of the Raspberry Pico. It became clear rather quickly that the base frequency of 125 MHz and even the heightened frequency of 130 MHz were not enough to handle the new CAN FD bit rates. At this point it was unclear whether the Pico could even be overclocked to a frequency at which a realization of the bit rate switch was even possible. Finding a frequency went hand in hand with writing code that was short enough to still be supported in only about half the time previously intended for bit-banging, all the while checking for sudden bit rate switches and recalculating timings. Eventually a proper frequency was found, but sending frames exposed some mistakes made in the previous section, like wrong gray code in the SBC. After going back and forth between frame building and sending code for a while, the CANHack toolkits first ever valid CAN FD frame was bit-banged.

But this was not a point to stop, as all other functionality still had yet to be adjusted for working with CAN FD frames and speeds. Some of those were quickly handled, others not. Especially the Janus Attack was a major challenge, due to the little time inbetween value switches, even shorter than what was deemed too short previously. While close to the end of the project the functionality could finally be implemented, it remains to be improved and made more stable. All other API methods, though, were able be implemented and gained additional arguments to help the user specify the properties of attacks targeting CAN FD frames just as much as with classical CAN.

Whilst being the most notable, the implementation of CAN FD was not the only task of this project. In order to allow for the CANHack toolkit to be more accessible the software build process has been adjusted as well, making it more flexible and by that simpler and faster under the right conditions. To increase overall performance, the usage of IRQs was also analyzed, but decided to not be suited for this kind of bit-banging software.

This thesis ends on the note of new possibilities in the field of automotive security. These types of attacks have been long known to work on classical CAN and can now also be tested on a newer and more widely used standard for the first time. With the knowledge of new weaknesses comes the ability to fix them, ultimately enabling for a more secure experience of automotive products.

A Appendix

A.1 Build Instructions

```
To build the firmware:

0. Get the tools
----------------

Use the latest gcc cross compiler supplied by Arm. See the blog post:

https://kentindell.github.io/2022/07/26/canpico-c-debug/

for a description of getting the C cross compiler from Arm.

1. Get the MicroPython source
-----------------------------

$ git clone http://github.com/micropython/micropython.git
$ cd micropython
$ git checkout v1.18
$ git submodule update --init

2. Add MIN
----------

$ cd lib
$ git clone https://github.com/min-protocol/min.git
$ cd ..

3. Build mpy-cross
------------------

$ cd mpy-cross
$ make
$ cd ..

4. Check this has worked by making the stock Pico firmware
```

```
------------------------------------------------------------

$ cd ports/rp2/
$ cmake CMakeLists.txt
$ make

6. Patch the firmware
---------------------

Copy the file hierarchy in this directory to ports/rp2 in the MicroPython build

CMakeLists.txt
machine_pin.c
modrp2.c
tusb_config.h
tusb_port.c
main.c
mpconfigport.h
canis/
canhack.h
canhack.c
common.h
common.c
rp2_min.h
rp2_min.c
rp2_can.h
rp2_can.c
rp2_canhack.h
rp2_canhack.c

7. Add the Canis CAN SDK
------------------------
$ cd canis
$ git clone https://github.com/kentindell/canis-can-sdk candrivers
$ cd ..

7. Re-build the firmware
---------------------
To do a proper 'clean' between builds, use:

$ rm -rf Makefile CMakeFiles CMakeCache.txt genhdr generated frozen_content.c p

To make the firmware for the CANPico and CANHack boards:
```

```
$ cmake CMakeLists -DCAN=1 -DCANPICO=1
$ make

For the CHV DEF CON 30 badge:

$ cmake CMakeLists -DCAN=1 -DCHV_DEFCON30_BADGE=1
$ make

The firmware is produced in firmware.uf2. Program the Pico with this firmware
in the normal way:

- Press and hold the boot button while powering on the Pico
- Wait until the board mounts as a removable disk
- Copy the firmware into the disk
- Power off the board and power it on again
```

