Abstract

The main objective of this project is to develop a Python application to transfer files in Intel hex format from a PC to a single STM32F407VG microcontroller using Can Bus protocol. The USB-Can adapter/converter used for this purpose is the leaf light HS from Kvaser. The microcontroller has to run in boot system mode to be able to reprogram the flash memory by using CAN. The boot from system memory is provided by STMicroelectronics.

The application of this project is intended to allow working with the whole STM32 family but it was only tested with the mentioned device. There are some restrictions in the embedded boot system that entails some limitations in the program functioning: the built-in bootloader from ST is fixed for all microcontroller, what means that if we want to program more than one microcontroller we have to connect them one by one to the Can interface. The flash memory can only be written when it is erased but the flash erase command is not supported by the manufacturer. In this project we describe an alternative way to erase the flash memory prior the programming.

The second part of this project is related to the design and development of an application based on the Python Tkinter module to create a graphic user interface (GUI) to make the program ease to use. This program utilizes the class developed in the previous application and it is separated from it to facilitate modifications or improvements by other developers.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>1</td>
</tr>
<tr>
<td>CONTENTS</td>
<td>2</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>4</td>
</tr>
<tr>
<td>1.1. Goals</td>
<td>4</td>
</tr>
<tr>
<td>1.2. Scope</td>
<td>4</td>
</tr>
<tr>
<td>2. PROJECT MANAGEMENT</td>
<td>6</td>
</tr>
<tr>
<td>2.1. Project planning</td>
<td>6</td>
</tr>
<tr>
<td>2.2. Project budget</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1. Project development cost</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2. Project execution</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3. Project impact</td>
<td>9</td>
</tr>
<tr>
<td>3. CONFIGURATION AND COMMUNICATION TESTS</td>
<td>11</td>
</tr>
<tr>
<td>3.1. Hardware description</td>
<td>11</td>
</tr>
<tr>
<td>3.1.1. STM32F407VG Discovery kit</td>
<td>11</td>
</tr>
<tr>
<td>3.1.2. Kvaser Leaf Light HS</td>
<td>12</td>
</tr>
<tr>
<td>3.1.3. Waveshare SN65HVD230 CAN Board transceiver</td>
<td>13</td>
</tr>
<tr>
<td>3.1.4. PL2303 UART board</td>
<td>14</td>
</tr>
<tr>
<td>3.2. Flash memory</td>
<td>15</td>
</tr>
<tr>
<td>3.3. Can bus</td>
<td>16</td>
</tr>
<tr>
<td>3.3.1. Can bus definitions</td>
<td>16</td>
</tr>
<tr>
<td>3.3.2. Frame in a Can bus</td>
<td>17</td>
</tr>
<tr>
<td>3.3.3. STM32 Can bus protocol</td>
<td>18</td>
</tr>
<tr>
<td>3.4. Boot system mode</td>
<td>18</td>
</tr>
<tr>
<td>3.5. Kvaser Canking</td>
<td>22</td>
</tr>
<tr>
<td>3.6. Kvaser Canlib library</td>
<td>23</td>
</tr>
<tr>
<td>3.7. Python programming</td>
<td>24</td>
</tr>
<tr>
<td>3.7.1. Canlib module</td>
<td>24</td>
</tr>
<tr>
<td>3.7.2. Python commands</td>
<td>26</td>
</tr>
<tr>
<td>4. PROGRAMMING OF THE FILE TRANSFER APPLICATION</td>
<td>29</td>
</tr>
<tr>
<td>4.1. CommandInterface class</td>
<td>29</td>
</tr>
<tr>
<td>4.2. Module usage</td>
<td>34</td>
</tr>
<tr>
<td>4.3. Future work</td>
<td>35</td>
</tr>
</tbody>
</table>
5. PROGRAMMING THE GRAPHIC USER INTERFACE .............. 36
   5.1. Class APP() ................................................................. 36
   5.2. Application flowchart ................................................... 39

6. CONCLUSION .................................................................. 41

7. ACKNOWLEDGEMENTS ..................................................... 42

8. REFERENCES .................................................................. 43
   8.1. Basic references ............................................................. 43
   8.2. Complementary references .............................................. 44

9. ANNEX.1 INSTALL SDK AND CANLIB LIBRARY ............... 46

10. ANNEX 2 STM32CANLOADER APPLICATION ....................... 47

10. ANNEX 3: CANPYTHON APPLICATION................................ 54
1. Introduction

The ETSEIB Motorsport team designs and develops a single seat racing car for the Formula Student's competition every year. In order to reprogram the Electronic Control Units (ECUs) of the car, which are currently composed by 4 Discovery Kit STM32F407VG microcontroller based boards, a new methodology to transfer applications from the development PC to the ECUs is needed. Until now, the team has been using an ST-Link/V2 programmer for both debugging and transferring applications. The characteristics of this programmer make it difficult to access to the different parts of the vehicle (once the ECUs are set up) because of the limited wire length and because it has to be done one by one.

On the contrary, the controller area network (CAN) is a broadcast type of bus what means that all the devices connected to the bus will receive the data transmission. The maximum bus speed and cable length (up to 500 kbit/s for 100 meters) makes it suitable as a second option for transferring these applications. For these reasons, in order to transfer the machine code (.hex files) generated by the compiler, it is preferred to use a CAN bus communication.

The application must be written into the Flash memory location using the built-in boot system mode of the microcontroller unit (MCU). It will be transferred using a Kvaser Leaf Light HS for the host side (PC) and a Waveshare SN65HVD230 CAN Board transceiver for the MCU side.

1.1. Goals

The main objective of this project is to develop a Python application to transfer Intel hex files from the PC to a single MCU using CAN communication protocol.

This application will use some basic commands supported by the device, such as write to flash memory, verification of written application and get product information.

1.2. Scope

This project includes the programming in Python of an application to send files in Intel Hex format to the flash memory of a STM32F407VG Discovery kit. The communication between the PC and MCU will be performed using a Kvaser Leaf Light HS can interface and a Waveshare SN65HVD230 can board transceiver for the MCU.
Additionally, this application will have other useful functions, such as verification of the written file once sent to the flash, setting up of flash address and read of device characteristics.

Finally, this project includes a graphic user interface (GUI) programmed also in Python. This application will be independent from the main program to make it easier to add changes in the future by other developers.
2. Project management

In this section, it will be estimated the time dedicated to the realization of this project from the very first phase of study of technical documentation until the redaction of this document. In addition, it is specified the project breakdown, which is divided basically into 4 main blocks. Finally, there is an estimation of the project budget if it is intended to be executed by the reader.

2.1. Project planning

Prior the start of a project there has to be a planning estimation of the activities to be carried out. In this project it is used an application [1] to draw a Gantt chart (see Fig. 1) given the number of activities and start/end time. The final table of dedicated hours (see Fig. 2) was slightly readjusted to take into account the unexpected incidences, especially while developing the programs, but maintaining the scheduled deadlines.

The project can be divided in 4 main phases:

1. Study and recompilation of information: once defined the objective and scope of this project it was important to start searching/reading the documentation on the web and study the characteristics of the device to know what it allows us to do and how. That is why I have in parallel of the communication tests using CanKing and the CANlib library

2. Stm32canloader application: it is the main program that opens and manages the channel to communicate via CAN. In this phase, it was intended to build a class with functions that allows us to transfer complete files in Intel hex format.

3. CanPython application: this is the GUI program to manage the stm32canloader. It had to be developed after the previous program was finished and tested. This program is independent from the other one so that developers can improve it or modify it for other purposes.

4. Project documentation: it includes the documentation acquired in the first phase and the test results of the subsequent ones.

The work breakdown is shown in Fig. 2. One of the most effort and time consuming tasks was the stm32canloder application because it involves time in applying what was learnt
previously in a real hardware device. In addition, there have been some mistakes in the datasheets and application notes that lead us to some errors that could be eventually solved, at least the most important features.

![Gantt chart for the project](image)

**Fig. 1.** Gantt chart for the project. Source: own.

<table>
<thead>
<tr>
<th>Name</th>
<th>Duration</th>
<th>Start</th>
<th>Finish</th>
<th>State</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Study and recapitulation of MCU documentation and Kvaser library</td>
<td>35 h</td>
<td>6/03/17</td>
<td>31/03/17</td>
<td>released</td>
<td>manual</td>
</tr>
<tr>
<td>Can connection tests using Canking</td>
<td>15 h</td>
<td>14/03/17</td>
<td>16/03/17</td>
<td>released</td>
<td>manual</td>
</tr>
<tr>
<td>Can connection tests using canlib library</td>
<td>55 h</td>
<td>18/03/17</td>
<td>31/03/17</td>
<td>100%</td>
<td>manual</td>
</tr>
<tr>
<td>Project’s documentation</td>
<td>75 h</td>
<td>1/05/17</td>
<td>25/06/17</td>
<td>75%</td>
<td>manual</td>
</tr>
<tr>
<td>stm32Canloader application development</td>
<td>95 h</td>
<td>1/04/17</td>
<td>12/05/17</td>
<td>75%</td>
<td>manual</td>
</tr>
<tr>
<td>CanPython application development (GUI)</td>
<td>58 h</td>
<td>14/05/17</td>
<td>12/06/17</td>
<td>75%</td>
<td>manual</td>
</tr>
</tbody>
</table>

**Fig. 2.** Work breakdown structure. *Estimated hours and state of the activities. Source: own.*
2.2. **Project budget**

This section includes the cost in the development of this project and the corresponding execution. In addition, a small analysis of the impact and contribution to the users, society and environment is included.

