# VoiceCommands.h

This file is used to communicate with the AMR\_voice app used on an Android phone (iPhone wont work). It will open a connection and allow anyone to connect to it.

The app always sends commands in the **\*message#** format, allowing us to easily detect when the command starts (using the \* character) or when a message ends (using the # character).

A picture containing text

Description automatically generated

We use the “SoftwareSerial” library to communicate with the Robot.

This library allows us to do serial communication over pins that don’t support it.

We initialize the SoftwareSerial library instance on the “SirSA\_BT” variable using **PIN 4** and **PIN 33**.

Text

Description automatically generated with low confidence

We include “BluetoothSerial.h” which is used to use Bluetooth classic. This will be used to communicate with our phone.

We also defined a variable “BT\_DEVICE\_NAME” which will be the Bluetooth name we will be able to pair with (Sir S.A. voice control).

**We initialize the BluetoothSerial library on the voiceBT variable.**

And we define a global boolean variable that we can use in the rest of the file to determine if there currently is an active Bluetooth connection. We will initialize this variable (isConnected) with the value “false” (since we are not connected at the start).

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

Text

Description automatically generated with low confidence

The “getBTVoiceCommand()” function is used to receive data from the other Bluetooth device. It will return an empty string (“”) when no message was received, and it will return the actual received 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 voiceBT on the ESP32 that we defined before.

When if(!!voiceBT.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 then create an “buildSerialInput” variable which is set to false. This variable will **ignore all incoming data untill a certain character is received**. This will always be the “**\***” character due to the App we use on the phone.

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 “**#**” character due to the App we use on the phone.

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 went by in the loop, we can do “if(seconds >= 3)” to check if more then 3 seconds was passed.

If more then 3 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.

Graphical user interface, text

Description automatically generated

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(!isConnected())” 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.



Since this while loop is triggered very quickly (10.000 of times or more per second) there is a chance that no characters are currently coming in. We can determine this by checking if “!voiceBT.available() ” is true. 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.

A picture containing text

Description automatically generated

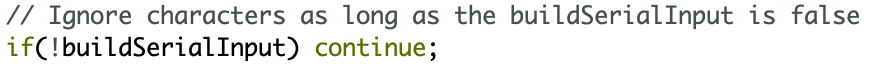
Here we are receiving the character that came in through the Serial comminucation.

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

Text

Description automatically generated with medium confidence

We check if the character came in matches the first character of the command (\*). If that is the case we will set the “buildSerialInput” variable to true, so we can start appending characters to the “serialInput” string. We then run continue to stop the rest of the loop and start again from the beginning. This way we make sure this first character (\*) is not added to the “serialInput” string.



If any other character came in, and we didn’t detect the first character of the command yet (so buildSerialInput will be false), we won’t append any characters to the serialInput. To prevent this from happening we will run continue and start back at the beginning of the loop for as long as “buildSerialInput” is false.

Graphical user interface, text

Description automatically generated with medium confidence

We then check if the incoming character is a “#”, if that is the case this is the end of the command. We will stop the loop by setting “endOfLine” to true, and will run continue so that this last character is not appended to the “serialString” string.

Text

Description automatically generated with medium confidence

We then append this character to the “serialInput” string created before by doing “serialInput += newChar”.

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

Description automatically generated with medium confidence

This is the last part of the “getBTVoiceCommand()” 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 “listenToVoiceCommands()” function is used to handle the incomming voice commands and is triggered from the main “loop()” function. If it’s a command that we support it will pass the corresponding command the Robot can use using the “SirSA\_BT.println()” function.

If we are currently not connected to bluetooth (if(!isConnected)) there is not much we can do, so we use a return to stop the rest of the code from executing.

We then create a new variable “voiceCommand” and set it to the value returned by the “getBTVoiceCommand()” function (explained before).

If this is returning an empty string (“”) no command was received, so we use a return to stop the rest of the code from executing.

When a command was returned, we will output it to the Serial Monitor from the Arduino IDE for debugging.

The following commands are used on the arduino:

Graphical user interface, text, application

Description automatically generated

“if(voiceCommand == “straight”):

This means “drive forwards” and will send the corresponding command (“11”) to the robot using “SirSA\_BT.println(“11”)”.

“else if(voiceCommand == “go”):

This means “drive forwards” and will send the corresponding command (“11”) to the robot using “SirSA\_BT.println(“11”)”.

