CAN Logger Project Report

by Joshua Sonnet et al.

Cyber security project 2020

1 Abstract

The CAN logger is a device build around the PyBoard v1.1 which features two integrated controllers and combined with two transceivers offers us the possibility to log any activity on a normal cars high speed Controller Area Network (CAN) that runs at 500 kbps, as well as concurrently send out messages. All this is implemented in a rather efficient code design in micropython, a small python3 implementation optimized to run on microcontrollers, that aims to log all CAN activities as fast as possible and at the same time write them to a SD card. But of course one can also filter the packets to be logged to a smaller subset, such one can focus and analyze only the important packets. In addition, it will also keep track of the current GPS location.
Furthermore, the CAN logger features a so called attack mode, which allows one to inject arbitrary messages into the bus. This feature is realized via a GPRS data connection and a Telegram Bot, so you can use the messenger Telegram and send your commands to that bot if permitted to do so. During this mode, there are commands available to initialize a replay, reply or bus-off attack in addition to plainly inject packets into the CAN bus. But of course the Telegram Service can also be used to remotely retrieve the log files, as well as updating the code.

Contents

2 Goal

This project aims to create a small, cost-effective and low-power CAN Logging Device, which also offers the ability to inject packets into the CAN bus. The idea was to use a small microcontroller which offers enough computing power to get the task done. Furthermore, the design should implement logging of multiple sensors in addition to the CAN packets, for that reason it should have an easy to use interface to add new sensors if desired in addition to the already existing GPS sensor.
But we also want to keep the code updated, for that reason we needed an over the air functionality. Therefore, we opted to add a mobile communication ability for which we decided to use GSM as it is not bound to a local connection or device.
As a final feature, we wanted some kind of interface to talk to the logger and issue certain commands remotely, but as there is no way to use protocols like ssh, we opted to use the GPRS data connection of the already existing GSM module and retrieve commands that way.

3 Hardware Design

The whole design is based around the PyBoard V1.1 which uses the STM32F Cortex M4 processor. The board offers us a wide range of connectivity options, as it has five UART interfaces and two integrated CAN controllers, as well as several GPIO pins. Main reason to use a PyBoard was for one the really slow speed of any Arduino controller compared to this one and on the other hand the increasing popularity in micropython. But of course another good alternative would have been the popular ESP32 but it only features one CAN controller and two UART interfaces, as well as not having a direct place to connect a SD card.
Additionally, the project uses multiple external modules to collect data, like the NEO6M GPS module, which runs fully on its own, once supplied power and found at least three satellites to determining the current location and outputting it through the serial interface.
Further we use two SN65HVD230 CAN transceivers, one for sending purposes, the other one for receiving, such that we can listen and in the same time send messages. They are 3.3V compatible and are able to keep up to a automotive grade CAN. A Controller Area Network (CAN) is a serial two-wire half-duplex network typically used for communication between nodes without loading the microcontroller. A CAN transceiver now interfaces between the CAN protocol controller and the physical wires of the CAN bus lines and has the job to convert the logic level to bus signals and the other way around. That said the CAN controller receives the command from the microcontroller on what to send and translate that to logic signals send to the CAN transceiver. It takes either 1, that means not driving the bus, so the HIGH and LOW lines float to 2.5V, also known as a recessive bit, or 0, which means driving the HIGH line high and the LOW line low, also known as a dominant bit.
Finally, there is also the SIM800L GSM module, which can be fully used as a phone which both pins for a microphone and speaker but that is not our focus here. Instead, we make use of the RING pin which is pulled down as soon the modules receives a call or text message and the DTR pin that can be used to take the module into sleep mode and reduces the power consumption. But very important for later, the module can still receive text messages or calls normally. However, this is only one side of our use, the other one is using GPRS data connection to connect to the internet and be able to do GET and POST requests.
Of course all of this also has to be powered and for that we are using a OBD-2 to Sub-D cable, which will then be connected to a Sub-D male connector from the logger. So on pins 2 and 7, we get chassis ground and battery power respectively from the vehicle as well as the needed CAN HIGH and LOW on pin 3 and 5. But obviously the direct voltage of the battery would be way too much for our sensitive electronics, for that reason we are using a LM2596 step down power converter to a stable 4V. Now one would ask why such an odd number. But there is a good reason and that is the GSM module which takes 3.5V - 4.3V to work, as it was designed to work with a LiPo battery. Furthermore, we have to cope with current spikes up to 2 amps for when the GSM is connecting to a network, as it else would reset itself due to low power. Therefore, we also incorporated

a capacitor array to feather of these spikes. And last but not least all other modules are powered by the PyBoard itself at 3.3V, as the PyBoard takes a range of input voltage and internally converts it to a safe 3.3V but is only able to supply at about 200 mA of current, but is more than adequate for the rest.