A.2 PCAN scope

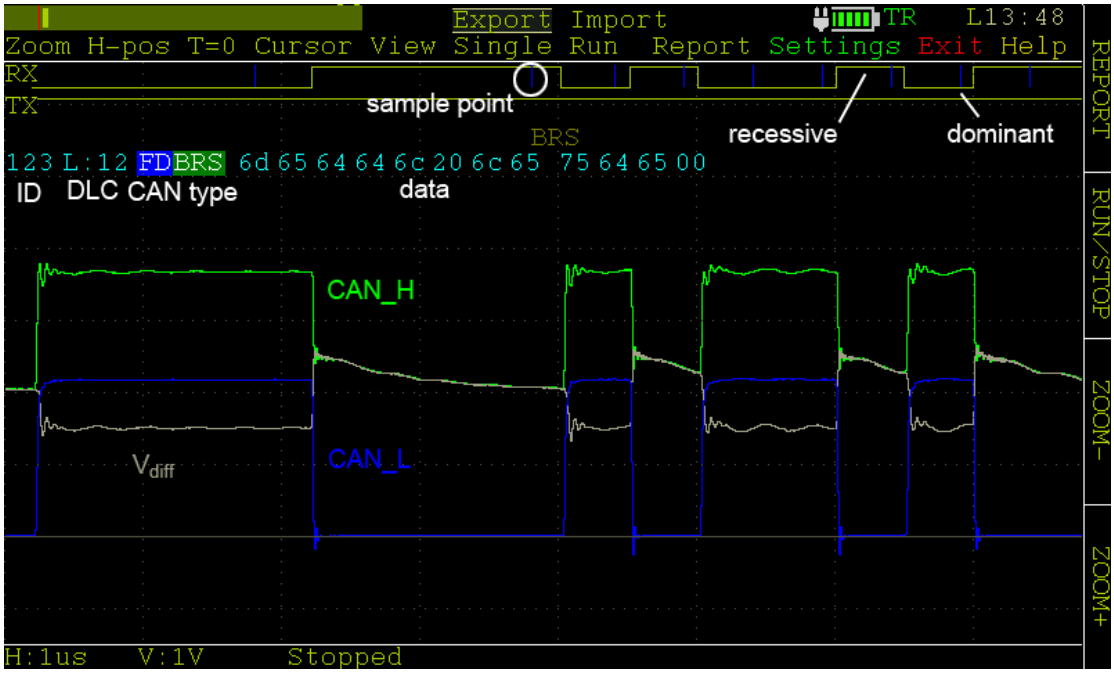

Figure A.1: PCAN-Diag Scope explained

The grey dotted lines in figure A.1 symbolize 1 microsecond in horizontal and 1V in vertical orientation. No CAN type means classical CAN.