2.2.1. **Project development cost**

<table>
<thead>
<tr>
<th>Task</th>
<th>Hours or units</th>
<th>EUR/h or EUR/unit</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>STM32F407VG</td>
<td>2 units</td>
<td>19.12</td>
<td>38.24</td>
</tr>
<tr>
<td>Kvaser leaf Light HS</td>
<td>1 unit</td>
<td>275</td>
<td>275</td>
</tr>
<tr>
<td>PL2303</td>
<td>1 unit</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DSUB 9 pins connector + cover</td>
<td>1 unit</td>
<td>2.7</td>
<td>2.7</td>
</tr>
<tr>
<td>Time dedicated by the designer</td>
<td>333 h</td>
<td>10</td>
<td>3330</td>
</tr>
<tr>
<td>Hours consulted to an expert (project director)</td>
<td>30 h</td>
<td>40</td>
<td>1200</td>
</tr>
<tr>
<td>University enrollment</td>
<td>12 ETCS</td>
<td></td>
<td>588.6</td>
</tr>
<tr>
<td>PC and licenses amortization(^3) (Total price:1200 EUR)</td>
<td>1 unit</td>
<td>-25</td>
<td>-25</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td></td>
<td></td>
<td>5413 EUR</td>
</tr>
</tbody>
</table>

---

1 Estimated hours

2 Only time for technical consulting. The management, document revision and qualification is not included

3 Max redeemable 26% of total in 4 years (minimum 4 years) for the 4 months of the project
2.2.2. Project execution

The project execution only includes the material and devices required. It is not included the time needed to customize the program (if necessary).

<table>
<thead>
<tr>
<th>Task</th>
<th>Units</th>
<th>EUR/unit</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>STM32F407VG</td>
<td>1 units</td>
<td>19.12</td>
<td>19.12</td>
</tr>
<tr>
<td>Kvaser leaf Light HS</td>
<td>1 unit</td>
<td>275</td>
<td>275</td>
</tr>
<tr>
<td>PL2303</td>
<td>1 unit</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DSUB 9 pins connector + cover</td>
<td>1 unit</td>
<td>2.7</td>
<td>2.7</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td>300.82 EUR</td>
</tr>
</tbody>
</table>

2.2.3. Project impact

There are two main blocks considered for this analysis, the first one is an estimation of the carbon footprint and the second one the impact in the society.

Due to the small dimension of this project, for the environment impact analysis, its most relevant factors are the energy consumed while developing it (desktop PC energy consume) and the CO₂ generated by the electronic devices, wires and packaging. These electronics devices do not consume much energy so it was not considered.

It is very difficult to estimate a value of the energy and emission generated for manufacturing an electronic device because it is composed by different materials (plastic, metals like silicon, copper, etc.). For this study, it is approximated to only plastic because that is the majority component. The following table contains the approximate CO₂ emissions for the development of this project.

---

*Estimated hours*
<table>
<thead>
<tr>
<th>Material/energy</th>
<th>Weight (kg) or Energy (kWh)</th>
<th>Factor</th>
<th>CO₂ emissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electronic devices, wires, packaging.</td>
<td>0.5 kg</td>
<td>4.5x10⁷ J/kg⁵ and 308 gCO₂/kWh</td>
<td>1940.4 gCO₂</td>
</tr>
<tr>
<td>Energy consumed</td>
<td>40.0 kWh</td>
<td>308 gCO₂/kWh</td>
<td>12320 gCO₂</td>
</tr>
<tr>
<td>(desktop PC – 333 hours at 120 W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>14260.4 gCO₂</td>
</tr>
</tbody>
</table>

The factor for the plastic (electronic devices) is the average process energy requirements for most plastics, which is 4.5x10⁷ J/kg [19]. Once the energy is calculated, it is applied the electric mix factor (emission associated to the electric energy in Spain [20]) which is 308 gCO₂/kWh. The total emission is 14260.4 gCO₂.

The impact in the society includes the applications in which this project and its adaptations can take place. For example, it can be used to, in addition to another microcontroller, reset and run a group of stm32 microcontrollers in boot system mode in order to reprogram them from distance. It also has an educational and instructive component, its relative simplicity allows reading the code and learning about Can protocol.

---

5 1J=2.8x10⁻⁷ kWh
3. Configuration and communication tests

In this section, it is introduced the MCU set up, a brief description of the Can protocol and the hardware used in this project. In addition, the steps for this project development are discussed, which involves the first communication tests using a Can bus monitor software.

3.1. Hardware description

3.1.1. STM32F407VG Discovery kit

As mentioned previously, the device to be programmed is an STM32F407VG Discovery kit, which is a development board with a 32-bit microcontroller with Cortex M4 core. It has an embedded flash memory of 1024 kbytes and features advanced communication interfaces such as UART, ETHERNET and CAN. Actually the only available Can interface is CAN2 because CAN1 manages internal communication during boot loader execution. Further information can be found in the product datasheet [2]. The MCU is fed with 5V using an USB interface and the Can board transceiver is connected to pins PB5 and PB13 that corresponds to CAN_RX and CAN_TX, respectively. To initialize the MCU in boot system mode the BOOT0 and BOOT1 pins must be connected to 1 and 0, respectively. Fig. 8 shows the connections to set up the MCU for Can communication.

![STM32F407VG Discovery kit](image)

*Fig. 3. STM32F407VG Discovery kit.*
*Source: STM32 Datasheet[2].*
3.1.2. Kvaser Leaf Light HS

The Kvaser Leaf Light HS supports full speed USB interface for CAN. The CAN channel has a 9 pin D-SUB plug (Fig. 5). The Can Low and Can High signals are connected to pin 2 and pin 7, respectively. This device is connected to the PC with the USB interface and to the Can board transceiver with a D-SUB plug. Fig. 7 shows the result once all the connections are done.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit rate</strong></td>
<td>5-1000 kbit/s</td>
</tr>
<tr>
<td><strong>Clock accuracy</strong></td>
<td>100 μs</td>
</tr>
<tr>
<td><strong>Max message rate</strong></td>
<td>8000 messages/s</td>
</tr>
<tr>
<td><strong>Can physical layer</strong></td>
<td>High speed (ISO 11898-2)</td>
</tr>
</tbody>
</table>

Fig. 4 Kvaser Leaf Light HS. Source: [8]

Table 1 Technical data for Kvaser Leaf Light HS. Source: [8].

Fig. 5. The D-SUB connector pin numbers. Source: [8].
3.1.3. Waveshare SN65HVD230 CAN Board transceiver

It is an accessory board used for connecting the MCU to the CAN network. It operates with 3.3V supply. The CANL and CANH terminals are connected to the CAN network using 2 wires and a 2 pole connector, as shown in Fig. 7. Table 2 shows a summary of the connection described in the datasheet.

![Can board transceiver](image)

*Fig.6 CAN board transceiver. The pins are connected to the MCU and the terminals to the CAN network. Source: [21].*

<table>
<thead>
<tr>
<th>Can transceiver pin</th>
<th>MCU terminal</th>
<th>Wire color</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3V</td>
<td>3V</td>
<td>Red</td>
</tr>
<tr>
<td>GND</td>
<td>GND</td>
<td>Black</td>
</tr>
<tr>
<td>CAN RX</td>
<td>PB5</td>
<td>Yellow</td>
</tr>
<tr>
<td>CAN TX</td>
<td>PB13</td>
<td>Green</td>
</tr>
</tbody>
</table>

*Table 2. Technical data for Kvaser Leaf Light HS*
3.1.4. PL2303 UART board

It is a module for USB type A connector to UART interface. This module was used during the development of the stm32canloader application for testing and to discard errors in the MCU. The data transmission is done using a free software application [3] that also allows us to erase the flash memory. That is, converting all bytes in memory banks to be used to 1s. We realized that if we overwrite bytes over the flash there will be many errors, so we have to erase flash memory before transferring files [4].