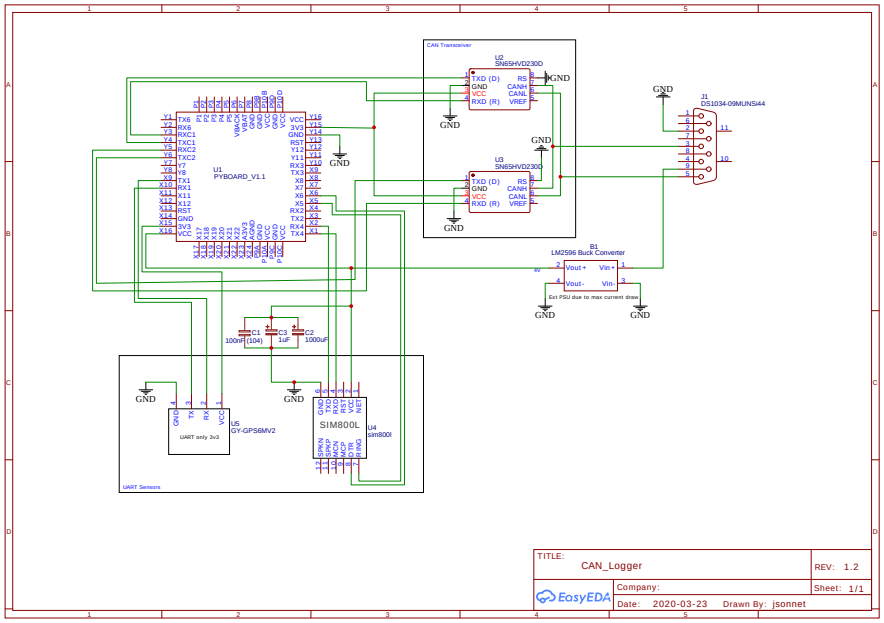

Figure 1: The design schematic (can be found on GitHub in full)

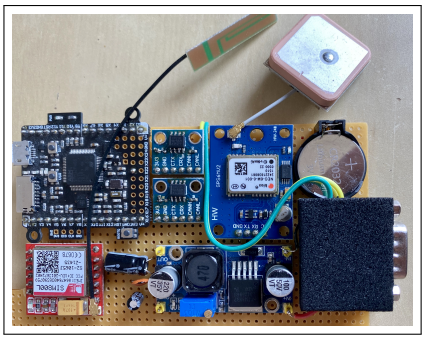

Figure 2: Finished prototype front

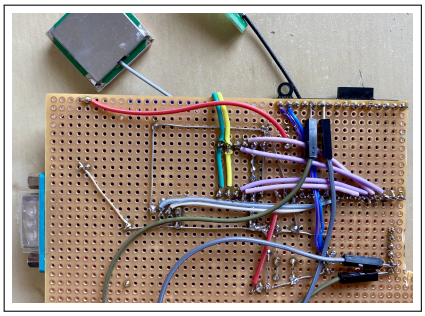

Figure 3: Finished prototype back

4 Execution Plan

In the following I will explain the procedure of the code. Upon powering the CAN logger up, it will first initialize itself as well as the CAN controllers, the GPS and GSM module. In addition, it will also try to do an over-the-air update of the code if a newer version is available, as well as start the Telegram bot service if a successful GPRS connection is established.
After the init phase it will enter the main stage, which primarily consists of logging all CAN packets to a log file, in combination with the current GPS location every couple of seconds.
Since no CAN data is transmitted when the vehicle is switched off, the CAN logger automatically goes into sleep mode after 30 seconds without receiving new packets. During the sleep phase it will shut off the GPS and set the GSM to a low power mode to further save power and will only check every couple of seconds for new packets being sent by the CAN bus, to again enable all functionality.
In addition, the logger also has an attack phase that can be activated by calling the SIM card in any of the previously mentioned phases. This wakes the logger up again even when it has been sleeping, since we did not enable airplane mode rather a low power mode. But instead of logging CAN messages, it starts to wait for new commands sent to the telegram bot.
As soon as the bot receives an exit command, it will quit the attack phase and return to its regular logging.

