# Bluetooth.h (ESP32\_VROOM)

This file will handle the BLE Bluetooth module for the timer.

By default, the Bluetooth modules are in “slave” mode. This means that it’s an open connection accepting other devices to connect to it. However, in this case we actually want to connect to another device. We can achieve this by setting the module into “master” mode.

This can be done by sending command directly to the module using Serial communication (RX + TX). Normally you can simply prepare the device and set it in the right settings, but I found that in some situations the module out of nowhere is back in “slave” mode if it loses connection. Due to that there is logic that will send the commands to the BLE module to set it in the correct mode.

Establishing a connection with another BLE device is also done by sending commands to the BLE module. For a complete list of all possible commands (and an in-depth documentation of this specific module) have a look here:

<http://www.martyncurrey.com/hm-10-bluetooth-4ble-modules/#HM-10%20-%20AT%20commands>

A picture containing text

Description automatically generated

We first define the SirSA\_BT\_ADDRESS, this is the MAC address of the device we will connect to (the BLE bluetooth module on the robot). The MAC address is unique to the device and thus make sure we connect to the right device.

We also define the BT\_STATE\_PIN, which is attached to PIN 4. This PIN will be used to determine if the BLE bluetooth device is currently connected.

Graphical user interface, text, application

Description automatically generated

The “isConnected()” function will return a boolean (true / false) based on if there currently is another BLE devices connected. We do this by using the “digitalRead()” function and passing the BT\_STATE\_PIN (PIN 4) to read the current power state. When the PIN output from the BT\_STATE\_PIN is HIGH, we are connected.

We set this value to the “bluetoothIsConnected” variable, which is defined in “ESP32\_VROOM.ino” so we can determine in other parts of the code if we are currently connected or not. We also return the value so we can trigger this function directly in this file and get the current connection state back directly after we updated the “bluetoothIsConnected” variable.

The next function is use to send commands directly to the BLE device.

We will send a command, and then wait for the response. These commands always start with “AT+” and will return something like “OK+” when the command was received successfully by the BLE module.

An example:  
We pass “AT+RESET” to the device, we will then receive “OK+RESET” back when the BLE module successfully received the command. The BLE device will then execute the command and restarts the device.

This function is a bit more complex, and will be split into segments.

Text

Description automatically generated

The “sendBTCommand” is the function we use to send the commands. It can receive 3 parameters in total.

“command”:  
This is the command we want to send to the BLE device

“response”:

This is the expected response we should get back after sending the command.

“timeout” (set to 1000ms by default):

This is the amount of time we wait for a response, if it took longer then it will stop the while loop used to gather the data for the response.

We use “Serial2” on the ESP32 to communicate with the BLE bluetooth device. This is using the pin 16 (RX) and pin 17 (TX). The ESP32 automatically translates these 2 pins the the “Serial2” instance.

We then send the command to the BLE module using “Serial2.print(command)”. \*Note that we are using “print” here, and not “println”. “println” will automatically close the command send with an enter character (“\n”). The BLE device doesn’t support this, so by using the “print” function, it will just send the command without any extra closing character.

A picture containing text

Description automatically generated

We define a new string variable called “serialInput” which is set as an empty string (“”).

We will add all the incoming data to this string over time. Serial communication doesn’t send the data in 1 go, but it sends it character by character. So we will need a loop that can catch all these characters and append them to the “serialInput” variable.

We also tell the Arduino te reserve up to 200 bytes for this string using “serialInput.reserve(200)” This is probally not a neccesity, but makes sure we can append longer messages.

We also create a boolean “endOfLine” which is set to false. This variable is used to know whether the end of the data was received. This variable will be used to run the while loop.

Text

Description automatically generated

Here we are creating a variable called “startMillis” which will be set to the value that “millis()” returns (an integer). “millis()” is an Arduino function (which also works on the ESP32) that tells us how many milliseconds has passed since the arduino started. We will use this value to determine how much time has passed in the “while” loop to stop the loop if no data was passed back within a certain time.

Next we will start a while loop, which will keep on running as long as “endOfLine” is false: “while(!endOfLine)”. Inside of the loop we will again get the time passed since the Arduino was turned on using the “millis()” function and set this to the “currentMillis” variable.

We can now calculate to total milliseconds that passed since the while loop was started.