Furthermore, since the procedure of how to do erase flash memory is not explained in the Can application note AN3154 [5], but in the UART application note AN3155 [6], it was necessary to use this module.

<table>
<thead>
<tr>
<th>UART pins</th>
<th>MCU terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>RXD</td>
<td>PB10</td>
</tr>
<tr>
<td>TXD</td>
<td>PB11</td>
</tr>
</tbody>
</table>

**Table 3 Pins connection for UART module. Source: AN3155 [6].**
3.2. Flash memory

The flash memory goes from addresses 0x08000000 – 0x080FFFFF. The memory map is shown in Fig. 10. This memory is divided in memory banks or sectors (see Fig. 11). These sectors has to be specified with the erase operation because mass erase is not allowed.

**Fig. 10. Memory map for the STM32F407VG microcontroller. Source: Datasheet [2].**
3.3. Can bus

3.3.1. Can bus definitions

The Can bus protocol is a message-based protocol with application especially in the automobile sector to connect electronic control units (ECUs) to various subsystems such as engine control unit, transmission, airbags, battery and recharging systems for hybrid/electric cars, etc. These ECUs, also known as nodes, are connected to each other through a two wire bus. Fig. 12 shows the connections in a multi-master serial bus and inside a Can bus node.
3.3.2. Frame in a Can bus

The CAN protocol uses short messages (or frames) to communicate. It can be of four different types. The Kvaser handles these kind of frames, such as the ‘Data frame’ type, which is the most common message type.

It is not intended to explain in detail every part of the frame but its most important characteristics. According to the stm32 can protocol [5], the identifier (ID in the arbitration field) contains the command to be sent to the MCU (see section 3.3.3). It can be an 11-bit Identifier for Can 2.0A (standard CAN) or a 29-bit identifier for Can 2.0B (extended CAN). Fig. 13 shows the structure of a standard frame with an 11-bit identifier.

The control field contains the data length code (DLC), which is the number of bytes (from 0 to 8) that contains the Data Field. These frame parts will be used frequently in the next sections and chapters while working with the CanKing and the Canlib library for the python programming.
3.3.3. STM32 Can bus protocol

Once the system memory boot mode is entered and the STM32 device has been configured, the bootloader code waits for a frame on the CANx_Rx pin [5].

The STM32 CAN is compliant with the 2.0A and B (active) specifications with a bitrate up to 1 Mbit/s. It can receive and transmit standard frames with 11-bit identifiers as well as extended frames with 29-bit identifiers. The supported commands are listed in Table 4.

In this application, the CAN settings are (see Fig. 13):

• Standard identifier (not extended).

• Bitrate: at the beginning it is 125 kbps; during runtime it can be changed via the speed command to achieve a maximum bit rate of 1 Mbps.

The transmit settings (from the STM32 to the host) are:

• Tx identifier: (0x00, 0x01, 0x02, v03, 0x11, 0x21, 0x31, 0x43, 0x63, 0x73, 0x82, 0x92).

The receive settings (from the host to the STM32) are:

• Synchronization byte, 0x79, is in the RX identifier and not in the data field.

• RX identifier depends on the command (0x00, 0x01, 0x02, 0x03, 0x11, 0x21, 0x31, 0x43, 0x63, 0x73, 0x82, 0x92).

3.4. Boot system mode

At startup, boot pins are used to select one out of three built-in boot options provided by STMicroelectronics [2]:

1. Boot from user Flash

2. Boot from embedded SRAM

3. Boot from system memory

The boot loader is located in system memory. It is used to reprogram the Flash memory by using USART1 (PA9/PA10), USART3 (PC10/PC11 or PB10/PB11), CAN2 (PB5/PB13), USB OTG FS in Device mode (PA11/PA12) through DFU (device firmware upgrade).

The boot from system memory is achieved by connecting two jumpers to the correct pins:
Boot0 connected to VDD and Boot1 connected to GND (the MCU Boot1 pin is the PB2). There are different patterns to initialize in boot system mode as described in application note AN2606 [7] but in this case, we use the one mentioned above.

Once the device is connected to the pins for boot system mode, we have to connect the Can transceiver to the correct MCU pins. Fig. 8 shows the connection between the MCU and the Can transceiver.

After initializing in boot system mode the MCU turns on LEDs LD1/COM and LD2/PWR and now it is possible to send commands using its Can protocol.
<table>
<thead>
<tr>
<th>Command</th>
<th>Command code</th>
<th>Command description</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get</td>
<td>0x00</td>
<td>Gets the version and the allowed commands supported by the current version of the bootloader</td>
<td></td>
</tr>
<tr>
<td>Get Version &amp; Read Protection Status</td>
<td>0x01</td>
<td>Gets the bootloader version and the Read Protection status of the Flash memory</td>
<td>The command doesn’t give the read protection status, only the bootloader version</td>
</tr>
<tr>
<td>Get ID</td>
<td>0x02</td>
<td>Gets the chip ID</td>
<td></td>
</tr>
<tr>
<td>Speed</td>
<td>0x03</td>
<td>The speed command allows the baud rate for CAN run-time to be changed.</td>
<td>We cannot change the baudrate of the MCU, neither initialize it in a baudrate different from 125K. Furthermore, the function to modify the baudrate using canlib is not implemented.</td>
</tr>
<tr>
<td>Read Memory</td>
<td>0x11</td>
<td>Reads up to 256 bytes of memory starting from an address specified by the application</td>
<td></td>
</tr>
<tr>
<td>Go</td>
<td>0x21</td>
<td>Jumps to user application code located in the internal Flash memory or in SRAM</td>
<td></td>
</tr>
<tr>
<td>Write Memory</td>
<td>0x31</td>
<td>Writes up to 256 bytes to the RAM or Flash memory starting from an address specified by the application</td>
<td>In order to transfer files, it is only possible to send data frames of 4 bytes and addresses</td>
</tr>
</tbody>
</table>
Can bus bootloader for the STM32F407VG

<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erase</td>
<td>0x43</td>
<td>Erases from one to all the Flash memory sectors</td>
<td>It is not possible to erase all the flash memory using can in boot system mode. The procedure of how to correctly erase sectors memory is not explained in the AN3154</td>
</tr>
<tr>
<td>Write Protect</td>
<td>0x63</td>
<td>Enables the write protection for some sectors</td>
<td>Command not used</td>
</tr>
<tr>
<td>Write Unprotect</td>
<td>0x73</td>
<td>Disables the write protection for all Flash memory sectors</td>
<td>Command not used</td>
</tr>
<tr>
<td>Readout Protect</td>
<td>0x82</td>
<td>Enables the read protection</td>
<td>Command not used</td>
</tr>
<tr>
<td>Readout Unprotect</td>
<td>0x92</td>
<td>Disables the read protection</td>
<td>Command not used</td>
</tr>
</tbody>
</table>

*Table 4. Commands supported by the STM32F407VG in boot system mode. Source: AN3154[5].*
In summary, the procedure to connect to the STM32 using CAN bus is as follows:

1. Start MCU in boot system mode and connect it to the CAN network.
2. Send synchronization byte (0x79 in ID field) to the MCU and wait for its response.
3. Start communication, now it is possible to send new commands following the instructions detailed in AN2606 [7].

### 3.5. Kvaser Canking

In order to test the commands and to try the CAN protocol it is used the Kvaser CanKing bus monitor because its compatibility with the Kvaser Leaf Light HS. The basic features and how to use this software are found in the official web site from Kvaser [17][18]. The commands from the CAN protocol used in this project and the procedure on how to write over the Flash memory have been explained in sections 3.2 and 3.4.

Fig. 14 shows the frames sent between the host (PC) and the MCU after sending the synchronization byte (0x79), a request of the boot loader version (0x01) and the ID chip (0x02).

Table 5 contains a summary of the data obtained with this commands and other parameters.

<table>
<thead>
<tr>
<th>Model</th>
<th>Bootloader version</th>
<th>Synchronization byte (ACK)</th>
<th>Product ID</th>
<th>Baud rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>STM32F407VG</td>
<td>2.0</td>
<td>0x79</td>
<td>0x413</td>
<td>125 kbps</td>
</tr>
</tbody>
</table>

*Table 5 MCU\’s parameters and data obtained with Can King. Source: own.*

From the output window we can see that the synchronization is done correctly (ACK received), the boot loader version is 2.0 (32 in hexadecimal format) and the chip ID is 0x413 (Fig. 14 shows data in decimal format). At the beginning, there are 2 ErrorFrame messages that the CanKing receives just before sending the synchronization byte so we will have to take it into account while writing the Python application.
Can bus bootloader for the STM32F407VG

3.6. Kvaser Canlib library

Canlib SDK [10] is a software product from Kvaser that provides the application programmer with an access to the CAN network. It allows us to develop our own applications that interacts with our Kvaser device. The current software version is v5.19 and it is free from the manufacturer main website. The software includes 3 different libraries [11]:

1. The first one is CANlib. We have functions to set bus parameters (e.g. bit rate), go bus on/off and read/write CAN messages. All of this is done on a device that is attached to the computer, in this case the Kvaser Leaf HS, but they are universally available across all supported Kvaser Devices.