4.1 Logging Phase

During logging the logger will capture every packet sent via the CAN bus and log it to a log file either on an external SD card if present or the internal storage, but latter is not advised due to the tiny storage the board offers. Every message that is sent we append the current epoch time, the CAN ID of the ECU that sent the packet and data itself. Furthermore, every five seconds by default, the logger requests the current GPS position, in addition to the current traveling speed, and log the information in combination to the epoch time on disk.
In the following is a schematic how such a log file could look like.

$$\begin{array}{llll} time_1, & id_1, & data_1, & \\ time_2, & id_2, & data_2, & \\ time_3, & lat_1, & long_1, & speed_1, \end{array}$$

As one can see based on that design, it is really easy to import the data into any plotting software of choice and analyze that data. But this is not where it ends, there is also the possibility to filter the log to certain CAN IDs and if at some point the car is shut off and all communication ends the logger will wait 30 seconds until it quits logging and enter sleep phase.

4.2 Attack Phase

As already mentioned, the attack phase is initiated by a call to the Sim card inserted into the GSM module. The logger then listens to all commands sent via telegram. For this purpose the API of Telegram is used, for which the logger is registered as a bot. Now you can use the commands to perform functions like downloading the log file or deleting it. Furthermore, it is possible to force an OTA update or to set or remove the previously mentioned filters for CAN packets. But as the name of this phase already suggests, attacks on the CAN bus are also possible.
First of all, there is the replay attack, which waits for a given packet to be sent along the CAN bus and replays the same packet again whenever the packet is seen again. Next there is the reply attack which waits for a given message and upon receiving will answer by sending a response packet. Next it also offers to just inject a custom message at a specified frequency, this means one can inject arbitrary packets with a custom ID and specify how often and how fast it should be sent. Last but not least there is a fairly manual bus off attack implemented which takes a marker, delay and payload and upon seeing the marker packet will wait for the specified delay, when supposedly the real message is sent and sends the payload at the exact time.

5 Code Design

This segment aims to further explain the implementation and highlight some key elements about it. The whole code is placed inside a python class, which allows for automatically initializing the logger inside the dunder init function. At that stage we open UART connections to both the GPS and the GSM module together with their respective libraries. Here it is important to note, that we need to enlarge the buffers such that we can receive enough bytes. Further we need to set the correct prescaler values for the CAN controller to operate at 500 kbps, these values can be calculated based on the base clock speed of the STM32 with 168 MHz. Additionally, we also need to set up the filter mask, such that we can receive 16 bit packets and have no ID filter set. Next we try to establish a GPRS connection and if it was successful continue to start the over-the-air update service, which will be explained below, as well as the Telegram bot service. But there are also some little things like the automatic recognition if an SD card has been inserted, so that we can write the protocol there instead of on the internal memory. We also set up the automatic wake-up call of our internal RTC when we enter the sleep phase and need a regular interruption. Last but not least, we enable the external interrupt pin for when the GSM module receives a call to enter the attack phase, which in return calls a callback function.
Speaking of said callback function, when the SIM800L modules receives a call it pulls the RING pin low and in return will activate this function, which hangs the call up and sets the corresponding flag for the main loop but also sends an acknowledgment to all authorized telegram users. As this function can also be activated when the logger was in its sleep phase, it will also reactivate the full functionalities of the GSM and GPS module.
As prior mentioned the logger offers the possibility to update its code base over-the-air if a newer version is found. For that we request the latest version number found inside of a designated version file and compare it against the current one. If we now find that a newer version is available, we download it and restart the device to load the overwritten file.
But there is also the sophisticated function for logging, which allows you to specify any file, that is automatically loaded and written to and as many further parameters as one would choose to add to each line printed. In Section 4.1 is an example schematic how such an output could look like. However, if we do not specify the file or use the default log file, we check if the file is still opened, else we will reopen the file and then append the data to said file. We keep the file open because we want the maximum possible speed when writing the log, since closing the file each time would cause a massive overhead. In any case, the logging function can be seen as a logging interface, since we only need to specify the data and can easily add new modules with values to be logged.
Next there is the message handler for our Telegram service. Once started, the bot will send an acknowledgment to the connected user and set a custom keyboard (seen in Figure 4) to easily send the provided commands. But of course not every user is permitted to use the Bot, for that reason we check the unique UID of every everyone and check it against a whitelist. Additionally, the messages sent by the bot are not broadcasted to