By substracting the startMillis from the currentMillis we know how many milliseconds passed.

Now that we know how many milliseconds went by in the loop, we can do “if(millisPassed >= timeout)” to check if we passed the timeout (defaults to 1000) that was passed as a variable.

A picture containing graphical user interface

Description automatically generated

Since this while loop is triggered very quickly (10.000 of times or more per second) there is a change that no characters are currently coming in. We can determine this by checking if “Serial2.available()” is false. If this is the case, we will tell the loop to continue. This will stop the rest of the while loop from executing, and make it start the while from the start again.

Text

Description automatically generated with medium confidence

Here we are receiving the character that came in through the Serial comminucation.

We receive this by using “Serial2.read()”. We will then set this to the “newChar” variable.

We then append this character to the “serialInput” string created before by doing “serialInput += newChar”. (+= is short for “serialInput = serialInput + newChar).

A picture containing text

Description automatically generated

When the serialInput ends with the expected reponse (that we passed as a parameter) we can stop the while loop by setting “endOfLine” to true. The reason we are using the “endsWith” function, is because for some commands we will receive more then just the expected response alone. And example would be when we try to connect to another device.

It will first return “OK+CONNA”, to tell us it will try to start the connection.

Then if it wasn’t able to do this it will will return “OK+CONNF”

Right now the “serialInput” variable will hold “OK+CONNAOK+CONNF”. Due to this we always look if the string ends with the response we expect.

This is where the while loop ends (“}”).

Text

Description automatically generated with low confidence

At the end of this function we will return the serialInput. This way we know in our code if the response was returned. If the response did not come true, the serialInput will always return an empty string (“”).

Text

Description automatically generated

This function will check if the BLE device is still in the correct “state”. As explained at the start of this documentation, there are some cases where the device out of nowhere completely resets itsself to factory mode.

We also define the “initializedBLE” variable here, which is set to false. We use this variable to make sure we only run this code the first time when the ESP32 starts up.

Graphical user interface, text

Description automatically generated

The structure that is shown in this code snippet is used for almost all the code in the “initializeBLE()” function. I will explain it here once, but won’t repeat it for the following code snippets.

When the ESP32 starts, we always want to reset the device. We can achieve this using the “sendBTCommand()” we explained before. We then pass the “AT+RESET” as the command and the “AT+RESET” as the expected response. We assign the output of this function to the “setBLEReset” variable.

If the “setBLEReset” now hold an empty string, it means that we did not receive the expected response. When that is the case we will use “Serial.println()” to output an error message to the Serial monitor in the Arduino IDE. Since we didn’t get a response, there is nothing else we can do here since the rest of this function will most likely fail as well. So we will use a return to stop the rest of the code from executing.

If we did get a reponse, it means the BLE device is now restarting. Lets give it a short moment to restart using “delay(1000)”.

Graphical user interface, text, application

Description automatically generated

When the BLE module is in factory state, there will be a blinking light to give us visual feedback that the device is currently not connected. The problem is however that this LED is connected to the state pin, which we use to know if we are connected to another device (PIN 4). If the LED is blinking it will give use false readings, because the PIN will return HIGH / LOW depending on the blinking state of the LED.

We solve this by using the “AT+PIO11” command, this will make it so the LED is off when its not connected, and on when it is connected.

All these commands can take some time to activate, it is luckely possible to also check which state the BLE device is currently in. We can use this to determine if the module is currently in the correct state or not.

We use “String BLEIsBlinking = sendBTCommand(“AT+PIO?”, “OK+GET:1”)” to get back the current state of the BLE device.

When the response is “OK+GET:1” we are in the correct state. If not it the BLE module will return “OK+GET:0”. As mentioned before, the “sendBTCommand()” ALWAYS returns an empty string if it was not able to get the expected response. So when the BLE module is currently not in the correct state, the “BLEIsBlinking” variable we defined will be an empty string.

When the “BLEIsBlinking” variable is indeed an empty string, we will continue to send the command to actually put it in the correct state. This is followed by another command to restart the device, since the new set state won’t be picked up untill the device is rebooted.

Graphical user interface, text, application, email

Description automatically generated

This code snippet works very simular as the one we just explained, however it solved a different problem.

