Flashing ECU software over-the-air using

Raspberry Pi

-documentation-

Table of contents:

1.[Overview of the project](#project_overview)

2.[Prerequisites](#prerequisites)

3.[Server-side application](#server_side)

4.[Client-side application](#Client_4)

5.[Graphical User Interface](#graphical_user)

6.[SPI](#flash_spi)

7.[Bibliography](#bibliography)

1. Overview of the project

The main goal of this project is to send an already compiled file wirelessly to a Raspberry PI, that will further relay it to an Atmega 328 microcontroller. Once the file has been fully received by the target Raspberry PI, the device will use its GPIO pins to direct the program to the connected microcontroller, that will execute the code.

1. Prerequisites

On the client-side (the device that will send the program), the user must have the means to compile the C-written program into a HEX file, and thus, the Arduino IDE is required. The software can be found [here](https://www.arduino.cc/en/software).

In order to have a working connection between the client-side and server-side part of the program, the server (the Raspberry that will receive the file) must be connected on the same local network that the client-side is.

The configuration of AVRDude software will be presented later, on the sixth chapter of the documentation, but it is necessary in the configuration stage that the receiving side is connected to the internet, such that the required software can be downloaded.

For the connection between Raspberry PI and Atmega 328 microcontroller, a breadboard and some wires are required.

3.Server-side application

The code has been separated into three files: the server, the client and the GUI.

As mentioned in the previous chapters, the server-side of the program represents the part that receives the code to be transmitted to the microcontroller. In the first phase, the program will automatically get the host IP. The port that will be used is hard-coded into the program and must be changed manually if the port will be used already by another software. Some of the program’s capabilities are:

-the possibility of receiving text messages from the connected client

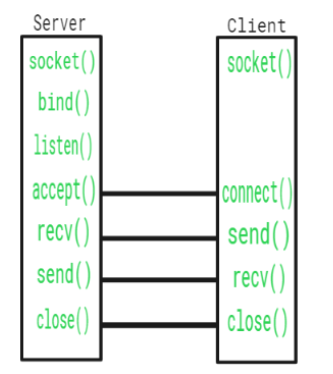
-the receiving of already-compiled C code files that will be further sent to the microcontroller

-the ability to continue the acquiring of above-said file, when the connection between the client and the server has been closed mid-transfer, provided that the client has reconnected.

-the ability to flash the microcontroller with the given firmware

-checking the validity of the file both when it receives it and when it has finished sending it to the Atmega device

The program has taken an object-oriented approach, for better readability and for future improvements to be added with ease. Thus, the " ServerSideServerDevice " class has been defined, which contains the majority of the code. The application relies heavily on the "socket" module. This module is used to connect devices through sockets. A socket is one endpoint of a two-way communication link between two programs running on the network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication take place.



|  |  |
| --- | --- |
| Function | Description |
| Create() | Creates a socket |
| Bind() | Socket identification |
| Listen() | Ready to receive a connection |
| Connect() | Ready to act as a sender |
| Accept() | Accepts the connection from sender |
| Write() | Sends data |
| Read() | Receives data |
| Close() | Closes the connection |

In addition to the “socket” module, the program also relies on the “select” module that calls the OS **select** function, which determines the status of one or more sockets, waiting if necessary, to perform synchronous I/O. The application also adopts threading concepts, using three threads : the main one, **thread\_SaSr** (“Thread Server Accept, Server Receive”) and **thread\_Ss** (“Thread Server Send”). Finally, the server-side uses the “subprocess” module, so when the transfer has been finished, the application will use Linux terminal commands to load the compiled code to the Atmega.

At the beginning of the class definition two constants stand out: HEADER\_LENGTH and BUFFER\_SIZE. HEADER\_LENGTH represents the length of the header, which is a sequence of the form "message-length\_acknowledgement\_meta-file\_file-packet" (ex: "4096\_0\_0\_1"). BUFFER\_SIZE represents the amount of data (in bits) that will be sent/received at once, in one packet. First, the header is received, conveying information about the actual data that will follow.

To further detail the header:

|  |  |
| --- | --- |
| message\_length | contains the size of the message |
| acknowledgement | when true, tells the other party(the client) that the message has been received succesfully |
| meta\_file | when true, the message contains information on the file about to be received/sent |
| file\_packet | when true, the message represents a file packet(is a part of a file) |

Last two can only be "1" when a file transfer takes place. Last three can't be "1" simultaneously.

For a two-way connection between client and server, the "**send\_message**" and "**receive\_message**" are to be found in both client and server.

In the “**send\_message**” function:

-header is initialized with the length of the message and then the other three variables are added in a string format;

-if the message is a file packet, only the header is encoded using “**UTF-8**”, otherwise, the full message is encoded that way;

-the message is sent to the client socket; the function returns a true value if no socket errors are raised, otherwise returns false;

"**recvall**" is a helper function to receive n bytes or return None if EOF is hit.

"**receive\_message**" function reads all the data sent by the client socket. It verifies if the client closed the connection by checking the length of the header(if it's 0 the user has disconnected). It extracts the information from the header and returns it or if an error has been encountered, the value returned is False.

"**save\_packet**" writes the packets on the local storage.

"**ft\_check**" checks previous connections of the current client to the server by comparing the current username to an username saved in the *file\_dict* dictionary. If the comparison returns True, then the “**size**” key in the dictionary is checked, and if it’s bigger than 0, the program figures out that there are still packets to be send, and the transfer will take place from that packet number (“size” divided by BUFFER\_SIZE).

The "**server\_accept\_and\_receive**" function manages the flow of information between server and client, using the "**select**" module. "**select.select**" returns a list of sockets that we can read from or sockets that raise an exception. If the "**notified\_socket**" is the "**server\_socket**", then a client has requested to connect. The following if statement verifies if the connection has ended before the username has been received. Then, the client socket is appended to the sockets list, *event\_Vsl* will be set, if it wasn't already and "**ft\_check**" will be called.

If the notified socket is not the server socket, then the server will receive a message from a client. It will verify if the message is not empty (otherwise it will remove the socket from the "**sockets\_list**" and the "**client\_dict**", and clear the *event\_Vsl* event). Furthermore, the program will verify what type of message it has just received, and proceed accordingly.

If it's just a text message, it will display it in the console.

If it's an acknowledgement, it will display it specifically as it is.

If it's metadata about the file, it will store the data into the *file\_dict*('username','name','size','flash'), and then if the name of the file already exist, the old file will be overwritten.

Finally, if it's a file packet, the server will call the "**save\_packet**" function, and then out of the "**size**" variable, the BUFFER\_SIZE will be subtracted out of it. When "**size**" gets equal or below zero, the transfer has finished, an acknowldgement will be sent back to the client-side. Ultimately, the program will verify if the client had the intent: to flash the file. If that's the case, then "*AVR\_FLASH\_STATE*" will be set to True, and "**AVR\_FLASH\_client\_socket**" will further hold the client’s socket for future acknowledgements. At the end of the function, the program verifies if there are any sockets that have raised any exception, and displays a message before closing them, if any.

“**server\_send**” is a function with a thread of its own, due to some blocking functions that reside inside it:

-self.event\_Vsl.wait() – waits for *sockets\_list* to contain client sockets

-self.event\_Ss.wait() – waits for *username* to be set from another thread

“**server\_send**” refreshes username list in case a new user has been added, checks if the username is connected to the server, and if so gets the client socket to send a message from socket: username dict (assuring that the message is not an empty string). Finally, it clears *event\_Ss* flag to prepare for another request from another thread to send messages.

“**send\_to\_client**” is a helper functions that sets the *event\_Ss* flag.

In the main function:

* The sleep function is active for 500 miliseconds, in every iteration of the while loop
* If AVR\_FLASH\_STATE is True, then the flashing process will take place
* The AVRDude command is executed using “subprocess” module
* Checks if the flashing has been done correctly
* If RPI\_SPI\_EN is true, the SPI flashing process will begin
* If successful, the following message will be printed: "SPI: Successfully received acknowledgement from ECU."

**4. Client-side application**

**Class “ClientSideClientDevice”**

The structure on which our client-side application rely on is class “**ClientSideClientDevice**”. The class implements the client side for our network communication: allows us to connect to a server, send messages, send files to the server and impose flashing the files to the ECU connected to the server.

**Methods:**

**\_\_init\_\_:**

def \_\_init\_\_(self, username: str, family=socket.AF\_INET, sock\_type=socket.SOCK\_STREAM)

**Parameters:**

1. username

The username which our client will identify with from now on.

1. family (optional)

The family of the socket, default as Address Family Internet

1. sock\_type (optional)

The type of socket, defulat as SOCK\_STREAM for TCP socket.

**Description:**

When instancing class “ClientSideClientDevice”, an username must be given as parameter. The necessary variables which the client class will work with are initiated within \_\_init\_\_ method. Furthermore, the secondary thread of our client application, called “thread\_Cr” (thread client receive) will be started once class is instanced.

**set\_address:**

def set\_address(self, ip, port):

**Parameters:**

1. ip

The ip address of the server to which client will connect (Ex: 10.0.0.1)

1. port

The port of the server to which client will connect (Ex: 1235)

**Description:**

Verify if the IP and PORT given as parameters abide internet protocol (they are valid). If they are, then store them for later use in “SERVER\_IP” and “SERVER\_PORT” class variables.

**connect:**

def connect(self):

**Parameters:**

**Description:**

Creates the socket object and uses it to connect to the server. Within the method the following conditions are verified:

1. Client must have a valid address to connect to. “SERVER\_IP” and “SERVER\_PORT” must have been configured. “set\_address” method must be called before calling “connect”
2. Client must not already be connected to a server when trying to connect.
3. Client must not already be trying to connect to a server then calling “connect”.

If client successfully connects, it will instantly send its username to the server. Server will now on identify us by our username.