A.3 CMakeLists.txt Changes

```
diff --git a/pico/micropython/CMakeLists.txt b/pico/micropython/CMakeLists.txt
index cea3125..2f84f62 100644
--- a/pico/micropython/CMakeLists.txt
+++ b/pico/micropython/CMakeLists.txt
@@ -93,14 +93,10 @@ set(CRYPTOCAN_SOURCE_LIB
$CC_CRYPTOCAN_PATH/init.c
$CC_CRYPTOCAN_PATH/tx.c
$CC_CRYPTOCAN_PATH/rx.c
)

-set(MIN_SOURCE_LIB
-    $MICROPY_DIR/lib/min/target/min.c
-)
-
set(MICROPY_SOURCE_LIB
$MICROPY_DIR/lib/littlefs/lfs1.c
$MICROPY_DIR/lib/littlefs/lfs1_util.c
$MICROPY_DIR/lib/littlefs/lfs2.c
$MICROPY_DIR/lib/littlefs/lfs2_util.c
@@ -144,11 +140,10 @@ set(MICROPY_SOURCE_PORT
rp2_flash.c
rp2_pio.c
tusb_port.c
uart.c
canis/common.c
-    canis/rp2_min.c
# TODO add Bullseye CCover support for code coverage measurement
#   /opt/BullseyeCoverage/run/libcov-memoryDump.c
)

set(MICROPY_SOURCE_QSTR
@@ -170,11 +165,10 @@ set(MICROPY_SOURCE_QSTR
$PROJECT_SOURCE_DIR/modrp2.c
$PROJECT_SOURCE_DIR/moduos.c
$PROJECT_SOURCE_DIR/modutime.c
$PROJECT_SOURCE_DIR/rp2_flash.c
$PROJECT_SOURCE_DIR/rp2_pio.c
-    $PROJECT_SOURCE_DIR/canis/rp2_min.c  # TODO make this conditionally includ
)

set(PICO_SDK_COMPONENTS
```

```
hardware_adc
hardware_base
@@ -220,10 +214,20 @@ if(CAN OR CRYPTOCAN)
$PROJECT_SOURCE_DIR/canis/rp2_canhack.c
$PROJECT_SOURCE_DIR/canis/rp2_can.c
)
endif()

+if (CANHACK)
+    list(APPEND MICROPY_SOURCE_PORT
+        $PROJECT_SOURCE_DIR/canis/canhack.c
+        $PROJECT_SOURCE_DIR/canis/rp2_canhack.c
+    )
+    list(APPEND MICROPY_SOURCE_QSTR
+        $PROJECT_SOURCE_DIR/canis/rp2_canhack.c
+    )
+endif()
+
if(CRYPTOCAN)
list(APPEND MICROPY_SOURCE_PORT
$HSM_SOURCE_LIB
$CRYPTOCAN_SOURCE_LIB
$PROJECT_SOURCE_DIR/canis/rp2_hsm.c
@@ -233,10 +237,19 @@ if(CRYPTOCAN)
$PROJECT_SOURCE_DIR/canis/rp2_hsm.c
$PROJECT_SOURCE_DIR/canis/rp2_cryptocan.c
)
endif()

+if (RP2MIN)
+    list(APPEND MICROPY_SOURCE_PORT
+        $PROJECT_SOURCE_DIR/canis/rp2_min.c
+    )
+    list(APPEND MICROPY_SOURCE_QSTR
+        $PROJECT_SOURCE_DIR/canis/rp2_min.c
+    )
+endif()
+
if(MICROPY_PY_BLUETOOTH)
list(APPEND MICROPY_SOURCE_PORT mpbthciport.c)
target_compile_definitions($MICROPY_TARGET PRIVATE
MICROPY_PY_BLUETOOTH=1
MICROPY_PY_BLUETOOTH_USE_SYNC_EVENTS=1
@@ -297,11 +310,10 @@ target_sources($MICROPY_TARGET PRIVATE
```

```
$MICROPY_SOURCE_PY
$MICROPY_SOURCE_EXTMOD
$MICROPY_SOURCE_LIB
$MICROPY_SOURCE_DRIVERS
$MICROPY_SOURCE_PORT
-    $MIN_SOURCE_LIB
)

target_link_libraries($MICROPY_TARGET usermod)

target_include_directories($MICROPY_TARGET PRIVATE
@@ -351,19 +363,21 @@ target_compile_definitions($MICROPY_TARGET PRIVATE
# Definitions for various targets
if (CRYPTOCAN)
target_compile_definitions($MICROPY_TARGET PRIVATE
CAN=1
CRYPTOCAN=1
+        CANHACK=1
SM_CPU_LITTLE_ENDIAN=1
SM_RAM_TABLES=1
SM_KEY_EXPANSION_CACHED=1
SM_CODE_IN_RAM=1
)
endif()
if (CAN OR CRYPTOCAN)
target_compile_definitions($MICROPY_TARGET PRIVATE
CAN=1
+        CANHACK=1
MCP2517FD=1
HOST_CANPICO=1
)
endif()
if (CHV_DEFCON30_BADGE)
@@ -379,10 +393,15 @@ endif()
if (CANHACK)
target_compile_definitions($MICROPY_TARGET PRIVATE
CANHACK=1
)
endif()
+if (RP2MIN)
+    target_compile_definitions($MICROPY_TARGET PRIVATE
+        RP2MIN=1
+    )
+endif()
```

```
target_link_libraries($MICROPY_TARGET
$PICO_SDK_COMPONENTS
)
```

Bibliography

[AJ17] Hyun Su An und Jae Wook Jeon. „Analysis of CAN FD to CAN message routing method for CAN FD and CAN gateway“. In: *2017 17th International Conference on Control, Automation and Systems (ICCAS)*. 2017, Seiten 528–533. DOI: 10.23919/ICCAS.2017.8204292.