When the device is in factory mode, it will automatically transmit “notifications” about its connection state. This would imply that we are receiving incoming data from the BLE bluetooth device that is not coming from the robot. To prevent this we can use the “AT+NOTI0” command. This will disable notifications on the BLE device.

Just as before we will first check which notification state we are currently in, when this is not in state 0 (“OK+Get:0”) we will send the command to set it to the correct notification state. After that we again restart the device so the new notification state is picked up.

Graphical user interface, text, application, email

Description automatically generated

This is where we are setting the “master” role to the device so we are able to connect to another BLE device.

We again first check if we currently are in the correct role state, this is done using the “AT+ROLE?” command. When this returns “OK+Get:1” it means we are currently in the correct master role.

If this is not the case, we will set it in the correct role.

After that will also send the “AT+IMME1” command, this command makes sure that the device doesn’t go in “auto connect” mode. Because when it does, we are not able to connect to another device using the MAC address (which we defined at the top of the page).

We then restart the device so these new settings are applied.

Text

Description automatically generated with low confidence

At the end of the function we will set the “initializedBLE” to true, this will prevent us from running this function a second time.

Graphical user interface, text, application, email

Description automatically generated

The “connectToBLEDevice()” function will try to connect to the robot’s BLE bluetooth.

If the “initializedBLE” variable is false, it means we didn’t check the BLE device’s state yet to determine if everything is setup correctly. If this is the case, we will run the “initializeBLE()” function (defined above) to make sure everything is in order.

We then will try to connect to the other device. We do this by sending the “AT+CO0” command. We will have to paste the MAC address of the device we want to connect to at the end of this command, so it knows it has to connect to this specific device.

“String("AT+CO0")+String(SirSA\_BT\_ADDRESS)” will therefore end up looking like:

AT+CO00035FF0BDAD9

We will try to wait for the “not connected” response (“OK+CONNF”) for up to 3 seconds. The reason we wait for the “not connected” response is because when the connection is actually established, we will receive nothing at all. We simply give a timeout of 3 seconds so the module has 3seconds the time to establish the connection all together. (Since the “sendBTCommand()” function will wait for this timeout before it stops the while loop).

After this we run the “isConnected()” function that was explained earlier, this will check if we are currently connected using the STATE pin (PIN 4) and will assign this to the global variable “bluetoothIsConnected” that we use in other places of our code.

The following function will be split in segments since there is quite a lot going on.

Text

Description automatically generated with medium confidence

The “getSerialString(): function is used to receive data from the other BLE bluetooth device.

It will return an empty string (“”) when no message was received, and it will return the actual receive message if a message is received.

We define a new string variable called “serialInput” which is set as an empty string (“”).

We will add all the incoming data to this string over time. Serial communication doesn’t send the data in 1 go, but it sends it character by character. So we will need a loop that can catch all these characters and append them to the “serialInput” variable.

We also tell the Arduino te reserve up to 200 bytes for this string using “serialInput.reserve(200)” This is probally not a neccesity, but makes sure we can append longer messages.

For this device we are using Serial2 on the arduino, this is TX pin 16 and RX pin 17. The arduino Mega automatically translate this to the “Serial2” instance, which allows us to read the data.

When if(!Serial2.available()) is true, it means that no data is currently coming in.

If this is the case there is no reason to continue with the rest of the function, so we return the “serialInput” variable which currently is an empty string.

Text

Description automatically generated

Here we are creating a variable called “startMillis” which will be set to the value that “millis()” returns (an integer). “millis()” is an Arduino function that tells us how many milliseconds has passed since the arduino started. We will use this value to determine how much time has passed in the “while” loop to stop the loop if no data was passed back within a certain time.

We also create a boolean “endOfLine” which is set to false. This variable is used to know whether the end of the data was received. This will always be an enter whitespace character (which can be read as the “\n” character).

Next we will start a while loop, which will keep on running as long as “endOfLine” is false: “while(!endOfLine)”. Inside of the loop we will again get the time passed since the Arduino was turned on using the “millis()” function and set this to the “currentMillis” variable.

We can now calculate to total seconds that passed since the while loop was started.

By substracting the startMillis from the currentMillis we know how many milliseconds passed. If then we divide this by 1000 we get the amount of seconds.

This is exactly what we do when we set the “seconds” variable.