If client fails to connect, “socket.error” exception will occur. After three exceptions, client will no longer try to connect.

**connect\_thread:**

def connect(self):

**Parameters:**

**Description:**

Calls “connect” method in a secondary thread. The reasoning behind it is given by the conflict created by the function call “self.socket.connect((self.SERVER\_IP, self.SERVER\_PORT))” which is a blocking function and the GUI. For GUI to work smoothly it needs to control the flow of the main thread. When pressing “Connect” button within our application, if “connect” was called instead of “connect\_thread”, the main thread will get blocked for some seconds and the GUI will become unresponding.

**reconnect:**

def reconnect(self):

**Parameters:**

**Description:**

There are 2 situations in which client is disconnected from server.

1. Client willingly disconnected by pressing “Disconnect”.
2. Client got disconnected unwillingly by signal loss, server restart… When client is disconnected by one of those events, client will try to reconnect to the server.

**disconnect:**

def disconnect(self):

**Parameters:**

**Description:**

Forcefully disconnect client from server by pressing “Disconnect”.

**send\_message:**

def send\_message(self, client\_socket, msg, encoding="utf-8", acknowledgement: bool = False, meta\_file: bool = False, file\_packet: bool = False):

**Parameters:**

1. client\_socket

The socket object of the client which was used to connect to the server

1. msg

The useful information to be sent to the server. May be a string of a byte array. String in case of a message, byte array in case of file transfer.

1. encoding

When sending a string through the network using TCP, we must convert it into bytes. We can only send bytes. Encoding selects the type of bytes the message will be converted in (Ex: utf-8 bytes).

1. acknowledgement

A component of the header of our full message. When true, it means we’ve receive a message from the server and that we want to acknowledge that message. We send a message of acknowledgement.

1. meta\_file

A component of the header of our full message. When true, it tells the server that our msg contains important information about the file we want to transmit.

1. file\_packet

A component of the header of our full message. When true, it tells the server that msg is not a string and it’s a byte array of “BUFFER\_SIZE” length or less(for the last file packet). When a file is transmitted, it is divided into multiple packets using the following formula:

If filesize is not a multiple of BUFFER\_SIZE, last packet will be smaller than BUFFER\_SIZE.

**Description:**

Process the message we want to send to the server and creates a full message. The full message is a header-message structure. The header contains information about the message itself. After processing the message, it sends the message using the socket object.

full\_msg

msg

header

The server knows about how the message is structured and will extract the information about the message from the header and interpret it as needed. Very important is the length of the header. Both client and server must have the same “HEADER\_LENGTH” to be able to communicate. When a message is sent, the header of fixed length will first be received (Ex of header: “6\_0\_0\_0 ”). Server will extract the fact that the message itself has length 6 and will receive it all. (Ex of full message: “6\_0\_0\_0 Hello!”)

**recvall:**

def recvall(sock, n):

**Parameters:**

1. sock

Socket object used to receive data

1. n

The size of the message to receive in number of bytes.

**Description:**

Fully receive a message through TCP.

**Return:**

Returns: A bytearray containing the received bytes.

**receive\_message:**

def receive\_message(self, client\_socket, encoding='utf-8'):

**Parameters:**

1. client\_socket

The socket object of the client which was used to connect to the server

1. encoding

Encoding used by the server when sending messages.

**Description:**

Receive a full message. A full message is formed by a header bound by a string or bytearray.

Firstly, receive the header. Process the header.

Lastly, receive the message itself.

**Return:**

Returns: A dictionary containing all the information about the message for the receiver to interpret.

**THREAD - receive\_message\_thread:**

def receive\_message\_thread(self):

**Parameters:**

**Description:**

A thread started when instancing “ClientSideClientDevice” class. The flow of the thread is blocked by the method “receive\_message” which contains the function “recvall” which is a blocking function. Blocking functions creates conflicts with the main thread, where the GUI is, thus the necessity of the thread.

Try receiving messages from server. Detect if server triggered a disconnect and try to reconnect.

When a message is successfully received, process it according to the information found within the header.

**send\_msg:**

def send\_msg(self, message):

**Parameters:**

1. message

The message to send.

**Description:**

Check if connected to the server then send the message. If failed to connect ,try reconnecting.

**send\_message\_thread:**

def send\_message\_thread(self, message):

**Parameters:**

1. message

The message to send.

**Description:**

Call “send\_message”method within a thread. Because “send\_message” may call “reconnect” then it means it can block the flow of the program thus starting a secondary thread when sending a message is required.

**send\_file:**

def send\_file(self, filename, flash\_ecu=False):

**Parameters:**

1. filename

The name of the file to send.

1. flash\_ecu

Flag that will be used by server to check if he have to flash the ECU or not.

**Description:**

Send a file to the server.

Break the file into packets with size equal to BUFFER\_SIZE. Send the packets to the server sequentially. Program will return from function only when the whole file has been sent.

If a connection issue occurred and client got disconnected from the server, when reconnected, the client will get a message from server with the amounts of packets left to transmit. The transmission will resume from where the connection has been lost.

**send\_file\_thread:**

def send\_file\_thread(self, filename, flash\_ecu=False):

**Parameters:**

1. filename

The name of the file to send.

1. flash\_ecu

Flag that will be used by server to check if he have to flash the ECU or not.

**Description:**

Method starts “send\_file” method in a thread.

**gui\_update\_constatus:**

def gui\_update\_constatus(self):

**Parameters:**

**Description:**

Method used exclusively to reference the ClientSideClientDevice class from within the GUI.

Visually updates the GUI whenever using the “Configure Network” menu within the GUI.

**5.****Graphical User Interface**

The GUI of our application can be found within “AppGui.py” file in our project. The module used in order to create the graphic interface is “customtkitner” made by TomSchimansky. Also, “customtkinter” is based on “tkinter” module thus most of the time methods working with tkinter also work with customtkitner.

**Getting into customtkinter…**

The roots at the base of the GUI have been developed learning from 2 docs from TomSchimansky’s github:

1. <https://github.com/TomSchimansky/CustomTkinter/wiki/App-structure-and-layout>
2. <https://github.com/TomSchimansky/CustomTkinter/blob/master/examples/complex_example.py>

The first doc gives us a sense of how GUIs are being structured in CTk(customtkinter). Summarily there are 2 main ideas to take into consideration:

* The code for our GUI should be structured within classes for better readability.
* The structure of our GUI should be made using .grid() method instead of .place() or .pack().

**Learning about .grid():**

The .grid() method allows us to organize CTk elements in a grid-like manner. There are 2 steps that need to be made when creating a CTk element:

* Creating an instance of the CTk element, for example of a textbox

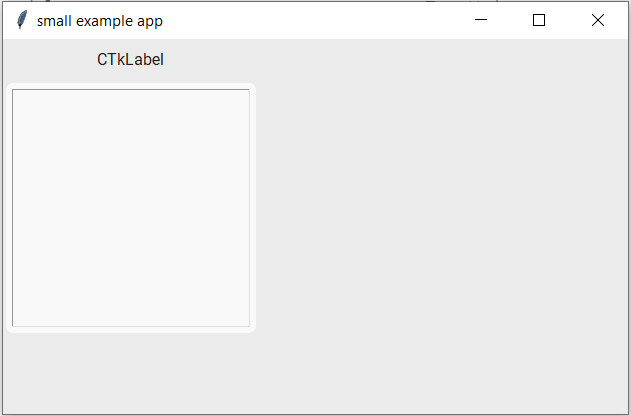
self.textbox = customtkinter.CTkTextbox(master=self)

* Displaying the instance of the CTk element using .grid()

self.textbox.grid(row=0, column=0, columnspan=2, padx=20, pady=(20, 0), sticky="nsew")

**Example** – Understanding grid configuration and sticky property:

Let’s say you have a 1x2 grid system that contains a CTk label on row 1 and a CTk textbox on row 2. Without any configuration the application will look empty, the elements will not fill the whole window as the rows and the column will occupy just enough space to fit the default resolution of the CTk elements (Fig.5.1.).

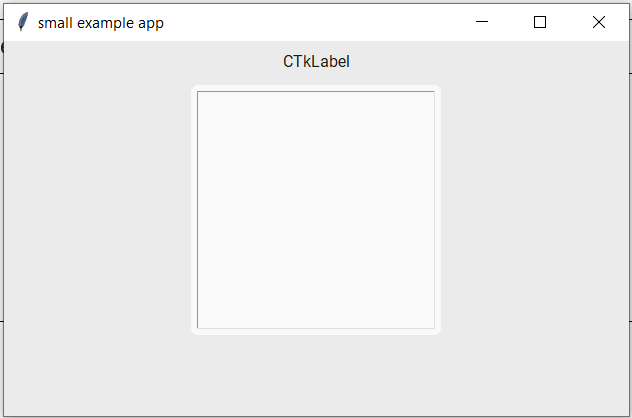


*Fig.5.1. Grid system without configuration.*

Initially we will first make the elements occupy the whole column thus we will uncomment the following instruction which will impose that the only column in our program will occupy the whole space it’s given by the application:

self.grid\_columnconfigure(0, weight=1)

As seen in figure 5.2. the column now occupies the whole space, but the elements kept their default resolution. In order to stretch the elements, we will add the sticky property which allows them to stick to left and right or west and east.