2. The second is Kvrlib, used to configure some Kvaser devices, known as remote devices that can be connected via Ethernet.

3. The last library is Kvmlib. This library is used to interact with Kvaser Memorator devices that can record CAN messages.

We only use the CANlib library, which is implemented for python and it is enough for our purposes. After checking out that our device responds to Can messages using CanKing we
can proceed to start sending and reading frames using python programming.

3.7.  **Python programming**

In this section we test the can protocol commands from AN3154 [5] and learn how to manage the data received from the device. The correct procedure to initialize a Can communication, in general terms, is the following:

1. Open a channel: it is necessary to have a handle different to 0. A handle is a pointer to the communication port, it is where information is sent and received.

2. Set bus mode on: there is access to the bus and it is possible to send and receive information.

In order to use the Canlib module in our program we have to import it by typing the following code line:

   ```python
   import canlib
   ```

3.7.1.  **Canlib module**

There is an example on how to use the Canlib library in the Canlib SDK help [12]. There are also some useful functions in the module that we can use to connect a device to the CAN network and to send/receive messages. For this purpose it is necessary to initialize the Canlib class:

**Canlib class**

It is a wrapper class for the Kvaser CANlib. It allows us to open the canChannel class and initializes the driver. It must be called before any other function in the Canlib library:

   ```python
   cl=canlib.canlib() # When we initialize the canlib class it initializes the driver
   ```

**canChannel class:**

It allows us to open and set up a channel and to use its functions

   ```python
   ch = cl.openChannel(chan, flagsmode) # Retrieves a canChannel object for the given CANlib channel number using the supplied flags. We have to create a Canlib object to use this function. The openChannel arguments are:
   ```
   ```python
   - 'chan' (int) is the channel number. In this case we only use channel 0.
   ```
Flaggsmode (int): In this case we use `canOPEN_ACCEPT_VIRTUAL`, whose value is 0x0020. Allow opening of virtual channels as well as physical channels.

```python
ch.setBusParams(self, freq)  # This function sets the bus timing parameters for the specified CAN controller. The library provides default values for the bus parameters when `freq` is specified to one of the pre-defined constants. The only possible values used by the MCU are:

- `canBITRATE_125K`: Sets bus to baudrate 125 kb/s. It is the default baudrate that the STM32 uses.
- `canBITRATE_250K, canBITRATE_500K, canBITRATE_1M`: Correspond to baudrates of 250 kb/s, 500 kb/s and 1M/b/s, respectively.
```

```python
ch.busOn(self)  # Takes the specified channel on-bus.
```

```python
ch.busOff(self)  # Takes the specified channel off-bus. Closes the channel associated with the handle.
```

```python
ch.close(self)  # Closes the channel associated with the handle.
```

```python
ch.write(self, id_, msg, flag=0, dlc=None)  # Send a CAN message. Arguments:
    - `id_`: The identifier of the CAN message to send.
    - `msg`: An array or bytearray of the message data
    - `flag`: A combination of message flags, `canMSG_xxx`. Use this parameter e.g. to send extended (29-bit) frames. This flag is not used since we only send standard frames.
    - `dlc`: The length of the message in bytes. For Classic CAN `dlc` can be at most 8, unless `canOPEN_ACCEPT_LARGE_DLC` is used. For CAN FD `dlc` can be one of the following 0-8, 12, 16, 20, 24, 32, 48, 64. Optional, if omitted, `dlc` is calculated from the `msg` array.
```

```python
ch.writeWait(self, id_, msg, flag=0, dlc=None, timeout=0)  # Sends a CAN message and waits for it to be sent. It returns when the message is sent, or the timeout expires. The arguments are the same as in `write()` and one else:
    - `timeout`: The timeout, in milliseconds. 0xFFFFFFFF gives an infinite timeout.
```

```python
ch.read(self, timeout=0)  # Read a CAN message and metadata. Reads a message from
the receive buffer. If no message is available, the function waits until a message arrives or a timeout occurs. Arguments:

- `timeout (int)`: Timeout in milliseconds, -1 gives an infinite timeout.

Returns a tuple containing:

- `id (int)`: CAN identifier
- `msg (bytes)`: CAN data - max length 8
- `dlc (int)`: Data Length Code
- `flag (int)`: Flags, a combination of the canMSG_xxx and canMSGERR_xxx values
- `time (float)`: Timestamp from hardware

### 3.7.2. Python commands

To program this application we use Python integrated development environment (IDLE) version 3.6.1 [13]. In this section we program a simple application that opens a Can channel, sends the synchronization byte and writes and reads data to/from the MCU flash memory.

The program to test the Can communication is shown in Fig. 15. The first lines initializes the library, open the channel, set baudrate to 125kb/s and sets bus on. The next line sends the synchronization byte. This command has a timeout of 255 ms. Then, it follows the 2 error frames seen with the CanKing and the ACK byte from the MCU.

The next commands are for the write operation:

1. The standard ID is 0x31, DLC=5, data[0] = 0xXX: MSB flash memory address, ..., data[3] = 0xYY: LSB address, data[4] = N-1 (number of bytes to be written - 1). The address where we are writing is 0x08000008.
2. The standard ID is 0x31, DLC=4; data = byte_m1, ..., byte_m4.

After the first command for write operation the MCU sends 1 ACK. After the second one the MCU sends 2 ACK.

The last commands are for the read operation:

The standard ID is 0x11, DLC=5, data[0] = 0xXX: MSB flash memory address, ..., data[3] =

---

6 Before executing the test program the flash memory was erased.
0xYY: LSB address, data[4] = N-1 (number of bytes to be read - 1).

Fig. 16 shows the result of executing the test program. The read operation returns a tuple where its first element is the ID of the requested command and the second one is the data received by the host. The ‘y’ element is an ACK byte (0x79). The second Error frame line was deleted because for some reason, in this case, it only returns one when we send the synchronization byte.

```python
import canlib.canlib as canlib

def canconfig():
    print("Initializing Canlib")
    cl = canlib.canlib()
    chan = 0
    print("Opening channel \%d" % (chan))
    ch = cl.openchannel(chan, canlib.canOPEN_ACCEPT_VIRTUAL)
    print("Setting bitrate to 125 Kb/s")
    ch.setBusParams(canlib.canBITRATE_125K)
    print("Going on bus")
    ch.busOn()
    ch.writeWait(0x79, [0, 0, 0, 0, 0, 0, 0, 0], 0, timeout=0x0FF)
    print(ch.read(timeout=0xFF)) # first error frame
    syncro = ch.read(timeout=0xFF)[1][0] # ACK
    if syncro==0x79:
        print("Syncro OK")
    else:
        print("*X") # (syncro)
    # write operation
    ch.writeWait(0x31, [8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 0, dlc=5, timeout=0x0FF)
    ch.writeWait(0x31, [0x01, 2, 0x03, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 0, dlc=4, timeout=0x0FF)
    print(ch.read(timeout=0xFF)) # First 0x31 ACK
    print(ch.read(timeout=0xFF)) # Second 0x31 ACK
    print(ch.read(timeout=0xFF)) # Second 0x31 ACK

    # read operation
    ch.writeWait(0x11, [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 0, dlc=5, timeout=0x0FF)
    print(ch.read(timeout=0xFF)) # 0x11 ACK
    tmp32 = ch.read(timeout=0xFF) # data read
    print(tmp32)
    print(ch.read(timeout=0xFF)) # 0x11 ACK
    ch.busOff()
```

*Fig. 15. Can communication test. Source: own.*
Fig. 16. Result of executing the test program. Source: own.
4. Programming of the file transfer application

In this section, we describe the steps for the application development once the previous Python tests are successfully executed. The main program has a command interface class to group all functions and make it easy to read or modify. The whole program is in the ANNEX 2 stm32canloader application.

4.1. CommandInterface class

The functions implemented for this class are described below:

- **openCan**: Initializes library, set bus baudrate and sets bus on. The arguments are the channel that is 0 in this case and baudrate set to a default value of 125 kb/s because it is not possible to initialize in a different value. There is a dictionary, dict_rate[abaudrate], with all possible values from AN3154[5] but it is not possible to change the abaudrate variable.