Now that we know how many seconds have been that went by in the loop, we can do “if(seconds >= 1)” to check if more then 1 seconds was passed.

\*Something to note here is that the arduino executes code really fast, and in 1second it will probally have cycled the while loop more then 10 thousands of times!

If more then 1 seconds passed, we will print a message to the Serial monitor from the Arduino IDE to inform that the message took to long.

After that we set the serialInput back to an empty string, and set the endOfLine to true so the while loop stops.

Text

Description automatically generated with low confidence

If for whatever reason we lose connection with the other device while the loop is going, we also want to stop the loop. This is done using the “if(!isBluetoothConnected())” statement. When this returns true we will again set the serialInput to an empty string (“”) and set “endOfLine” to true to stop the while loop.

Graphical user interface, text, application

Description automatically generated

Since this while loop is triggered very quickly (10.000 of times or more per second) there is a change that no characters are currently coming in. We can determine this by checking if “Serial2.available()” is false. If this is the case, we will tell the loop to continue. This will stop the rest of the while loop from executing, and make it start the while from the start again.

Graphical user interface, text, application

Description automatically generated

Here we are receiving the character that came in through the Serial comminucation.

We receive this by using “(char)Serial2.read()”. Char here defines the type that the function will return. We will then set this to the “newChar” variable.

We then append this character to the “serialInput” string created before by doing “serialInput += newChar”. (+= is short for “serialInput = serialInput + newChar).

A picture containing text

Description automatically generated

This is where we listen if we received the last character to end the serial communcation. We will always receive an enter character (“\n”). So if(newChar == “\n”) we know the communcation is done, and we can set the “endOfLine” to true to stop the while loop.

This is also where the code for the while loop ends. (the final “}” that you can see)

Text

Description automatically generated with medium confidence

This is the last part of the “getSerialInput()” function. Here we trim the “serialInput” string. What this does is remove all the whitespace at the start and at the end of the string. In our case it will remove the last enter character (“\n”) from the serialInput string.

We then return the “serialInput” back to the initator of this function so it can be used to do something depending on the data we received.

Text

Description automatically generated

The “handleBLEResponse” function will handle the incomming command from the robot’s BLE bluetooth device, it uses the received command as a parameter.

We only listen to 2 commands.

“1” start timer:

If the BLESerialInput == “1”, we will start the timer. However, if the timer is already running there is no need to. We will use the “timerRunning” variable (defined in “LedMatrix.h”) to determine this. When the timerRunning variable is true, we will use return stop the rest of the code in this function from executing.

When the timer is currently not running, we will use the “setLedAnimatonType(“1”)” (defined in “LedStrip.h”) to set the ledstrip to play the blue wave animation.

After that we set the “startTheTimer” variable to true. This variable is used in “LedMatrix.h” to know if the timer has to run. The reason we are using a variable like this is because this code is all currently running on CPU core 0, and we want the timer to run in CPU core 1.

We set the variable to true, so the main loop() function can trigger the “runLedMatrix()” function to start the timer when the “startTheTimer” is true. This will then run the ledMatrix code on Core 1.

“2” winner:

This command is used to know we won the game. We will use the “setLedAnimatonType(“1”)” (defined in “LedStrip.h”) to set the ledstrip to play the rainbow animation. We will also stop the timer using the “stopTimer()” function (defined in “LedMatrix.h”.

We will then delay the code for 24000ms (remember, the ledStrip runs on core 1, this code on core 0, so the delay won’t block the ledStrip animation) to wait untill the robots victory dance is done. We then reset the matrixBoard using the “resetMatrixBoard()” function (defined in “LedMatrix.h”) which will set the time back to “1:00” and the led animation back to “1” (blue wave animation).

Graphical user interface, text, application, email

Description automatically generated

The ”establishBLEConnection” will run as a timer task to for up to 3 times to try and connection when the ESP32 starts.

We first create a new variable called “tryToConnectTimer” with the “Timer<>::Task” type (this is a type used for timer tasks). This is the variable we will use to attach the timer task to that triggers the “establishBLEConnection()” function.

We then define “BLEConnectionAttempts”, which will be a number holding the amount of tries we did so far.