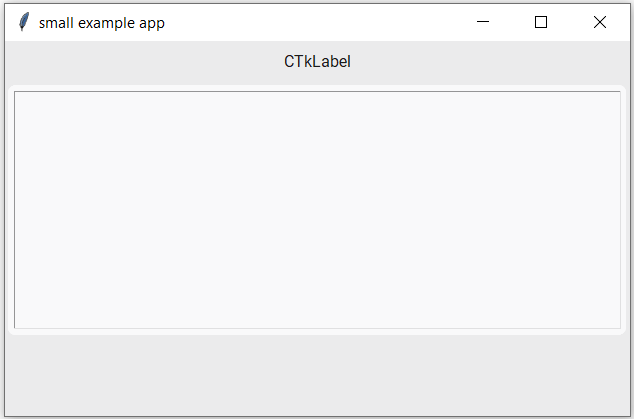


*Fig.5.2. Grid system with configured column*

To make CTk elements sticky we will modify the following code lines:

[…]  
self.label\_title.grid(row=0, column=0, padx=2, pady=2, sticky=”ew”)  
[…]  
self.textbox.grid(row=1, column=0, padx=2, pady=2, sticky=”ew”)

After modifying the code lines, we successfully made our CTk elements fill the whole column they’ve been placed on(Fig.5.3.).



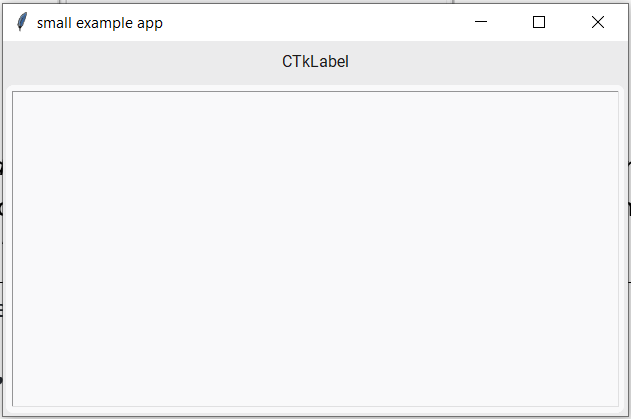
*Fig.5.3. Grid system with configured column and sticky elements*

We may wish to make the textbox occupy the whole space to the bottom of the interface. We’ll configure the row it is in and we’ll modify it’s sticky property.

self.grid\_rowconfigure(1, weight=1)

[…]  
self.label\_title.grid(row=0, column=0, padx=2, pady=2, sticky=”new”)  
[…]  
self.textbox.grid(row=1, column=0, padx=2, pady=2, sticky=”nsew”)

Now the application looks exactly like we wanted (Fig.5.4.).



*Fig.5.4. Grid system with configured columns and rows*

In case of the columns, there was no other column to configure, but for the rows there was row 0 left to configure. By default rows and columns have a default weight of 0. When one row has a weight of 1 and another one has a weight of 0, the one with weight 0 will occupy as much space as the CTk elements need to fit based on their resolution and the one with weight 1 will occupy the left space. If we display a label in row 0 it will occupy less space that a textbox, thus the remaining space for row1 would be smaller for the latter

In addition, it is possible to have an element with weight higher than 1. Let’s say we have a 1x3 grid system where row0 has weight 0, row1 has weight 1 and row2 has weight 5. As we previously said, row0 will occupy as much space as needed to fit the CTk element. The left space will be occupied based on the weight magnitude as follows:

* Row1 will occupy of the available space
* Row2 will occupy of the available space because it’s weight is 5 times higher that row1’s thus the occupied space should be 5 time bigger than row1’s

Code for unmodified application:

import customtkinter  
  
class App(customtkinter.CTk):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 self.geometry("500x300")  
 self.title("small example app")  
  
 # create 2x1 grid system  
 # self.grid\_columnconfigure(0, weight=1)  
 # self.grid\_rowconfigure(1, weight=1)  
  
 self.label\_title = customtkinter.CTkLabel(master=self)  
 self.label\_title.grid(row=0, column=0, padx=2, pady=2)  
  
 self.textbox = customtkinter.CTkTextbox(master=self)  
 self.textbox.grid(row=1, column=0, padx=2, pady=2)  
  
 def button\_callback(self):  
 self.textbox.insert("insert", self.combobox.get() + "\n")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app = App()  
 app.mainloop()

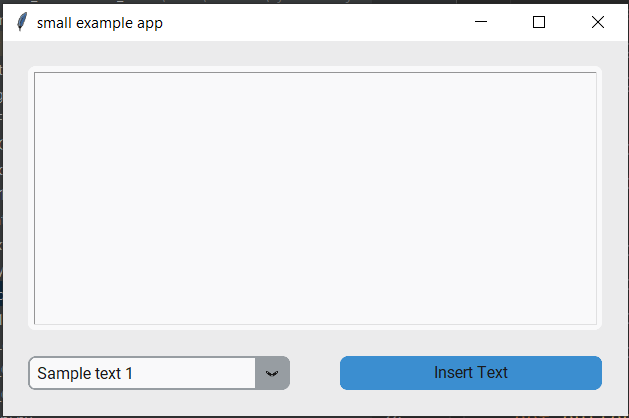
**Analyzing first doc – a GUI with .grid**

Inside the first doc, the “Small onefile example app with .grid” section documents an implementation of a 2x2 grid system.

From reading the code we can tell that we have a 2x2 grid system GUI where the first row (row0) will occupy the remaining space to display the CTk.Textbox and row1 will occupy the needed space for the CTk.ComboBox and CTk.Button. Both columns weight are set on 1 so columns will both get half the application resolution. We expect the application to look like the table below:

|  |  |
| --- | --- |
| CTk.Textbox | |
| CTk.Combobox | CTk.Button |

As seen in Fig.5.5. the application looks as expected:



*Fig.5.5. 2x2 Grid system*

**Analzying a complex customtkinter example**

The following example can be found on github at the following address: <https://github.com/TomSchimansky/CustomTkinter/blob/master/examples/complex_example.py>

When creating an instance of a CTk object you must select the frame in which it will be displayed within. Until now, we placed our CTk elements within “self” which is the main frame of our application. One of the best practices is to separate our main frame in multiple frames. In the complex example we will discover 2 frames within the main frame: “frame\_left” and “frame\_right”. As frames are also CTk elements, they are created using the same 2 steps: Instancing and displaying.

# ============ create two frames ============  
  
# configure grid layout (2x1)  
self.grid\_columnconfigure(1, weight=1)  
self.grid\_rowconfigure(0, weight=1)  
  
self.frame\_left = customtkinter.CTkFrame(master=self,  
 width=180,  
 corner\_radius=0)  
self.frame\_left.grid(row=0, column=0, sticky="nswe")  
  
self.frame\_right = customtkinter.CTkFrame(master=self)  
self.frame\_right.grid(row=0, column=1, sticky="nswe", padx=20, pady=20)

The most important advantage when having multiple frames is that we can also have multiple grid systems. One complex grid system can be broken into simpler grid systems found in multiple frames.

Let’s analyze “create two frames” section within our example. The two CTk.Frames has both been assigned the master of “self”. It means they are both placed within the main frame and that both are sharing the same grid system.

The grid system created within the main frame is 2x1. The row will occupy the whole space the application has. Second column will fill the remaining space the application has.

Both frames has sticky on all directions “nswe” – north south east west. It means the frames will stretch to fill the rows and columns.

**Analyzing “frame\_left” – 1x11 Grid System**

* Row 0 – Spacing of 10px between top frame border and Row 1.

self.frame\_left.grid\_rowconfigure(0, minsize=10) # empty row with minsize as spacing

* Row 1 – Display of a CTk.Label
* Row 2 – Display of a CTk.Button
* Row 3 – Display of a CTk.Button
* Row 4 – Display of a CTk.Button
* Row 5 – Used for Spacing of the remaining space

self.frame\_left.grid\_rowconfigure(5, weight=1) # empty row as spacing

* Row6,7 – Unused. Grid system behaves like they don’t exist.
* Row 8 – Spacing of 20px

self.frame\_left.grid\_rowconfigure(8, minsize=20) # empty row with minsize as spacing

* Row9 – Display of a CTk.Label
* Row10 – Display of a CTk.OptionMenu
* Row11 – Spacing of 10px

self.frame\_left.grid\_rowconfigure(8, minsize=20) # empty row with minsize as spacing

**Analyzing “frame\_right” – 3x7 Grid System**

Visualizing frame\_right as a table

|  |  |  |  |
| --- | --- | --- | --- |
| **X** | **0** | **1** | **2** |
| **0** | frame\_info – 1x2 Grid System | | label\_radio\_group |
| **1** | radio\_button\_1 |
| **2** | radio\_button\_2 |
| **3** | radio\_button\_3 |
| **4** | slider\_1 | | switch\_1 |
| **5** | slider\_2 | | switch\_2 |
| **6** | checkbox\_1 | checkbox\_2 | combobox\_1 |
| **7** | NONE – Spacing with weigth = 10 | | |
| **8** | entry | | button\_5 |