  ```python
def openCan(self, chan=0, abaudrate="125K", flagsmode=canlib.canOPEN_ACCEPT_VIRTUAL):
    #canOPEN_ACCEPT_VIRTUAL: Allows opening of virtual channels as well as physical channels.
    self.cl = canlib.canlib()
    self.ch = self.cl.openChannel(chan, flagsmode)
    print("Setting bitrate to %s" % (abaudrate))
    self.ch.setBusParams(dict_rate[abaudrate])
    print("Going on bus")
    self.ch.busOn()
  ```

- **_wait_for_ask**: reads frame received from MCU. Returns 1 if it receives an ACK, 0 if it is a NACK and raises an Exception if it is something else. It was implemented this way to discard possible errors.

  ```python
def _wait_for_ask(self, info = ":
    # wait for ask
    ask = self.ch.read(timeout=0xFF)[1][0]
    if ask == 0x79:
      # ACK
      return 1
    else:
      if ask == 0x1F:
        # NACK
        return 0
  ```
else:
    # Unknown response
    raise CmdException("Unknown response. "+info+":
    "+hex(ask))

- initChip: asks for synchronization byte and return 1 if it was done correctly. It
  disregards 2 error frames received before the ACK byte.

  
def initChip(self):
    # Set boot
    self.ch.write(0x79,[0,0,0,0,0,0,0,0],0,0)#timeout=0xFF
    self.ch.read(timeout=0xFF) #there are 2 ErrorFrames before receiving ACK
    self.ch.read(timeout=0xFF)
    return self._wait_for_ask("Syncro") #ACK sent by the device

- releaseChip: sets bus off

  
def releaseChip(self):
    self.ch.busOff()
    self.ch.close()

- cmdGeneric: it is used to send commands given an ID, address and dlc. It returns 1 if
  an ACK was received.

  
def cmdGeneric(self,cmd,addr_g=[0,0,0,0,0,0,0,0],dlc=0):
    self.ch.writeWait(cmd, addr_g, 0, dlc, timeout=0xFF)
    return self._wait_for_ask(hex(cmd))

- cmdGet: Asks and returns a string of commands allowed by the MCU

  
def cmdGet(self):
    stat = self.cmdGeneric(cmd=0x00)
    if stat==1:
        nbytes = self.ch.read(timeout=0xFF)[1][0]
        bootread = hex(self.ch.read(timeout=0xFF)[1][0][2:])
        #bootread =0x20 is version 2.0
bootver = bootread[0]+".\"+bootread[1]
commands_str=''
for i in range(0,nrbytes):
gcom=self.ch.read(timeout=0xFF) # get commands
commands_str=commands_str + hex(gcom[1][0])+''
"+commands[gcom[1][0]]+"\n'"
if self._wait_for_ask("0x00 end")==1:
    print("Bootloader version: "+ bootver) #print 2.0
    return commands_str #return list of commands
else:
    return stat #NACK or 0

- cmdGetVersion: Returns a string with the bootloader version of MCU

def cmdGetVersion(self):
    stat = self.cmdGeneric(cmd=0x01)
    if stat==1:
        temp = self.ch.read(timeout=0xFF)[1][0]
        bootread = hex(temp)[2:] #bootread =0x20 is version 2.0
        bootver = bootread[0]+".\"+bootread[1]
        print(bootver)
        self.ch.read(timeout=0xFF) #dummy bytes-> 0x00, 0x00
        self._wait_for_ask("0x01 end")
        return ("Bootloader version: "+ bootver+"\n") #return 3.2
    else:
        return 0

- cmdGetID: Returns a string with the product ID specified for each STM32 family.

def cmdGetID(self):
    stat = self.cmdGeneric(cmd=0x02)
    if stat==1:
        Pid_Read = self.ch.read(timeout=0xFF)[1] #Product ID
        self._wait_for_ask("0x02 end")
        print("Product ID: "+hex(Pid_Read[0])[2:]+'\n'+hex(Pid_Read[1])[2:]+'\n'+"Device name:"
        +chip_ids.get(Pid_num, "Unknown")+'\n')
        Pid = "Product ID: 0x"
        +hex(Pid_Read[0])[2]+hex(Pid_Read[1])[2]+'\n'+"Device name:"
        +chip_ids.get(Pid_num, "Unknown")+'\n'
    return Pid
else:
    return 0
- \_encode_addr: The input is an integer and returns an array with \text{data}[0]: MSB .. \text{data}[4]:LSB and the rest 0's. Used to encode flash memory addresses.

```python
def \_encode_addr(self, addr):
    byte = [0, 0, 0, 0, 0, 0, 0, 0]
    byte[3] = (addr >> 0) & 0xFF
    byte[2] = (addr >> 8) & 0xFF
    byte[1] = (addr >> 16) & 0xFF
    byte[0] = (addr >> 24) & 0xFF
    return (byte)
```

- cmdGo: executes the downloaded program to a specified address. The user must check if the flash memory address corresponds to the specified in MCU datasheet.

```python
def cmdGo(self, addr_go):
    addr_encoded = self.\_encode_addr(addr_go)
    if self.cmdGeneric(cmd=0x21, addr_g=addr_encoded, dlc=4):
        print("*** Go command ***")
    else:
        raise CmdException("Go (0x21) failed")
```

- writeMemory: given a start address and an Intel hex object, it writes the whole file to flash memory in frames of 4 bytes. Returns 1 when it finishes.

```python
def writeMemory(self, addr_w, ih_w):
    pydict = ih_w.todict() \# data in dictionary format
    add=ih_w.addresses() \# addresses read from file
    counter = addr_w \# flash memory start address
    maxdata=len(add)-3 \# last frame will write 4 bytes
    for i \in range (0, int(maxdata), 4):
        nxtadd = self.\_encode_addr(counter) \# address
        [8,0,0,0,0,0,0,0]
        datats = [0, 0, 0, 0, 0, 0, 0, 0]
        for j \in range (0,4):
            datats[j] = pydict[add[i+j]] \# data
        nxtadd[4]=3
        if self.cmdGeneric(0x31, addr_g=nxtadd, dlc=5):
            \# def write(self, id_, msg, flag=0, dlc=None): input to canlib library
            self.ch.write(id=0x31, msg=datats, flag=0, dlc=4)
```
if (self._wait_for_ask("0x31 address failed")):
    self._wait_for_ask("0x31 address failed")
else:
    raise CmdException("Write memory (0x31) failed")

counter = counter + 4

return 1

---

- cmdReadMemory: Given a starting address and an integer, it reads that number of bytes from flash memory and returns it as an array. It reads frames with data of 8 bytes.

```python
def cmdReadMemory(self, addr, nrbyte_total):
    #Input: nrbyte_total is N+1
    #Input: start address in format 0x08000000
    data = []
    assert (1 <= nrbyte_total <= 256)
    addr_encoded = self._encode_addr(addr)
    addr_encoded[4] = nrbyte_total - 1

    if self.cmdGeneric(cmd=0x11, addr_g=addr_encoded, dlc=5):
        nrframes = nrbyte_total // 8
        lastframe = nrbyte_total % 8
        for i in range(0, nrframes):
            temp_frame = self.ch.read(timeout=0xFF)
            data += list(temp_frame[1])
        if lastframe:
            temp_frame = self.ch.read(timeout=0xFF)
            data += list(temp_frame[1][0:lastframe])
        self._wait_for_ask("0x11 address failed")
    assert (len(data) == nrbyte_total)
    #returns list of bytes
    return data
```

---

- verifyMemory: Given an Intel hex object, it calls cmdReadMemory and reads and compares every byte read from flash memory.

```python
def verifyMemory(self, ih_v):
    add_ih = ih_v.addresses()
    pydict = ih_v.todict()
    total_bytes = len(add_ih)
    nr_frames1 = total_bytes // 256
    nr_frames2 = total_bytes % 256
    counter = add_ih[0]
    verified = 1

    for v in range(0, nr_frames1):
```
```python
#read 256 bytes
try:
    data = self.cmdReadMemory(counter, 256)
except Exception:
    print("data cmdReadMemory failed")
    raise CmdException("verifyMemory failed")
for i in range(0, 255):
    temp = counter+i
    if (data[i] != pydict[temp]):
        verified = 0
        print(counter, i)
        break
if not(verified):
    break
counter = counter + 256
if nr_frames2 and verified:
    for i in range(0, nr_frames2):
        data = self.cmdReadMemory(counter, nr_frames2)
        if data[i] != pydict[counter+i]:
            verified = 0
            print(counter, i)
            break