[AMB20] Reza Alaei, Payman Moallem und Ali Bohlooli. „Statistical based algorithm for reducing bit stuffing in the Controller Area Networks“. In: *Microelectronics Journal* 101 (2020), Seite 104794. ISSN: 0026-2692. DOI: https://doi.org/10.1016/j.mejo.2020.104794. URL: https://www.sciencedirect.com/science/article/pii/S0026269219303520.

[Blo21] Gedare Bloom. „WeepingCAN: A stealthy CAN bus-off attack“. In: *Workshop on Automotive and Autonomous Vehicle Security*. 2021.

[CF07] JA Cook und JS Freudenberg. „Controller area network (CAN)“. In: *EECS* 461 (2007), Seiten 1–5.

[Ele22] CSS Electronics. *CAN bus Errors Explained - A Simple Intro [2022]*. en. Mai 2022. URL: https://www.csselectronics.com/pages/can-bus-errors-intro-tutorial.

[HPL02] Steve Corrigan HPL. „Introduction to the controller area network (CAN)“. In: *Application Report SLOA101* (2002), Seiten 1–17.

[IMP15] Kristian Ismail, Aam Muharam und Mulia Pratama. „Design of CAN Bus for Research Applications Purpose Hybrid Electric Vehicle Using ARM Microcontroller“. In: *Energy Procedia* 68 (2015). 2nd International Conference on Sustainable Energy Engineering and Application (ICSEEA) 2014 Sustainable Energy for Green Mobility, Seiten 288–296. ISSN: 1876-6102. DOI: https://doi.org/10.1016/j.egypro.2015.03.258. URL: https://www.sciencedirect.com/science/article/pii/S1876610215005640.

[Int15] International Organization for Standardization. *Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling*. Standard. Geneva, CH: International Organization for Standardization, Dez. 2015.

[ISO21] ISO. Juni 2021. URL: https://www.iso.org/standard/63648.html.

[LXQ12] Guang Liu, Ai-ping Xiao und Hua Qian. „Communication system design based on TMS320F2407 with CAN Bus“. In: *AASRI Procedia* 3 (2012), Seiten 463–467.

[Mic23] MicroControl. *CAN FD - MicroControl*. März 2023. URL: https://www.microcontrol.net/en/service/basics/can-fd/.

[OS01] M. van Osch und S.A. Smolka. „Finite-state analysis of the CAN bus protocol“. In: *Proceedings Sixth IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking*. 2001, Seiten 42–52. DOI: 10.1109/HASE.2001.966806.

[Ras23] Raspberry Pi Ltd. *Raspberry Pi Pico C/C++ SDK*. Version a6fe703-clean. Raspberry Pi Ltd. 2023. URL: https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf.

[Vos19] Wilfried Voss. *CAN Bus And SAE J1939 Bus Voltage*. Juni 2019. URL: https://copperhilltech.com/blog/can-bus-and-sae-j1939-bus-voltage/.

List of Tables

List of Figures

List of Abbreviations

ACK Acknowledgement

API Application Programming Interface

BRS Bit Rate Switch

CAN Controller Area Network

CAN FD Controller Area Network Flexible Data-Rate

CEO Chief Executive Officer

CRC Cyclic Redundancy Check

DLC Data Length Code

EOF End Of Frame

ESI Error State Indicator

FDF Flexible Data-Rate Format

FSB fixed stuff bit

GND Ground

GP General Purpose (Input/Output)

ID identifier

IDE Identifier Extension

IFS Intermission Frame Space

IRQ Interrupt Request

ISO International Organization of Standardization

kbps kilobits per second

kHz kilohertz

Mbps megabits per second

MHz megahertz

MIN Microcontroller Interconnect Network

PWM pulse width modulation

r0 reserved bit 0

r1 reserved bit 1

REC Receive Error Counter

res reserved

RRS Remote Request Substitute

RTR Remote Transmission Request

RX Receive

SBC Stuff Bit Counter

SDK Software Development Kit

SOF Start Of Frame

SRR Substitute Remote Request

STBY Standby

TEC Transmit Error Counter

TRIG Trigger

TX Transmit

USB Universal Serial Bus

VCC Voltage at the common collector

VDD Voltage Drain Drain

VIO Voltage In and Out