Visualizing frame\_info as a table

|  |  |
| --- | --- |
| X | 0 |
| 0 | label\_info\_1 |
| 1 | progressbar |

**AppGui.py**

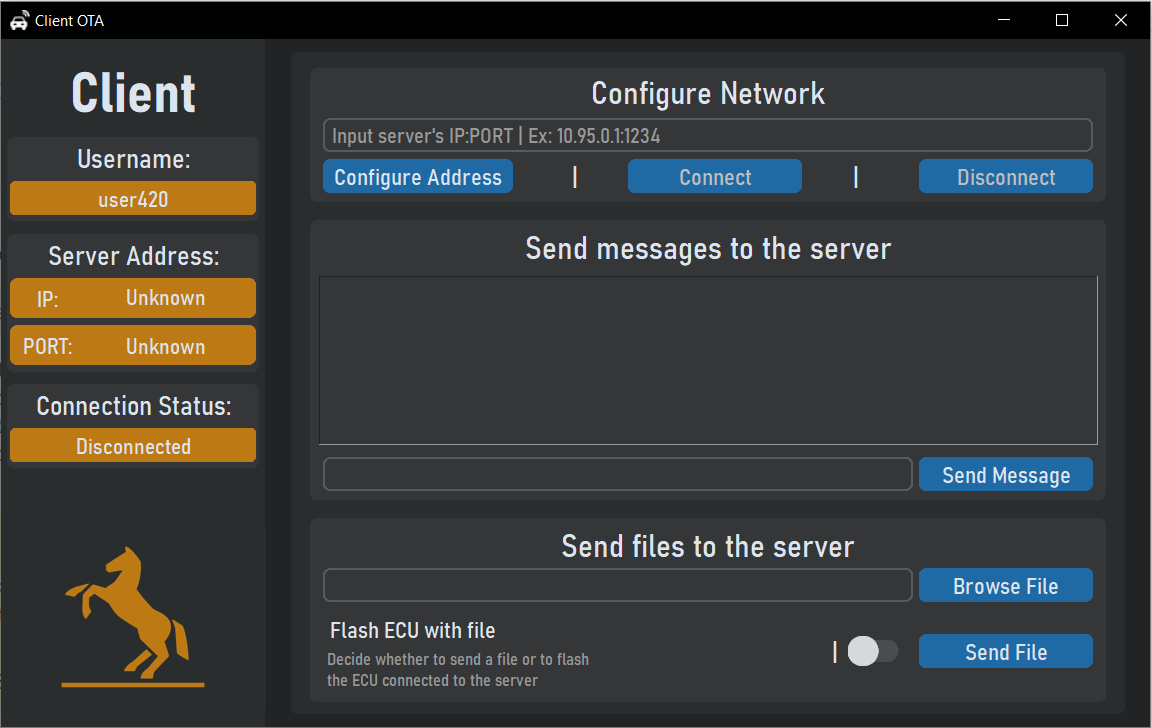
The GUI for our application and other related functionalities are implemented within AppGui.py module. The GUI is implemented with 2 big frames and some smaller insignificant ones. Those 2 frames are identified as “frame\_left” and “frame\_right”.

Main frame is structured as a 2x1 grid system.

Frame “frame\_left” is structured as a 1x7 grid system.

Frame “frame\_right” is structured as a 1x6 grid system.

**Block Analyzing Frame “frame\_left”.**

****

[**Row 7**](#frame_left_row7)

[**Row 6**](#frame_left_row6)

[**Row 5**](#frame_left_row5)

[**Row 4**](#frame_left_row4)

[**Row 3**](#frame_left_row3)

[**Row 2**](#frame_left_row2)

[**Row 0**](#frame_left_row0)

[**Row 1**](#frame_left_row1)

*Fig.5.6. Application GUI with Links to frame\_left rows*

**Row 0 – Spacing of 10px**

self.frame\_left.grid\_rowconfigure(0, minsize=10) # empty row with minsize as spacing

**Row 1 – label\_title\_1**

# --- Title Row 1 ---  
self.label\_title\_1 = customtkinter.CTkLabel(master=self.frame\_left,  
 text="Client",  
 text\_font=(self.LABEL\_FONT, -45, "bold"))  
# Obs: Fara padx, se strica marginea din dreapta  
self.label\_title\_1.grid(row=1, column=0, padx=5, pady=5)

The CTk element, “label\_title\_1” is going to be placed within frame\_left, will display the text “Client” and will have the font described within text\_font attribute. The element will be positioned on the second row in the frame. There will also be a distance of 5px added between the label and the frame coming from the padding attributes.

**Row 2 – frame\_username**

# --- frame\_username Row 2 ---  
self.frame\_username = customtkinter.CTkFrame(master=self.frame\_left,  
 height=50,  
 width=50)  
self.frame\_username.grid(row=2, column=0, sticky="ew", padx=5, pady=5)  
self.frame\_username.grid\_columnconfigure(0, weight=1)  
  
self.label\_username\_1 = customtkinter.CTkLabel(master=self.frame\_username,  
 text="Username:",  
 text\_font=(self.LABEL\_FONT, -22))  
self.label\_username\_1.grid(row=1, column=0, sticky="ew", padx=2, pady=2)

self.label\_username\_2 = customtkinter.CTkLabel(master=self.frame\_username,  
 text=self.DEFAULT\_INFO,  
 text\_font=(self.LABEL\_FONT, -18),  
 fg\_color=("white", self.WIDGET\_FGCOLOR),  
 corner\_radius=5)  
self.label\_username\_2.grid(row=2, column=0, sticky="ew", padx=2, pady=(2, 5))

Within row 2 of “frame\_left”, “frame\_username” is displayed, which is a 1x2 grid system. In frame\_username there are 2 labels. First one is used to display “Username”. The second is used to display the actual username.

Height and width attributes of “frame\_username” are used to set the default width and height. They once helped when first designing the GUI but now there have no role. The width and height attributes of a frame are overwritten when CTk elements are placed in the frame. The frame will expand accordingly to the size of the elements within it and will no longer take into consideration width and height attributes.

**Row 3 – frame\_address**

# --- frame\_address Row 3 ---  
self.frame\_address = customtkinter.CTkFrame(master=self.frame\_left)  
self.frame\_address.grid(row=3, column=0, sticky="new", padx=5, pady=5)  
self.frame\_address.grid\_columnconfigure(0, weight=1)  
  
self.label\_address\_1 = customtkinter.CTkLabel(master=self.frame\_address,  
 text="Server Address:",  
 text\_font=(self.LABEL\_FONT, -22))  
self.label\_address\_1.grid(row=0, column=0, columnspan=2, padx=2, pady=2)  
  
self.frame\_IP = customtkinter.CTkFrame(master=self.frame\_address,  
 fg\_color=("white", self.WIDGET\_FGCOLOR))  
self.frame\_IP.grid(row=1, column=0, padx=2, pady=2, sticky="we")  
  
self.label\_IP\_1 = customtkinter.CTkLabel(master=self.frame\_IP,  
 text="IP:",  
 width=50, # Change width from defaut  
 text\_font=(self.LABEL\_FONT, -18))  
self.label\_IP\_1.grid(row=0, column=0, padx=(5, 0), pady=2)  
  
self.label\_IP\_2 = customtkinter.CTkLabel(master=self.frame\_IP,  
 text=self.DEFAULT\_INFO,  
 text\_font=(self.LABEL\_FONT, -18))  
self.label\_IP\_2.grid(row=0, column=1, padx=(0, 2), pady=(0, 2))  
  
self.frame\_PORT = customtkinter.CTkFrame(master=self.frame\_address,  
 fg\_color=("white", self.WIDGET\_FGCOLOR))  
self.frame\_PORT.grid(row=2, column=0, padx=2, pady=(2, 5), sticky="we")  
self.label\_PORT\_1 = customtkinter.CTkLabel(master=self.frame\_PORT,  
 text="PORT:",  
 width=50,  
 text\_font=(self.LABEL\_FONT, -18))  
self.label\_PORT\_1.grid(row=0, column=0, padx=(5, 0), pady=2)  
self.label\_PORT\_2 = customtkinter.CTkLabel(master=self.frame\_PORT,  
 text=self.DEFAULT\_INFO,  
 text\_font=(self.LABEL\_FONT, -18))  
self.label\_PORT\_2.grid(row=0, column=1, padx=(0, 2), pady=2)

Row 3 of “frame\_left” is being occupied by another frame, “frame\_address” which implements a 1x3 grid system.

Inside first row of “frame\_address” is being placed “label\_address\_1” which contains information about the label “Server address”.

Inside second row of “frame\_address” there is another frame being placed, “frame\_IP” which implements a 2x1 grid. Inside “frame\_ip” there are 2 labels: “label\_IP\_1” and “label\_IP\_2” showing information about server’s IP.

The third row of “frame\_address” is like the second. There is another frame being place called “frame\_PORT” which implements a 2x1 grid. Inside “frame\_PORT” there are 2 labels: “frame\_PORT\_1” and “frame\_PORT\_2” showing information about server’s application PORT.

**Row 4 – frame\_constatus**

# --- frame\_constatus Row 4 ---  
self.frame\_constatus = customtkinter.CTkFrame(master=self.frame\_left)  
self.frame\_constatus.grid\_columnconfigure(0, weight=1)  
self.frame\_constatus.grid(row=4, column=0, sticky="new", padx=5, pady=5)  
self.label\_constatus1 = customtkinter.CTkLabel(master=self.frame\_constatus,  
 text="Connection Status:",  
 text\_font=(self.LABEL\_FONT, -22))  
self.label\_constatus1.grid(row=0, column=0, padx=2, pady=2)  
self.label\_constatus2 = customtkinter.CTkLabel(master=self.frame\_constatus,  
 text=self.DEFAULT\_CONNECTION\_INFO,  
 text\_font=(self.LABEL\_FONT, -18),  
 fg\_color=("white",self.WIDGET\_FGCOLOR),  
 corner\_radius=5)  
self.label\_constatus2.grid(row=1, column=0, sticky="ew", padx=2, pady=(2, 5))

Inside Row 4, “frame\_constatus” can be found. “frame\_constatus” is a 1x2 grid system which contains label with information about the connection status of the server.