return verified
```

### 4.2. Module usage

To use this application as a main program, it is necessary to read and follow the usage instructions. There are 5 main options to send as arguments to the application. These options are captured by the application by the getopt module:

1. –h: Prints the help instructions.
2. –w : writes to flash memory the specified .hex file. If selected this argument it is mandatory to provide a .hex file.
3. –v: reads flash memory and verifies that it contains byte by byte the same values as the file given. If selected this argument it is mandatory to provide a .hex file.
4. –a: sets a flash memory address. The default value is 0x08000000. If selected this option it is mandatory to give a new address
5. –g: executes the application.
4.3. Future work

There are many possibilities to get use of the code included in this project. Some of them go through developing a new bootloader to correct some of its limitations. A few of them are listed below:

- Add a low cost microcontroller to the same PCB of the ST32 and use of the built-in UART bootloader to program the MCU instead of the Can bus.

- Design of a new CAN bootloader from scratch.

- Adaptation of existing free source open CAN bootloaders to work with ST32 microcontrollers.
5. Programming the graphic user interface

The GUI CanPython is the application based on the Tkinter v8.6 module for Python. It calls the program stm32canloader and initializes the CommandInterface class for using its functions. The application interface is shown in Fig. 17, and consist of a group of widgets (buttons, entries, file openers, etc.) that allows running the mentioned program without typing any command.

5.1. Class APP()

It is the main class of any Tkinter GUI; when the application is executed, it initializes this class and shows in the screen the designed layout. The main widgets and its functions are described below:

1. Load file: The .hex file browser to transfer to the MCU flash memory. If it is not selected, the user cannot execute the ‘write button’ or ‘verify button’.

   ![Browse File]

2. Can Output: It is a text widget to know the action that the user should do in order to continue executing the program. It also shows the information of the product (if requested) and whether the program finished the current task (write or verify). This widget has been taken from the CanSniffer application developed by the professor Juan Manuel Moreno E.[22]
3. Clear screen: The output screen has a scrollbar but if all the information is not necessary, the user can clear the screen.

4. Initialize: it initializes and synchronizes with the MCU. It is mandatory to press this button before using other functions.

5. Flash memory address: the address is read by the program every time the ‘write’ or ‘verify’ buttons are pressed. The default value is 0x08000000 and in order to change it, the user should check the product datasheet.

6. Product ID: it gives the product information as well as commands provided by the manufacturer.

7. Send: it writes a .hex file once it has been selected and the device is initialized.

8. Verify: it reads and checks if the program existent in flash memory is the same as the .hex file selected in browser.

9. Go: runs the existent program in flash memory.
Fig. 17. GUI CanPython interface.
Source: own
5.2. Application flowchart

There are a few steps that the program should follow in order to execute correctly. For example, it is not possible to write the file if the MCU is not connected. Fig. 18 shows the order and meaning of messages showed in the output window:

Fig. 18 Application flowchart.

Source: own
6. Conclusion

1. The Can protocol is a protocol of communications used especially in the automotive sector. Its reliability, flexibility, and high range make it suitable for transferring data between different devices such as ECUs or between a PC and MCUs. It is highly extended and many manufacturers and programmers continuously develop new applications for different purposes.

2. Python is a powerful language that allows us to program, work and integrate different devices in the same application. In this project, it has been developed a stm32canloader program to write Intel hex files in the flash memory of a STM32F407VG microcontroller using a USB-CAN adapter/converter from Kvaser.

3. The main functionalities of this program are sending a hex file to the flash memory, comparison between the program in flash and a given .hex file and execution of the downloaded program.

4. Python also supports the tkinter module, which is an environment to design and program a layout to manage intuitively the main application. The CanPython application was designed and tested and it is programmed in a separated file to facilitate modifications in any of them.

5. The STM32F407VG is a 32-bit microcontroller with many functionalities and between them the capability of writing over the flash memory when it runs in its embedded boot system mode.

6. There are some software restrictions in the MCU that entails some limitations in the developed program. Some of the most relevant are the following:

   6.1. We can only communicate with one MCU at the same time. This means that we cannot connect more than one microcontroller in the same Can network. To program one MCU we have to run it in boot system mode and connect the CAN_high and CAN_low of the transceiver to the Kvaser Leaf HS.

   6.2. In order to write over the flash memory, it has to be erased first because write operations only allow transition of a bit from 1 to 0.

   6.3. The flash memory mass erase is not allowed with this MCU. The process of erase by sectors is not well defined by the manufacturer’s Can Protocol Application Note. In order to erase the needed flash memory sectors we use a free software for UART communication.
7. Acknowledgements

I dedicate this project to every educator I had in my whole life: teachers, professors, parents… because they awakened in me the desire to learn.

I would like to give special acknowledgements to my project director and professor Juan Manuel Moreno Eguilaz, for his support and sharing with me his knowledge and passion for electronics.
8. References

8.1. Basic references


8.2. Complementary references


9. ANNEX.1 Install SDK and canlib library

The final user will transfer the applications via CAN protocol and using a GUI based on python programming. In order to test the CAN communication, we need to install the Kvaser Canlib library.

First of all, download and install Python version 3.6 or later and the Kvaser CANlib SDK library version v5.19 from the official website.

To install the Canlib library type in a windows command prompt

C:\pip install canlib-1.2.163.zip

Notes:

Install the canlib SDK in a temporal file in C:\temp is you are planning to modify and run the samples provided by Kvaser in the same Samples folder. In this case, the library is installed in C:\temp\Canlib_SDK_v5.19

The pip (installer program for python) is already installed with this version of Python.

The environment path for Python should be configured properly when installing it.

To use the pip command be sure to be located in the correct path where canlib-1.2.163.zip is. In this case it's in C:\temp\Canlib_SDK_v5.19\Samples\Python.
10. ANNEX 2 stm32canloader application

# -*- coding: utf-8 -*-
# Can communication between PC and MCU STM32 (tested with STM32F407VG)
# using Kvaser Leaf Light HS (SDK canlib library).
# Copyright (C) 2017 Author: Ricardo Landeo
# <ricardo.landeo@estudiant.upc.edu>
# This file is part of a Master's thesys at Universitat Politècnica de Catalunya.
#
# stm32canloader is free software; you can redistribute it and/or modify
# it under
# the terms of the GNU General Public License as published by the Free
# Software Foundation; either version 3, or (at your option) any later
# version.
#
# stm32canloader is distributed in the hope that it will be useful, but
# WITHOUT ANY
# WARRANTY; without even the implied warranty of MERCHANTABILITY or
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
# for more details.
#
# You should have received a copy of the GNU General Public License
# along with stm32canloader; see the file COPYING3. If not see
# <http://www.gnu.org/licenses/>.

from intelhex import IntelHex
import sys, getopt
import canlib.canlib as canlib
import time

def main():
    # these come from AN2606
    chip_ids = {
        0x412: "STM32 Low-density",
        0x410: "STM32 Medium-density",
        0x414: "STM32 High-density",
        0x420: "STM32 Medium-density value line",
        0x428: "STM32 High-density value line",
        0x430: "STM32 XL-density",
        0x416: "STM32 Medium-density ultralow power line",
        0x411: "STM32F2xx",
        0x413: "STM32F4xx",
    }