every user, but rather independent to every user. These commands consist of uploading the log file via a multipart/form-data POST request to Telegram in order to remotely access this data. As well as adding or removing CAN IDs to the filter for when the logger returns to the logging phase, will only focus on packets from these particular IDs. We can also initiate the OTA service to force a code update.
Next there are the attacks, which I will explain in more detail next. First of all there is normal injection, which will take a packet followed by a custom frequency, in this case then the delay for the processor to wait before sending the next packet. All this wrapped around a loop to inject a specified amount of messages into the CAN bus via the second transmitting CAN controller. The replay attack will also receive a packet as parameter and will then wait for this packet to come along the CAN bus in order to be resent by the logger. Here is a prime example for using two CAN controllers, as it allows us to send and receive in a short time span. The reply attack is almost identical with the only difference that if will send a specified answer packet instead of replaying the same message.
This leaves us with the bus-off attack. At the current point of time the implementation is fairly manual, which means one has to calculate timings as well as decide on a marker packet in advance. Here we want to achieve to send our payload at the same time as the real node of the CAN bus would send his message in order to increase his Transmission Error Counter (TEC). This is achieved by sending the same packet and results in the victim having a bit error, due to having a dominant bit whereas the victim's is recessive. This results in the target ECUs TEC to increase by 8 and once reached 255 the target enters bus-off mode and is isolated from participating in the network. This leaves us with the option to spoof its packets without any other ECU knowing. But the implementation is still subject to change in the future, to make it more robust in working. And finally if any user sends the exit command the logger will return to its normal phase, to log CAN messages and thus stop listening for commands.

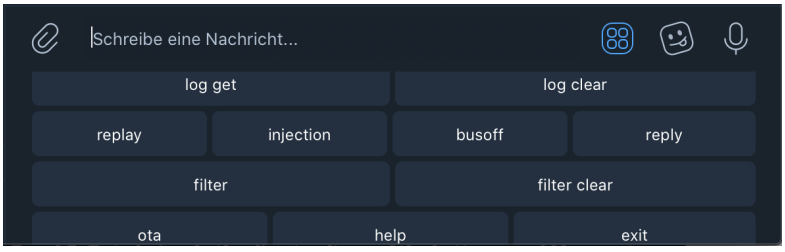

Figure 4: The telegram custom keyboard

And last but not least we will highlight the main routine of the logger code. There are three parts to the main loop, the first is the sleep stage whereas long as the sleep flag is set and there are no new CAN packets send over the bus, the device will return to sleep, but as soon as there are new activities, the logger will restart its modules and return to logging. The next stage is the normal mode where we check in specified interval the current GPS location by receiving the UART data from the module and interpret them by parsing with the designated ublox GPS library. This will then get logged together

with the current epoch time from the RTC. And of course the main reason for this logger, we check if a new CAN message can be received and either log it directly on disk or if a filter for specific IDs is set, check if the received packet is originating from this ID and then log. But if the timeout of receive function for the CAN Transceiver runs out as no new packets were available an error is throw. This way we can catch when this happens and initiate the above mentioned sleep phase of the logger. Finally, if the attack mode flag is set from the callback function, the logging part will be skipped and iterate checking if new commands are received by the Telegram bot.

6 Improvements

Parallel work As with most microcontrollers doing some pretty intense work, which is here certainly the concurrent receiving of CAN messages and writing them to disk, as storing them in memory is not an option with only about 120KB max. Here I am already using the best possible way, as I am storing data in the file buffer until it fills up and writes itself on disk. This is done in python by not closing the file during writes but leaves the potential of corrupting the file. If we e.g. had a faster processor or multiple truly parallel threads, we could simultaneously receive CAN packets as well as the current GPS position without ever missing some packets in between logging them.

GSM module Another improvement would be the GSM module and its ability to communicate to a data connection, as there is no easy way of using python requests to do get or post requests. But rather having to use AT commands to build your own way for such requests. Furthermore, the cheap SIM800L is a real pain to work with due to its voltage spikes and pickiness about connections.

Bus Off attack One last enhancement would be the Bus Off attack in its current implementation, as it is too manual and does not work with a high certainty. Therefore, one needs to go over the paper again and this time implement based on the complete description.