**Row 5 – Spacing – Fill remaining space**

self.frame\_left.grid\_rowconfigure(5, weight=1)

**Row 6 – Image**

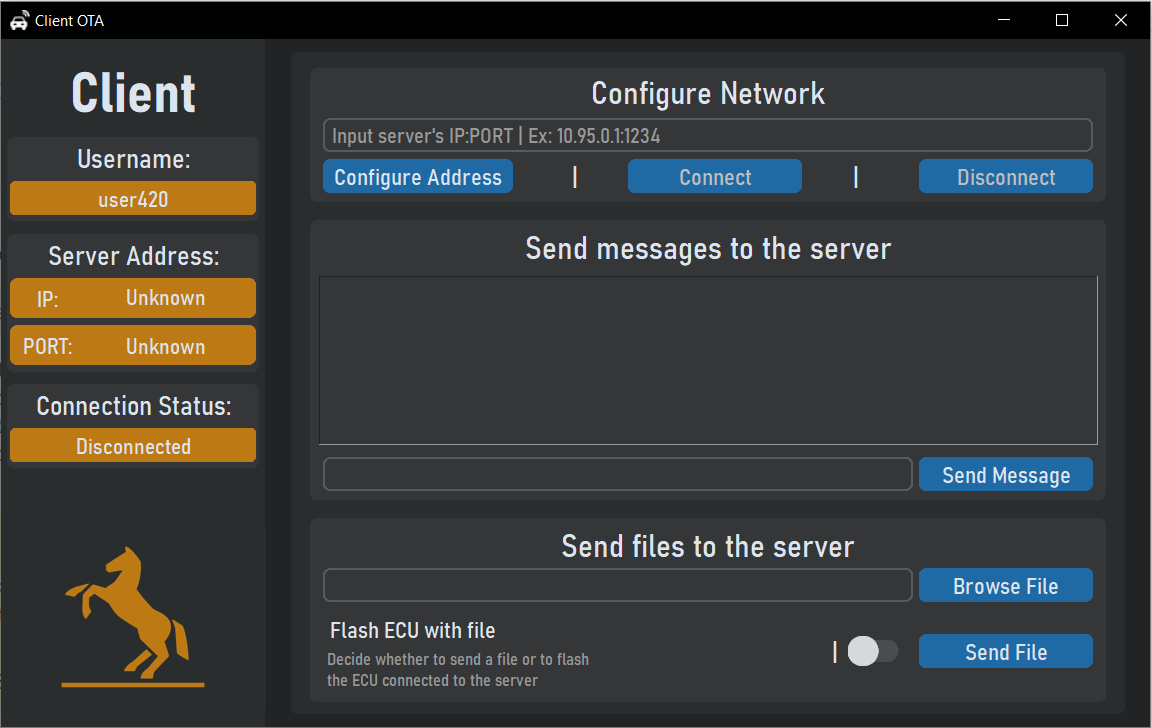
# --- Image label Row 6 ---  
  
self.img = Image.open("GUImages/conti.png").resize((190, 180))  
self.image\_wifi = ImageTk.PhotoImage(self.img)  
  
self.labelimg = customtkinter.CTkLabel(master=self.frame\_left,  
 image=self.image\_wifi)  
self.labelimg.grid(row=5, column=0, sticky="sew")

On row 6 an image is placed, the logo of the GUI.

**Row7 – Spacing of 20px**

self.frame\_left.grid\_rowconfigure(7, minsize=20)

**Block Analyzing Frame “frame\_right”.**

****

[**Row 0**](#frame_right_row0)

[**Row 6**](#frame_right_row6)

[**Row 5**](#frame_right_row5)

[**Row 4**](#frame_right_row4)

[**Row 3**](#frame_right_row3)

[**Row 2**](#frame_right_row2)

[**Row 1**](#frame_right_row1)

*Fig.5.7. Application GUI with links to frame\_right rows*

**Row 0 – Spacing of 10px**

self.frame\_right.rowconfigure(0, minsize=10)

**Row 1 – frame\_set\_address**

# --- frame\_set\_address Row 1 ---  
self.frame\_configure\_address = customtkinter.CTkFrame(master=self.frame\_right,  
 height=50)  
self.frame\_configure\_address.grid(row=1, column=0, padx=15, pady=5, sticky="nsew")  
self.frame\_configure\_address.columnconfigure((0, 1, 2, 3, 4), weight=1)  
self.frame\_configure\_address.rowconfigure(1, weight=1)  
self.label\_configure\_address = customtkinter.CTkLabel(master=self.frame\_configure\_address,  
 text="Configure Network",  
 text\_font=(self.LABEL\_FONT, -26))  
self.label\_configure\_address.grid(row=0, column=0, columnspan=5, padx=2, pady=2, sticky="new")  
  
self.entry\_configure\_address = customtkinter.CTkEntry(master=self.frame\_configure\_address,  
 text\_font=(self.ENTRY\_FONT, -17),  
 text\_color="gray60")  
self.entry\_configure\_address.grid(row=1, column=0, columnspan=5, padx=10, pady=2, sticky="ew")  
self.entry\_configure\_address.insert(0, "Input server's IP:PORT | Ex: 10.95.0.1:1234")  
self.entry\_configure\_address.bind("<1>", self.click\_entry\_netconfig)  
  
self.button\_configure\_address = customtkinter.CTkButton(master=self.frame\_configure\_address,  
 text="Configure Address",  
 text\_font=(self.BUTTON\_FONT, -19))  
self.button\_configure\_address.grid(row=2, column=0, padx=(10, 2), pady=(2, 7), sticky="sw")  
self.label\_configure\_address\_button\_separator = customtkinter.CTkLabel(master=self.frame\_configure\_address,  
 text="|",  
 text\_font=(self.LABEL\_FONT, -20),  
 width=50)  
self.label\_configure\_address\_button\_separator.grid(row=2, column=1, pady=(2, 7), sticky="sew")  
self.button\_connect = customtkinter.CTkButton(master=self.frame\_configure\_address,  
 text="Connect",  
 text\_font=(self.BUTTON\_FONT, -19),  
 state="disabled")  
self.button\_connect.grid(row=2, column=2, padx=2, pady=(2, 7), sticky="s")  
  
self.label\_configure\_address\_button\_separator2 = customtkinter.CTkLabel(master=self.frame\_configure\_address,  
 text="|",  
 text\_font=(self.LABEL\_FONT, -20),  
 width=50)  
self.label\_configure\_address\_button\_separator2.grid(row=2, column=3, pady=(2, 7), sticky="sew")  
  
self.button\_disconnect = customtkinter.CTkButton(master=self.frame\_configure\_address,  
 text="Disconnect",  
 text\_font=(self.BUTTON\_FONT, -19),  
 state="disabled")  
self.button\_disconnect.grid(row=2, column=4, padx=(2, 10), pady=(2, 7), sticky="se")

Row 1 of frame\_right contains a frame identified as “frame\_set\_address” which implements a 5x2 grid system.

self.entry\_configure\_address.grid(row=1, column=0, columnspan=5, padx=10, pady=2, sticky="ew")  
self.entry\_configure\_address.insert(0, "Input server's IP:PORT | Ex: 10.95.0.1:1234")  
self.entry\_configure\_address.bind("<1>", self.click\_entry\_netconfig)  
  
self.button\_configure\_address = customtkinter.CTkButton(master=self.frame\_configure\_address,  
 text="Configure Address",  
 text\_font=(self.BUTTON\_FONT, -19))  
self.button\_configure\_address.grid(row=2, column=0, padx=(10, 2), pady=(2, 7), sticky="sw")  
self.label\_configure\_address\_button\_separator = customtkinter.CTkLabel(master=self.frame\_configure\_address,  
 text="|",  
 text\_font=(self.LABEL\_FONT, -20),  
 width=50)  
self.label\_configure\_address\_button\_separator.grid(row=2, column=1, pady=(2, 7), sticky="sew")  
self.button\_connect = customtkinter.CTkButton(master=self.frame\_configure\_address,  
 text="Connect",  
 text\_font=(self.BUTTON\_FONT, -19),  
 state="disabled")  
self.button\_connect.grid(row=2, column=2, padx=2, pady=(2, 7), sticky="s")  
  
self.label\_configure\_address\_button\_separator2 = customtkinter.CTkLabel(master=self.frame\_configure\_address,  
 text="|",  
 text\_font=(self.LABEL\_FONT, -20),  
 width=50)  
self.label\_configure\_address\_button\_separator2.grid(row=2, column=3, pady=(2, 7), sticky="sew")  
  
self.button\_disconnect = customtkinter.CTkButton(master=self.frame\_configure\_address,  
 text="Disconnect",  
 text\_font=(self.BUTTON\_FONT, -19),  
 state="disabled")  
self.button\_disconnect.grid(row=2, column=4, padx=(2, 10), pady=(2, 7), sticky="se")

Frame “frame\_set\_address”:

* In row 0 of frame\_set\_address on a colspan of 5, there is “label\_configure\_address” with informative role.
* In row 1 of frame\_set\_address on a colspan of 5, there is “entry\_configure\_address”. This entry’s purpose is to get the server’s address from the user.
* In row 2 of frame\_set\_address are located 3 buttons and 2 separators. The buttons are: button\_configure\_address on column 0, button\_connect on column 2 and button\_disconnect on column 4. On columns 1 and 3 there are separators for the buttons. The separators are just two labels with the text “|”.

**Row 2 – Spacing of 5px**

self.frame\_right.rowconfigure(2, minsize=5)

v

**Row 3 – frame\_messagebox**

# --- frame\_messagebox Row 3 ---  
self.frame\_messagebox = customtkinter.CTkFrame(master=self.frame\_right)  
self.frame\_messagebox.grid(row=3, column=0, padx=15, pady=5, sticky="nsew")  
self.frame\_messagebox.grid\_columnconfigure(0, weight=1)  
self.frame\_messagebox.grid\_rowconfigure(1, weight=1)  
self.label\_send = customtkinter.CTkLabel(master=self.frame\_messagebox,  
 text="Send messages to the server",  
 text\_font=(self.LABEL\_FONT, -26))  
self.label\_send.grid(row=0, column=0, columnspan=2, padx=2, pady=(5, 0))  
  
self.textbox = customtkinter.CTkTextbox(master=self.frame\_messagebox,  
 text\_font=(self.TEXTBOX\_FONT, -20),  
 state='disabled',  
 height=100)  
self.textbox.grid(row=1, column=0, columnspan=2, padx=2, pady=2, sticky="nsew")  
  
self.entry\_send\_msg = customtkinter.CTkEntry(master=self.frame\_messagebox,  
 text\_font=(self.ENTRY\_FONT, -17))  
self.entry\_send\_msg.grid(row=2, column=0, padx=(10, 2), pady=(2, 7), sticky="ew")  
  
self.button\_send\_message = customtkinter.CTkButton(master=self.frame\_messagebox,  
 text="Send Message",  
 text\_font=(self.BUTTON\_FONT, -19))  
self.button\_send\_message.grid(row=2, column=1, padx=(2, 10), pady=(2, 7))

Within row 3 of frame\_right there is another frame: “frame\_messagebox”which implements a 2x3 grid.