We also define a boolean (true/false) “ignoredConnection”. This variable will be used to know if we can ignore the current connection state and start the timer anyways. This is used to start the timer automatically if within 15s no connection was established.

The “establishBLEConnection” will need to return a boolean to tell the timer it can run this task again. We will also need to accept the “void\*” (any) type for the parameter, even if we don’t use any parameters all together. This is because it is “possible” to send a parameter along, and therefor the type need to match this.

A picture containing text

Description automatically generated

We check if we are currently connected using the “isConnected()” function. If this is the case we will cancel the timer task that is currently running this function. We will also return false on top of that tell the timer is should not run this task again.

A picture containing text

Description automatically generated

When this is attempt 3, we will start the timer by setting the “startTheTimer” to true.

We will set the “ignoreConnection” variable to true, so we know we are currently ignoring the connection. This variable is used a little later in the code to make sure we only run the timer that triggers this function when we need to.

After all this, we will cancel the timer running this function, and also return false on top of that tell the timer is should not run this task again.

Text

Description automatically generated

The “connectToBLEDevice()” will try to establish the connection with the other BLE device, as explained before.

We will then add 1 to the “BLEConnectionAttempts” variable to keep track of the total attempts that we did to connect. (BLEConnectionAttempts++ == BLEConnectionAttempts = BLEConnectionAttempts + 1).

At the end of the function we will return true to tell the timer it can run this function again.

Graphical user interface, text, application

Description automatically generated

This function runs in a while loop from the “bluetoothSetup()” function, which will be explained later, and will listen to the incoming bluetooth commands.

If we are currently not connected and we are currently not ignoring the connection (!isConnected() && !ignoredConnection) we will start the timer that triggers the “establishBLEConnection()” that was explained before.

However, we only want to start this timer once. And since this code is running in the main loop(), it will be triggered a lot more then that. We use the “tryToConnectTimer” variable (explained before) to determine if there is already an active task running the establishBLEConnection function. When this is not the case, we will start a new timer task that runs this function every 5000ms (at least, untill the tasks gets cancelled).

When we are currently not connected and we are currently not ignoring the connection there is no need to run the rest of the code in this function, so we use return to stop the rest of the function from being executed.

Graphical user interface, text

Description automatically generated

If the timer is currently not running (“if(!timerRunning)”) we will trigger the “resetMatrixBoard()” function (defined in “LedMatrix.h”). This will make sure the LedMatrix updates the current bluetooth state (it will add the “.” at the end of the time to give a visiual cue when we are not connected to another BLE device).

Text

Description automatically generated

When we are still not connected, or when we lost connection, we will trigger the “connectToBLEDevice()” function to try and (re-)connect.

Text

Description automatically generated with medium confidence

When we are connected, we will set the “ignoredConnection” to true, so we don’t trigger the timer explained before in the case we lose the connection at some point.

We then will get the incoming serial command from the “getSerialInput()” function and assign this to the “BLESerialInput” variable.

If BLESerialInput is not an empty string (“”), it means we received a command.

We then use the “handleBLEResponse” and pass the received command (“BLESerialInput”) as a parameter, to handle what to do with this command accordingly.

Text

Description automatically generated with medium confidence

This function runs when the ESP32 starts.

It will initialize the BT\_STATE\_PIN (4) as INPUT using the “pinMode()” function, so we can read it value and know if we are currently connected to another BLE device or not.

We then start the serial communication between the BLE device and the ESP32 using “Serial2.begin(9600)” 9600 is the BAUDRATE, which essentially is the port that is used to communicate between the 2 devices.

We will wait for 2 seconds (delay(2000)) to give it some extra time to initialize everything.

We then run a while loop that runs forever and ever. The reason we do this is because the “bluetoothSetup()” is running on CPU core 0. So we essentialy create our own main “loop()” function here, just like the “loop()” in “ESP32\_VROOM.ino” that runs on CPU core 1.

Inside this loop we will trigger the “listenToBluetooth()” function, which will try to connect to the BLE device on the robot, and will listen to incomming commands.

Core 0 would sometimes crash because the loop was running to quickly. There is a “dogwatcher” (some logic that checks the ESP32’s state) on the ESP32 to prevent this from happening because it can cause the core to overheat. To avoid everything from crashing, we simply add a “delay(10)” so it gets a 10ms break between each run in the loop.