    # it's only possible to set a baudrate of 125kb/s
    dict_rate = {
        "1M": canlib.canBITRATE_1M,
        "500K": canlib.canBITRATE_500K,
        "250K": canlib.canBITRATE_250K,
        "125K": canlib.canBITRATE_125K,
        "100K": canlib.canBITRATE_100K,
        "83K": canlib.canBITRATE_83K,
        "62K": canlib.canBITRATE_62K,
        "50K": canlib.canBITRATE_50K,
        "10K": canlib.canBITRATE_10K,
    }

if __name__ == "__main__":
    main()
commands = {
    0x00: "Get version and allowed commands",
    0x01: "Gets the bootloader version",
    0x02: "Gets the chip ID",
    0x03: "Speed commands. canBITRATE_1M = -1. canBITRATE_500K = -2. canBITRATE_250K = -3. canBITRATE_125K = -4. canBITRATE_100K = -5. canBITRATE_62K = -6. canBITRATE_50K = -7. canBITRATE_83K = -8. canBITRATE_10K = -9",
    0x11: "Reads up to 256 bytes of memory",
    0x21: "Jumps to user application code",
    0x31: "Writes to the Flash memory",
    0x43: "Erases from one to all the Flash memory sectors",
    0x63: "Enables the write protection for some sectors",
    0x73: "Disables the write protection for all Flash memory sectors",
    0x82: "Enables the read protection",
    0x92: "Disables the read protection",
}

#Default CAN communication settings:
conf = {
    'baud': '125K',
    'flash_address': 0x08000000,
    'write': 0,
    'verify': 0,
    'go': -1,
}

class CmdException(Exception):
    pass

class CommandInterface:
    
    def openCan(self, chan=0, abaudrate="125K",
                 flagsmode=canlib.canOPEN_ACCEPT_VIRTUAL):  
        #canOPEN_ACCEPT_VIRTUAL: Allows opening of virtual channels as well as physical channels.
        self.cl = canlib.canlib()
        self.ch = self.cl.openChannel(chan, flagsmode)
        print("Setting bitrate to %s" % (abaudrate))
        self.ch.setBusParams(dict_rate[abaudrate])
        print("Going on bus")
        self.ch.busOn()

    def _wait_for_ask(self, info = ""):  
        # wait for ask
        ask = self.ch.read(timeout=0xFF)[1][0]
        if ask == 0x79:
            # ACK
            return 1
        else:
            if ask == 0x1F:
                # NACK
                return 0
            else:
                # Unknown responce
raise CmdException("Unknown response. "+info+":
"+hex(ask))

def initChip(self):
    # Set boot
    self.ch.write(0x79,[0,0,0,0,0,0,0,0],0,0)#timeout=0xFF)
    self.ch.read(timeout=0xFF) #there are 2 ErrorFrames before receiving ACK
    self.ch.read(timeout=0xFF)
    return self._wait_for_ask("Syncro") #ACK sent by the device

def releaseChip(self):
    self.ch.busOff()

def cmdGeneric(self,cmd,addr_g=[0,0,0,0,0,0,0,0],dlc=0):
    self.ch.writeWait(cmd,addr_g,0,dlc,timeout=0xFF)
    return self._wait_for_ask(hex(cmd))

def cmdGet(self):
    stat = self.cmdGeneric(cmd=0x00)
    if stat==1:
        nbytes = self.ch.read(timeout=0xFF)[1][0]
        bootread = hex(self.ch.read(timeout=0xFF)[1][0])[2:]
        #bootread =0x20 is version 2.0

        bootver = bootread[0]+"."+bootread[1]
        commands_str=""  
        for i in range(0,nbytes):
            gcom=self.ch.read(timeout=0xFF) #get commands
            commands_str=commands_str + + hex(gcom[1][0])+":"
        +commands[gcom[1][0]]+\n
        if self._wait_for_ask("0x00 end")==1:
            print("Bootloader version: "+ bootver) #print 2.0
            return commands_str #return list of commands
    else:
        return stat #NACK or 0

def cmdGetVersion(self):
    stat = self.cmdGeneric(cmd=0x01)
    if stat == 1:
        temp = self.ch.read(timeout=0xFF)[1][0]
        bootread = hex(temp)[2:] #bootread =0x20 is version 2.0
        bootver = bootread[0]+"."+bootread[1]
        print(bootver)
        self.ch.read(timeout=0xFF) #dummy byte-> 0x00, 0x00
        self._wait_for_ask("0x01 end")
        return ("Bootloader version: "+ bootver+"\n") #return 3.2
    else:
        return 0

def cmdGetID(self):
    stat = self.cmdGeneric(cmd=0x02)
if stat==1:
    Pid_Read = self.ch.read(timeout=0xFF)[1]  #Product ID
    self._wait_for_ask("0x02 end")

    product ID: 0x\{\text{hex(Pid_Read[0])}\}[2:] + "." + \text{hex(Pid_Read[1])}\}[2:]
    Pid_num = ((Pid_Read[0] << 8) & 0xFF00) + (Pid_Read[1] & 0xFF)

    if Pid_num:
        return Pid
    else:
        return 0

def _encode_addr(self, addr):
    byte = [0, 0, 0, 0, 0, 0, 0, 0]
    byte[3] = (addr >> 0) & 0xFF
    byte[2] = (addr >> 8) & 0xFF
    byte[1] = (addr >> 16) & 0xFF
    byte[0] = (addr >> 24) & 0xFF
    return (byte)

def cmdGo(self, addr_go):
    addr_encoded = self._encode_addr(addr_go)
    if self.cmdGeneric(cmd=0x21, addr_g=addr_encoded, dlc=4):
        print("*** Go command ***")
    else:
        raise CmdException("Go (0x21) failed")

def writeMemory(self, addr_w, ih_w):
    pydict = ih_w.todict()  # data in dictionary format
    add=ih_w.addresses()  # addresses read from file
    counter = addr_w  #flash memory start address
    maxdata=len(add)-3  #last frame will write 4 bytes

    for i in range(0,int(maxdata),4):
        nxtadd = self._encode_addr(counter)  #address
        [8, 0, 0, 0, 0, 0, 0, 0]
        datats = [0, 0, 0, 0, 0, 0, 0, 0]
        for j in range(0,4):
            datats[j] = pydict[add[i+j]]  #data
        nxtadd[4]=3
        if self.cmdGeneric(0x31, addr_g=nxtadd, dlc=5):
            #def write(self, id, msg, flag=0, dlc=None): input to canlib library
            self.ch.write(id=0x31, msg=datats, flag=0, dlc=4)
            if (self._wait_for_ask("0x31 address failed")):
                self._wait_for_ask("0x31 address failed")
            else:
                raise CmdException("Write memory (0x31) failed")

        counter = counter + 4
        return 1

def cmdReadMemory(self, addr, nrbyte_total):
    #Input: nrbyte_total is N+1
    #Input: start address in format 0x08000000

    byte = [0, 0, 0, 0, 0, 0, 0, 0]
    byte[3] = (addr >> 0) & 0xFF
    byte[2] = (addr >> 8) & 0xFF
    byte[1] = (addr >> 16) & 0xFF
    byte[0] = (addr >> 24) & 0xFF
    return (byte)
Can bus bootloader for the STM32F407VG

data=[]
assert(1<=nrbyte_total <= 256)
addr_encoded = self._encode_addr(addr)
addr_encoded[4] = nrbyte_total - 1

if self.cmdGeneric(cmd=0x11, addr_g=addr_encoded, dlc=5):
    nrframes = nrbyte_total // 8
    lastframe = nrbyte_total % 8
    for i in range(0,nrframes):
        #read 8 bytes
        temp_frame = self.ch.read(timeout=0xFF)
        data+=list(temp_frame[1])
    if lastframe:
        temp_frame = self.ch.read(timeout=0xFF)
        data+=list(temp_frame[1][0:lastframe])
    self._wait_for_ask("0x11 address failed")
assert(len(data) == nrbyte_total)
#returns list of bytes
return data

def verifyMemory(self, ih_v):
    add_ih = ih_v.addresses()
    pydict = ih_v.todict()
    total_bytes = len(add_ih)
    nr_frames1 = total_bytes // 256
    nr_frames2 = total_bytes % 256
    counter = add_ih[0]
    verified = 1
    for v in range (0,nr_frames1):
        #read 256 bytes
        try:
            data = self.cmdReadMemory(counter, 256)
        except Exception:
            print("data cmdReadMemory failed")
            raise CmdException("verifyMemory failed")
        for i in range(0,255):
            temp = counter+i
            if (data[i] != pydict[temp]):
                verified = 0
                print(counter,i)
                break
        if not(verified):
            break
        counter = counter + 256
    if nr_frames2 and verified:
        for i in range(0,nr_frames2):
            data = self.cmdReadMemory(counter, nr_frames2)
            if data[i] != pydict[counter+i]:
                verified = 0
                print(counter,i)
                break

    return verified

def __init__(self):
    pass
def usage():
    print ("""Usage: %s [-hVvwbag] [-a addr] [-g addr] [file.hex]
    -h This help
    -V Verbose
    -w Write
    -v Verify
    -a addr Target address to write and start running at. See
    product datasheet (Flash memory, 0x08000000 for STM32F407VG)
    ./stm32canloader.py -w -v example/main.hex
    """ % (sys.argv[0]))

if __name__ == "__main__":

    #To use this program type for example:
    #***** stm32canloader.py -w Test.hex *****
    #in windows command prompt

    # https://docs.python.org/3.6/library/getopt.html

    try:
        opts, args = getopt.getopt(sys.argv[1:], "h\wva:g")
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err) # will print something like "option -a not recognized"
        usage()
        sys.exit()

    print(opts, args)
    for o, a in opts:
        if o == '-h':
            usage()
            sys.exit()
        elif o == '-w':
            conf['write'] = 1
        elif o == '-v':
            conf['verify'] = 1
        elif o == '-a':
            conf['flash_address'] = eval(a)
        elif o == '-g':
            conf['go'] = 1
        else:
            assert False, "unhandled option"