Frame “frame\_messagebox”:

* Row 0 of frame\_messagebox contains a label with info about frame.
* Row 1 of frame\_messagebox contains a textbox in which messages sent/received to/from the server and files send to the server are displayed.
* Row 2 | Column 0 of frame\_messagebox contains an entry(“entry\_send\_message”) in which the messages to be sent are written.
* Row 2 | Column 1 of frame\_messagebox contains a button whose function is to get the text from “entry\_send\_message” and send it to the server.

**Row 4 – Spacing of 5px**

self.frame\_right.rowconfigure(4, minsize=5)

v

**Row 5 – frame\_files**

# --- frame\_files Row 5---  
self.frame\_files = customtkinter.CTkFrame(master=self.frame\_right)  
self.frame\_files.grid(row=5, column=0, padx=15, pady=5, sticky="nsew")  
self.frame\_files.grid\_columnconfigure(0, weight=1)  
self.label\_files = customtkinter.CTkLabel(master=self.frame\_files,  
 text="Send files to the server",  
 text\_font=(self.LABEL\_FONT, -26))  
self.label\_files.grid(row=0, column=0, columnspan=4, padx=2, pady=(5, 0))  
  
self.entry\_browse\_file = customtkinter.CTkEntry(master=self.frame\_files,  
 text\_font=(self.ENTRY\_FONT, -17))  
self.entry\_browse\_file.grid(row=1, column=0, columnspan=3, padx=(10, 2), pady=(2, 2), sticky="ew")  
self.button\_browse\_file = customtkinter.CTkButton(master=self.frame\_files,  
 text="Browse File",  
 text\_font=(self.BUTTON\_FONT, -19),  
 command=self.browse\_file)  
self.button\_browse\_file.grid(row=1, column=3, columnspan=1, padx=(2, 10), pady=(2, 2))  
  
self.frame\_label\_file\_purpose = customtkinter.CTkFrame(master=self.frame\_files,  
 fg\_color=("white", "#343638"))  
self.frame\_label\_file\_purpose.grid(row=2, column=0, padx=(10, 2), pady=(2, 7), sticky="w")  
self.label\_file\_purpose = customtkinter.CTkLabel(master=self.frame\_label\_file\_purpose,  
 text="Flash ECU with file",  
 text\_font=(self.LABEL\_FONT, -19))  
self.label\_file\_purpose.grid(row=0, column=0, padx=2, pady=(2, 0), stick="w")  
self.label\_file\_purpose\_description = customtkinter.CTkLabel(master=self.frame\_label\_file\_purpose,  
 text="Decide whether to send a file or to flash\n"  
 "the ECU connected to the server",  
 text\_font=(self.LABEL\_FONT, -14),  
 text\_color="gray60",  
 width=50,  
 justify=tkinter.LEFT)  
self.label\_file\_purpose\_description.grid(row=1, column=0, padx=2, pady=(0, 2), sticky="nw")  
  
self.label\_file\_purpose\_separator = customtkinter.CTkLabel(master=self.frame\_files,  
 text="|",  
 text\_font=(self.LABEL\_FONT, -20),  
 width=15)  
self.label\_file\_purpose\_separator.grid(row=2, column=1, pady=(2, 7))  
  
self.switch\_file\_purpose = customtkinter.CTkSwitch(master=self.frame\_files,  
 text="",  
 height=24,  
 width=44)  
self.switch\_file\_purpose.grid(row=2, column=2, padx=2, pady=(2, 7))  
  
self.button\_send\_file = customtkinter.CTkButton(master=self.frame\_files,  
 text="Send File",  
 text\_font=(self.BUTTON\_FONT, -19))  
self.button\_send\_file.grid(row=2, column=3, padx=(2, 10), pady=(2, 7))

v

Inside row 5 of “frame\_right” there is placed another frame identified as “frame\_files” which implements a 4x3 grid system.

self.button\_send\_file = customtkinter.CTkButton(master=self.frame\_files,  
 text="Send File",  
 text\_font=(self.BUTTON\_FONT, -19))  
self.button\_send\_file.grid(row=2, column=3, padx=(2, 10), pady=(2, 7)

Frame “frame\_files”:

* Row 0: A label with information about the frame
* Row 1 | Columns 0, 1, 2: An entry (“entry\_browse\_file”) whose purpose is to store the address of the file we want to send to the server.
* Row 1 | Column 3: A button(“button\_browse\_file”) whose purpose is to open the file browser.
* Row 2 | Column 0: Frame “frame\_label\_file\_purpose” which implements a 1x2 grid system.

Frame “frame\_label\_file\_purpose”:

* Row 0: Label “label\_file\_purpose” which contains “Flash ECU with file” text.
* Row 1: Label “label\_file\_purpose\_descriptions” which contains a description about what the option of flashing ECU does.
* Row 2 | Column 1: Label “label\_file\_purpose\_seaprator” is just a label with the text “|”.
* Row 2 | Column 2: A switch (“switch\_file\_purpose”) which purpose is to let the user decide whether he wants to flash the selected file or not. By default it doesn’t flash.
* Row 2 | Column 3: Button “button\_send\_file” which function is to send the selected file to the server.

**Row 6 – Spacing of 5px**

self.frame\_right.rowconfigure(6, minsize=5)

v

**Functionality of the GUI:**

Most of the functions which allow the GUI to work are being implemented in client.py because of the necessity of references from the script. Also client.py is the script that launches the GUI.

The functionality of the buttons is provided by 6 functions, 5 of them from client.py and 1 of them from AppGui.py itself.

Button functions from client.py:

* **gui\_config\_address()** reads the string from “entry\_configure\_address”, checks wether the address is a valid one and updates the corresponding labels.
* **gui\_connect()** calls the connect function.
* **gui\_disconnect()** calls the disconnect function.
* **gui\_send\_message()** reads the string from “entry\_send\_message” and calls the function that allows to send that message
* **gui\_send\_file()** reads the status of the switch “switch\_file\_purpose”, reads the path of the file and then calls the function that allows to send that message. When sending the file it can also raise the flag “flash\_ecu” telling the server that the file sent must be flashed to ECU.

Button functions from AppGui.py:

* **browse\_file()** opens filedialog which allows client to browse a file and store the path into “entry\_browse\_file”

**Animations of the GUI:**

The GUI for our application uses animations for feedback whenever a client connects, disconnects or is trying to connect.

At the roots of most tkinter animations there is a method called .after() which must be analyzed in depth. When an animation is running in tkinter we want the GUI to still work. If using sleep() when animating, the GUI will stop working because it can not longer execute the code that keeps it alive.

Method .after() helps us get past this inconvenience. It takes a time interval in milliseconds and a function to execute. When calling the .after(), the program will wait the time interval and then call the function given as parameter without blocking the flow of programs. Instead of calling sleep and blocking the program for a number of milliseconds we instead call after and let the program flow while still timing the time interval.

**Animating the event of trying to connect:**

The method which implements the animation for trying to connect is “**is\_connecting\_animation**”.

if self.label\_constatus2.text.rstrip('.') == "Connecting":  
 self.label\_constatus2.configure(text=f"Connecting{'.'\*self.ANIMATION\_COUNTER2}")  
 self.ANIMATION\_COUNTER2 += 1  
 if self.ANIMATION\_COUNTER2 >= 4:  
 self.ANIMATION\_COUNTER2 = 1  
 self.ANIMATION\_ID2 = self.after(500, self.is\_connecting\_animation)

First condition enables us to detect whenever the necessary conditions for client trying to connect have been met. Whenever client tries to connect, “label\_constatus2” changes its content to “Connecting” from within “gui\_connect()” function. This is how we know that client tries to connect to a server.

If client attempted to connect, for every 500ms add “.” in the end of the string “Connecting” from “label\_constatus2”. “ANIMATION\_COUNTER2” tells the number of dots to be added (max of 3).

**Animating the events of connect and disconnect:**

The methods which implement the animations for successfully connecting and disconnecting are “**connect\_animation**” and “**disconnect\_animation**”.

Both methods have another method at their roots which is “**animation\_textcolor\_pulse**”. This method implements a linear gradient between 2 colors for a text inside a CTk element.

Let’s take as an example the simple case of transitioning from black to white. will represent black and will represent white.

Our method takes in the number of steps we want to take in order to get from to . If is the current step and is the total number of steps then the current color of our textbox will be represented by the formula.

In the body of the function, is “ANIMATION\_COUNTER1” and is “c”. The function also implements the returning to original color so for n represent both the steps to and the steps back to .

**6.** **Checking ECU Flash using SPI (Serial Peripheral Interface)**

After ECU has been flashed thanks to AVRDude programming tool, the Raspberry Pi will try to establish an SPI communication with the ATMEGA328. Raspberry will be the master and ATMEGA328 will be the slave.

**6.1. Bit-Banging SPI on Raspberry Pi**

There are two ways of implementing SPI protocol. Either use the hardware interface found by default on the Raspberry Pi or bit-bang SPI using a software implementation. For this project the latter has been chosen because of lack of wires. The avrdude programming tool has been implemented before SPI thus pins were already reserved for avrdude and could have been reused only if bit-banging SPI.

**RPiSPi.py module**

The main doc on which this module is based on is the following and will be further referred as microchip doc.

<https://ww1.microchip.com/downloads/en/Appnotes/Atmel-2585-Setup-and-Use-of-the-SPI_ApplicationNote_AVR151.pdf>

Functions:

* **get\_bit(u8, bit)**

Params:

* u8 – An unsigned 8 bit variable
* bit – The rank of the bit you want to read.

Returns: The state of the bit given as parameter. (1 or 0)

* **get\_bit\_gpio(u8,bit)** – Has a dependency on “get\_bit” function

Params:

* u8 – An unsigned 8 bit variable
* bit – The rank of the bit you want to read.

Returns: Depending on the state of the bit given as parameter it an return either “GPIO.HIGH” or “GPIO.LOW”

* **print\_bites(u8)**
* u8 – An unsigned 8 bit variable

Prints bites of a unsigned 8 bit variable

**Class “SPIMaster”**:

When instancing class, programmer gives as parameters pin numbers for SCK, MISO, MOSI and SS. Through bit-banging every pin can be used to communicate through SPI.

Within the class, initialization is represented by setting the direction and the state of the pins.

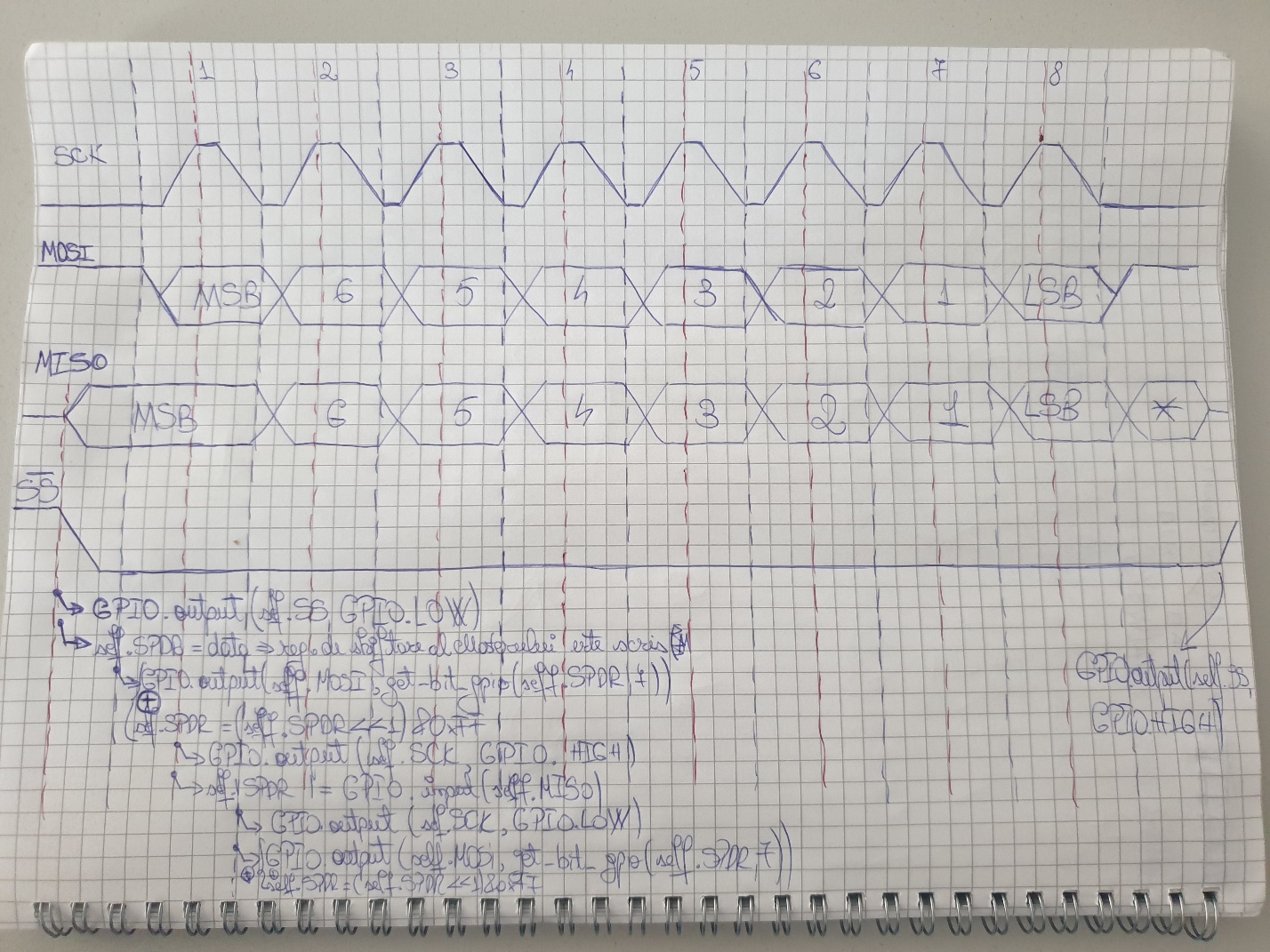
# Init Pins Purpose  
GPIO.setmode(GPIO.BCM)  
GPIO.setwarnings(False)  
GPIO.setup(self.SCK, GPIO.OUT) # SCK output  
GPIO.setup(self.MISO, GPIO.IN) # MISO input  
GPIO.setup(self.MOSI, GPIO.OUT) # MOSI output  
GPIO.setup(self.SS, GPIO.OUT) # SS output  
# Init Pins State  
GPIO.output(self.SCK, GPIO.LOW) # SCK is active HIGH  
GPIO.output(self.MOSI, GPIO.HIGH)  
GPIO.output(self.SS, GPIO.HIGH) # SS is active LOW

After initialization pins should have the following directions and states:

* SCK – OUTPUT – LOW
* MISO – INPUT
* MOSI – OUTPUT – HIGH
* SS – OUTPUT – HIGH

In the microchip documentation, section 1.4. there are 4 SPI modes of operation. Within the class only one of them is implemented thus there is no CPOL and CPHA bits. For the communication to work, the hardware SPI on ATMEGA must have CPOL and CPHA both on 0. CPOL = 0 means the clock will be active on the rising edge. CPHA = 0 means that the sampling of MISO(in case of master) and the sampling of MOSI(in case of slave) will be made on the active edge(rising edge) and the shifting of bits will take place on the other edge (falling edge).

Method **write\_nd\_receive**: Its purpose is to generate signals in agreement to the SPI Protocol as master.



*Fig.6.1 SPI Waveforms and Raspberry Pi instructions.*

A 8 bit transmission is made out of 3 stages:

* Initialization. Within this part, master have to put & hold SS line on 0.

GPIO.output(self.SS, GPIO.LOW)

Now the slave figured it out that the master intents to start a communication and prepares itself to communicate.

To fully abide SPI protocol we will save our data (which is an unsigned 8bit register) inside SPDR register (SPI Data Register) which will shift on the falling edge.

GPIO.output(self.MOSI, get\_bit\_gpio(self.SPDR, 7))

self.SPDR = data

Master then will write MOSI pin for it to be read by slave.

Clear last bit of SPDR so it will be ready to receive MOSI.

self.SPDR = (self.SPDR << 1) & 0xFF

* Loop. In the loop, the master has 2 tasks:
  + Rising Edge: Set SCK.

GPIO.output(self.SCK, GPIO.HIGH)

Read MISO

GPIO.output(self.SCK, GPIO.HIGH)

* + Falling Edge: Clear SCK

GPIO.output(self.MOSI, get\_bit\_gpio(self.SPDR, 7))

GPIO.output(self.SCK, GPIO.LOW)

Write MOSI

* End. Set SS (SS is Active Low)

GPIO.output(self.SS, GPIO.HIGH)

When receiving or sending a word, the process explained above will be repeated multiple times.

Method **send\_msg**: Send a string as a group of bytes through SPI. To be able to send a message through SPI, the slave should be informed both about the start of the string and the end of the string. As a consequence at the start of sending a string we first send the integer “1” = 0x01 and at the end of the string we send the integer “0” = 0x00.

# Send a message  
def send\_msg(self, msg):  
 self.write\_nd\_receive(1)  
 # print("---------------SEND\_MSG--------------- Sent 1, next is first char in str.")  
 for i in range(len(msg)):  
 sleep(self.TIME\_BETWEEN\_CHAR\_SENT)  
 self.write\_nd\_receive(ord(msg[i]))  
 # print(hex(self.SPDR))  
 sleep(self.TIME\_BETWEEN\_CHAR\_SENT)  
 self.write\_nd\_receive(0)  
 # print(hex(self.SPDR))  
 # print("---------------END\_SEND\_MSG--------------- Sent 0.")

Method **recv\_msg**: Receive a string through SPI. The challenging part when receiving a message is the synchronization required between master and slave. When master sends a message and slave receives it, master is the one to start the communication by clearing SS pin. When slave wants to send a message it has no way of telling Master it wants to send a message. The master is the one that needs to start a communication from time to time to check if slave has anything to send. In our case, slave has been programmed so that it will send a message right after it receives one thus master must receive right after he sends a message. Alike sending a whole string, receiving a string is signaled by receiving the integer “1” followed by the integer “0”.

def recv\_msg(self):  
 cuv = ""  
 sleep(self.TIME\_BETWEEN\_CHAR\_RECV)  
 self.write\_nd\_receive(0xFF)  
 # print(f"---------------START\_RECV\_MSG--------------- RECEIVED: {self.SPDR}")  
 if self.SPDR == 1:  
 while self.SPDR != 0:  
 sleep(self.TIME\_BETWEEN\_CHAR\_RECV)  
 self.write\_nd\_receive(0xFF)  
 if self.SPDR != 0:  
 cuv += chr(self.SPDR)  
 elif self.SPDR != 1:  
 return ""  
 return cuv

**6.2. Hardware SPI on ATMEGA328.**

For a successful communication between RPi and ATMEGA328, both must be configured to work with SPI. In this sense, a program for ATMEGA has been developed within Arduino IDE. After installing Arduino IDE there is a configuration to be made so it can compile code for ATMEGA.