    cmd = CommandInterface()  
    try:
        try:
            cmd.openCan(abaudrate=conf['baud'])
        except Exception:
            print('Use one of the following baudrates')
        print(dict_rate)
        sys.exit()  
        try:
            cmd.initChip()  
        except Exception:
print ("Can't init. Ensure that BOOT1 and BOOT0 are connected and reset device")
    sys.exit()

bootversion = cmd.cmdGet()
Pid = cmd.cmdGetID()  # product ID
if (conf['write'] or conf['verify']):
    try:
        ih = IntelHex(args[0])
    except Exception:
        print("File not found. Provide .hex file")
        sys.exit()

if conf['write']:
    cmd.writeMemory(addr_w = conf['flash_address'], ih_w = ih)

if conf['verify']:
    try:
        stat = cmd.verifyMemory(ih_v=ih)
        if stat==1:
            print("File OK")
        elif stat == 0:
            print("File doesn't match")
    except Exception:
        print("Check cmdReadMemory. Error")

if conf['go'] != -1:
    cmd.cmdGo(addr_go = conf['flash_address'])

finally:
    cmd.releaseChip()
10. ANNEX 3: CanPython application

# CAN flasher from Python & Tkinter
# Python 3.6, tkinter 8.6
# Ricardo Landeo
# June 2017

import tkinter as tk
from stm32canloader import *
from cantextwidget import *
from tkinter.filedialog import askopenfilename
from intelhex import IntelHex
import time

# these come from AN2606
chip_ids = {
    '0x412': "STM32 Low-density",
    '0x410': "STM32 Medium-density",
    '0x414': "STM32 High-density",
    '0x420': "STM32 Medium-density value line",
    '0x428': "STM32 High-density value line",
    '0x430': "STM32 XL-density",
    '0x416': "STM32 Medium-density ultralow power line",
    '0x411': "STM32F2xx",
    '0x413': "STM32F4xx",
}

# only 125K available
dict_bitratevalues = {
    '1M': -1,
    '500K': -2,
    '250K': -3,
    '125K': -4,
    '100K': -5,
    '62K': -6,
    '50K': -7,
    '83K': -8,
    '10K': -9,
}

class App():
    def __init__(self):
        self.root = tk.Tk()
        self.initialize()
        self.root.mainloop()

    def initialize(self):
        self.inivariables()
        self.iniGUI()
        self.update_screen()

    def inivariables(self):
        self.baudrate = '125K'

    def iniGUI(self):
self.root.title('CAN Flasher from Python \ UFC \ 2017')
self.root.resizable(False, False)
self.left = tk.Frame(self.root, padx=1, pady=1)  # Left Panel
leftline = "Select .hex file"

self.fname = ''
self.framebutton = tk.Frame(self.left)
Browse_button = tk.Button(self.framebutton, text="Browse",
command=self.load_file, width=10, padx=1, heigh=1)
Browse_button.pack(side=tk.LEFT, anchor=tk.W, fill=tk.Y, padx=1,
expand="yes")

self.fileText = tk.Text(self.framebutton, font=(0, 8, 'normal'), height=1, width=30)
self.fileText.pack(side=tk.LEFT, anchor=tk.W, padx=15)
self.fileText.config(state='disabled')
self.framebutton.pack(side=tk.TOP, expand=1, fill=tk.X, pady=1,
padx=1)

self.screenline = 'Open a .hex file and select command\n'

self.text = textwidget(self.left, 'Can flasher Output',
tk.TOP, width=50)

Init_button = tk.Button(self.left, text="Initialize CAN",
command=self.iniCAN, width=15, heigh=1)
Init_button.pack(side=tk.LEFT, anchor=tk.W, fill=tk.Y,
expand="yes")
Clear_button = tk.Button(self.left, text="Clear screen",
command=self.Text.clearscreen, width=15, heigh=1)

Clear_button.pack(side=tk.RIGHT, anchor=tk.W, fill=tk.Y,
expand="yes")
self.left.pack(side=tk.LEFT, fill=tk.Y)
self.right = tk.Frame(self.root)  # Right Panel

self.labeladdress = tk.LabelFrame(self.right, text = 'Flash memory address')

self.address = tk.Frame(self.labeladdress)
tk.Label(self.address, text='Addr').pack(side=tk.LEFT, padx=1)
self.addr = tk.StringVar()  # variable for address
tk.Entry(self.address, textvariable=self.addr, bg='white',
width=10).pack(side=tk.LEFT, padx=1)
self.addr.set('0x08000000')
self.address.pack(side=tk.TOP, expand=1, pady=1,
padx=1, fill='x')
self.labeladdress.pack(side=tk.TOP, expand=1, pady=1,
padx=1, fill='x')

self.prosend = tk.Frame(self.right)
tk.Button(self.prosend, text = 'Product ID', command =
self.ProdID, width=10).pack(side=tk.LEFT, padx=20)
tk.Button(self.prosend, text = 'Send', command = self.SendCAN,
width=10).pack(side=tk.LEFT, padx=20)
def finish(self):
    print('Bus Off
    self.screenline = "Bus Off"
    self.update_screen()
    try:
        self.cmd.releaseChip()
    except:
        pass
    self.root.destroy()

def load_file(self):
    self.fileText.config(state='normal')
    self.fileText.delete(1.0, 2.0)
    self.fname = askopenfilename(filetypes=(("Hex files", ".hex"),
                                         ("All files", "*.*")))
    if self.fname == '':
        print("None selected")
    else:
        self.fileText.insert(tk.END, self.fname)
    self.fileText.config(state='disabled')
    print(self.fname)

def getaddress(self):
    return self.is_number(self.addr.get())

def is_number(self, s):
    try:
        a = int(s, 0)
        return a
    except ValueError:
        a = 0
        return a

def SendCAN(self):
    if self.fname != '':
        try:
            ih = IntelHex(self.fname)
        except Exception:
            self.screenline = "File not found. Provide .hex file"
            self.update_screen()
            try:
                stat = self.cmd.cmdGet() #Returns 0 if there is no connection
                if stat != 0:
                    # Handle error or proceed with operation
            except Exception:
                self.screenline = "Connection error."
                self.update_screen()
            else:
                # Proceed with CAN send operation
self.screenline = "Sending code to flash. Wait some
seconds\n"
self.update_screen()

temp_add = self.getaddress()
self.cmd.writeMemory(addr_w = temp_add, ih_w = ih)
self.screenline = "File transfer completed\n"
self.update_screen()
else:
    self.screenline = "Restart Kvaser and MCU\n"
    self.update_screen()
except Exception:
    self.screenline = "Connect MCU and initialize hardware\n"
    self.update_screen()
else:
    self.screenline = "Select a .hex file\n"
    self.update_screen()

def GOapp(self):
    try:
        stat = self.cmd.cmdGet()  #Returns 0 if there is no connection
        if stat == 1:
            self.screenline = "Executing downloaded code\n"
            self.update_screen()
            self.cmd.cmdGo(addr_go = self.getaddress())
        else:
            self.screenline = "Restart Kvaser and MCU\n"
            self.update_screen()
    except Exception:
        self.screenline = "Connect MCU and initialize hardware\n"
        self.update_screen()

def iniCAN(self):
    self.screenline = 'Initializing CAN\n'
    self.text.writenewline(self.screenline)
    self.screenline = 'Setting bitrate to ' + self.baudrate + '\n'
    self.text.writenewline(self.screenline)
    self.cmd = CommandInterface()

    self.cmd.openCan(abaudrate = self.baudrate)  #initialize library
    try:
        stat = self.cmd.initChip()  #Synchro byte
        if stat:
            self.screenline = 'ACK received. Connected to MCU\n'
            self.text.writenewline(self.screenline)
        else:
            self.screenline = "Device not found. Ensure that BOOT1 and BOOT0 are connected and reset device twice\n"
            self.text.writenewline(self.screenline)
    except Exception:
        self.screenline = "Device not found. Restart MCU at least twice and initialize again\n"
        self.text.writenewline(self.screenline)

    def update_screen(self):
        self.text.writenewline(self.screenline)

    def ProdID(self):
        try:
            stat = self.cmd.cmdGet()  #Returns 0 if there is no connection
if stat != 0:
    bootversion = self.cmd.cmdGetVersion()
    self.screenline = bootversion
    self.text.writenewline(self.screenline)
    Pid = self.cmd.cmdGetID()  # product ID
    self.screenline = Pid
    self.text.writenewline(self.screenline)
    self.screenline = "***Commands***
    self.text.writenewline(self.screenline)
else:
    self.screenline = "Restart Kvaser and MCU\n"
    self.text.writenewline(self.screenline)
except Exception:
    self.screenline = "Connect MCU and initialize hardware\n"
    self.text.writenewline(self.screenline)

def CanVerify(self):
    try:
        ih = IntelHex(self.fname)
    except Exception:
        self.screenline = "File not found. Provide .hex file\n"
        self.update_screen()
        return 0
    try:
        self.screenline = "Verifying. Wait a moment\n"
        self.update_screen()
        time.sleep(0.1)
        stat = self.cmd.verifyMemory(ih_v=ih)
    except Exception:
        self.screenline = "Error in verify. Initialize CAN first\n"
        self.update_screen()
        return 0
    if stat == 1:
        self.screenline = "File OK\n"
    elif stat == 0:
        self.screenline = "File doesn't match\n"
        self.update_screen()