From <https://www.brennantymrak.com/articles/programming-avr-with-arduino> , those steps must be followed:

* Download the [breadboard-1-6-x configuration here.](https://www.arduino.cc/en/uploads/Tutorial/breadboard-1-6-x.zip)
* Determine the location of your Arduino sketchbook folder by opening the Arduino IDE, going to File, selecting Preferences and then looking at the Sketchbook location.
* Go to the location of your sketchbook folder and create a "hardware" folder if one does not exist already.
* Extract the "breadboard" folder from the "breadboard-1-6-x" zip file and move it into the newly created "hardware" folder. The "hardware" folder should now have the "breadboard" folder in it.
* Restart the Arduino IDE
* In the Arduino IDE, confirm that setup was completed correctly by going to Tools > Board. You should now see "ATmega328 on a breadboard (8 MHz internal clock)" as an option in the boards list.

And then:

* Go to the "hardware" folder we just created when loading the configuration and open the "breadboard" folder, then open the "avr" folder.
* Open the "boards.txt" file
* In the file, find the line that says "atmega328bb.build.f\_cpu=8000000L"
* Change the line to the following "atmega328bb.build.f\_cpu=1000000L"
* Change every instance of “atmega328p” with “atmega328” or any other avr microcontroller you’re using.
* Save and close the file

Now Arduino IDE is ready to compile code. The code for our SPI application can be found inside the documentation folder and it’s named “ATMEGA328\_SPIApp”.

**ATMEGA328 SPI Code**

* “**void SPI\_Init\_Slave()**”. Initialize uC, get it ready for SPI communication.
* MOSI, SCK and SS are inputs

DDRB |= (1<<MISO); /\* Make MISO pin as output pin \*/

DDRB &= ~((1<<MOSI)|(1<<SCK)|(1<<SS)); /\* Make MOSI, SCK, SS as input pins \*/

* MISO is output
* Clear SPIF (SPI Interrupt Flag) to prepare for interrupt

SPSR &= !(1<<SPIF); /\*Prevent false interrupt at start. If SPIE is set and SPIF is set at start => False interrupt

SPIF = 0 resolves issue.\*/

* Enable global interrupts

sei(); //Enable Global Interrupts

* Set SPIE(SPI Interrupt Enable) and SPE(SPI Enable)

SPCR = (1<<SPIE) | (1<<SPE);

**Line by line route of receiving a message:**

The microcontroller will use the hardware implemented SPI to receive bytes. When a byte is successfully received (SS goes from 0 to 1) SPIF is set automatically and uC’s processing unit will now focus on “**ISR(SPI\_STC\_vect)**”.

Inside ISR (Interrupt Service Routine) the following will happen:

* Save SPDR(SPI Data Register) in which the byte received from Master is stored.

spi\_rcv = SPDR;

* If spi\_rcv is 0x01 then slave knows the following bytes will represent ASCII characters from a string. In our case the condition is accomplished. uC sets SPI\_RECV\_S flag. Setting this flag determine that further bytes to be stored inside a buffer that will represent our received string.

if(spi\_rcv == 1)

{

SET\_BIT(spi\_st, SPI\_RECV\_S);

}

* Further into the ISR, SPI\_RECV\_S is 1 so below condition will be accomplished but because spi\_rcv is 1, there will be no character saved inside the buffer.

if(READ\_BIT(spi\_st, SPI\_RECV\_S))

{

if(spi\_rcv != 1)

{

buff\_addval(&SPI\_recv\_buffer, spi\_rcv);

}

}

* Program leaves ISR.
* uC receives another byte through SPI and enters ISR again. This time above SPI\_RECV\_S is “1” and spi\_rcv is different from “1” thus inside a buffer (vector of char variables), first character will be stored.
* Program leaves ISR then enters it again whenever it receives a char. This process will loop until receiving “0”.
* Slave receives “0”. Due to receiving “0”, the following will happen:
  + The flag indicating a string is being received, SPI\_RECV\_S, is cleared
  + Null character (‘\0’) is stored in the buffer, indicating end of string.
  + The position of the next character to be written in the buffer is reset (put on 0). This indicates that buffer is ready to be read and rewritten.
  + The flag indicating a string has been received successfully, SPI\_RECV\_SD, is set.

else if(spi\_rcv == 0)

{

CLEAR\_BIT(spi\_st, SPI\_RECV\_S);

buff\_addval(&SPI\_recv\_buffer, '\0');

SPI\_recv\_buffer.pos = 0;

SET\_BIT(spi\_st, SPI\_RECV\_SD);

}

* uC exits ISR. Inside the infinite loop from main, the condition targeting SPI\_RECV\_SD will now be accomplished.

SPI\_RECV\_SD will get cleared which will indicate that the condition has been treated.

Further, the string received will get compared with “FLASH”. If Raspberry Pi sent the string “FLASH” it means it wants a response from slave that he got successfully flashed. Taking this into consideration, we will prepare to send a string from server to slave by writing SDPR with 0x01 telling Master you also want to send a string. The buffer in which the string will be sent from is called “SPI\_send\_buffer” and the position of this buffer indicates the rank of the character that must be sent. The condition ends by setting “SPI\_SEND\_AS” flag.

**Line by line route of sending a string:**

while(1)

{

\_delay\_us(500);

if(READ\_BIT(spi\_st, SPI\_IF))

{

if(READ\_BIT(spi\_st, SPI\_RECV\_SD))

{

CLEAR\_BIT(spi\_st, SPI\_RECV\_SD);

if(str\_compare(SPI\_recv\_buffer.cells, "FLASH"))

{

SPDR = 0x01;

str\_copy(SPI\_send\_buffer.cells, "ACK\_FLASH");

SPI\_send\_buffer.pos = 0;

SET\_BIT(spi\_st, SPI\_SEND\_AS);

}

else

{

SPDR = 0x02;

}

}

}

} // -> while(1)

In our program to send a string “SPI\_RECV\_AS” must be set, as well as SPDR must be written to 0x01.The string that we want to send will be stored at “SPI\_send\_buffer.cells”. If those happen then after the next SPI interrupt the following will occur:

* When getting into **ISR(SPI\_STC\_vect)**, the condition implying SPI\_SEND\_AS will be accomplished. Then if the current character is different from null character (‘\0’) it will be assigned to SPDR (current character is the character from “SPI\_send\_buffer.cells” with the rank given by “SPI\_send\_buffer.pos”).Increment “SPI\_send\_buffer.pos” because you’ve prepared to send current char.s
* Program exits ISR. Then reenters and prepares again to send a character out of our buffer.

if(READ\_BIT(spi\_st, SPI\_SEND\_AS))

{

if(SPI\_send\_buffer.cells[SPI\_send\_buffer.pos] != '\0')

{

SPDR = SPI\_send\_buffer.cells[SPI\_send\_buffer.pos];

SPI\_send\_buffer.pos++;

}

else

{

SPDR = 0x00;

CLEAR\_BIT(spi\_st, SPI\_SEND\_AS);

}

}

SET\_BIT(spi\_st, SPI\_IF);

* After all the characters have been send and the current character to send is ‘\0’, set SPDR on 0x00 telling Master whole string has been sent and that he can stop generating SCK. Clear flag “SPI\_SEND\_AS” in order to signal uC that there is no message to send through SPI.

else

{

SPDR = 0x00;

CLEAR\_BIT(spi\_st, SPI\_SEND\_AS);

}

**7.****Bibliography**

**The following links lead to sources of research and inspiration that helped the project take its current form:**

* <https://www.youtube.com/watch?v=3QiPPX-KeSc>
* [(13) Sockets with Python 3 - YouTube](https://www.youtube.com/playlist?list=PLQVvvaa0QuDdzLB_0JSTTcl8E8jsJLhR5)
* [App structure and layout · TomSchimansky/CustomTkinter Wiki · GitHub](https://github.com/TomSchimansky/CustomTkinter/wiki/App-structure-and-layout)
* [CustomTkinter/complex\_example.py at master · TomSchimansky/CustomTkinter · GitHub](https://github.com/TomSchimansky/CustomTkinter/blob/master/examples/complex_example.py)
* [AVR151: Setup And Use of The SPI (microchip.com)](https://ww1.microchip.com/downloads/en/Appnotes/Atmel-2585-Setup-and-Use-of-the-SPI_ApplicationNote_AVR151.pdf)
* [Programming an AVR ATmega328P with an Arduino (brennantymrak.com)](https://www.brennantymrak.com/articles/programming-avr-with-arduino)
* [Python Programming Tutorials](https://pythonprogramming.net/server-chatroom-sockets-tutorial-python-3/?completed=/pickle-objects-sockets-tutorial-python-3/)
* [How to Transfer Files in the Network using Sockets in Python - Python Code (thepythoncode.com)](https://www.thepythoncode.com/article/send-receive-files-using-sockets-python)
* [Python Socket Receive Large Amount of Data - Stack Overflow](https://stackoverflow.com/questions/17667903/python-socket-receive-large-amount-of-data)
* [Overview | Program an AVR or Arduino Using Raspberry Pi GPIO | Adafruit Learning System](https://learn.adafruit.com/program-an-avr-or-arduino-using-raspberry-pi-gpio-pins)
* [Avr Atmega Atmega1632 Spi | Avr Atmega (electronicwings.com)](https://www.electronicwings.com/avr-atmega/atmega1632-spi)
* [Threading in Python: The Complete Guide (superfastpython.com)](https://superfastpython.com/threading-in-python/)