“else if(voiceCommand == “forward”):

This means “drive forwards” and will send the corresponding command (“11”) to the robot using “SirSA\_BT.println(“11”)”.

“else if(voiceCommand == “reverse”):

This means “drive backwards” and will send the corresponding command (“12”) to the robot using “SirSA\_BT.println(“12”)”.

“else if(voiceCommand == “backward”):

This means “drive backwards” and will send the corresponding command (“12”) to the robot using “SirSA\_BT.println(“12”)”.

“else if(voiceCommand == “back”):

This means “drive backwards” and will send the corresponding command (“12”) to the robot using “SirSA\_BT.println(“12”)”.

“else if(voiceCommand == “stop”):

This means “stop driving” and will send the corresponding command (“13”) to the robot using “SirSA\_BT.println(“13”)”.

“else if(voiceCommand == “left”):

This means “drive left” and will send the corresponding command (“14”) to the robot using “SirSA\_BT.println(“14”)”.

“else if(voiceCommand == “right”):

This means “drive right” and will send the corresponding command (“15”) to the robot using “SirSA\_BT.println(“15”)”.

“else if(voiceCommand == “dance”):

This will make the robot dance and will send the corresponding command (“16”) to the robot using “SirSA\_BT.println(“16”)”.

“else if(voiceCommand == “drive”):

This will make the robot go into self driving mode and will send the corresponding command (“17”) to the robot using “SirSA\_BT.println(“17”)”.

Graphical user interface

Description automatically generated with medium confidence

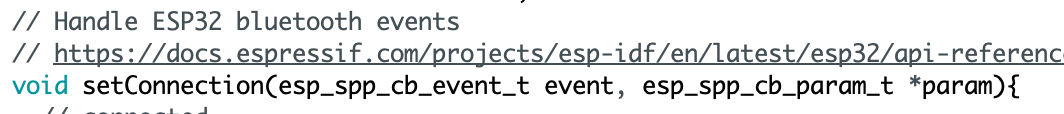
This function is used with the “timer” library and therefor needs to return a boolean to tell the timer it can run the function again in the next cycle. Since the timer can pass along a parameter of any type (void\*) we will also need set this type in our function, even if we don’t receive any parameters.

SirSA\_BT.println(“01”); will send the “01” command to the robot, which on the robot will trigger an MP3 file that says “*I can’t hear you, please come closer*”.

We then return true to tell the timer it can run this function again in the next cycle.



Here we first create a new variable of type “Timer<>::Task” called “bluetoothFeedbackTimer” which is undefined at first. This is a variable we can set to the value returned when we initialize a timer. This allows us to afterwards cancel the task using the “timer.cancel()” function, passing the variable holding the task (bluetoothFeedbackTimer) as a parameter.



The “setConnection()” is a function that gets triggered every time an event happens on the voiceBT instance. It will give us back the “event” with type “esp\_spp\_cb\_event\_t” (which is coming from the BluetoothSerial library, not sure what “type” it is).

It also gives back “params” as a second parameter of type “esp\_spp\_cb\_param\_t” (which again is coming from the BluetoothSerial library, not sure what “type” it is).

There is a variety of events, and here we only listen to the “**connection started**” and “**connection ended**” events.

Here you can see all events that are available:

<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/bluetooth/esp_spp.html#_CPPv2N18esp_spp_cb_param_t22spp_srv_open_evt_paramE>

Graphical user interface, text, application

Description automatically generated

If the event == ESP\_SPP\_SRV\_OPEN\_EVT (see link above) is true, a **connection was established**.

We check if the bluetoothFeedbackTimer variable is currently set, if this is the case the timer is currently running.

We then cancel the timer task using “timer.cancel(bluetoothFeedbackTimer)” and send command “02” to the arduino using “SirSA\_BT.println(“02”)”, which will play the “*Lets go*” MP3 file to let us know we reconnected.

We then set the “isConnected” variable to true, so we know in the rest of the code that we are currently connected.

Since we already received an event, there is nothing else we need to do. So we send a return to stop the rest of code from executing.

Graphical user interface, text, application

Description automatically generated

If the event == ESP\_SPP\_CLOSE\_EVT (again, see link above) is true, we got **disconnected**.

If this is the case we will trigger the “sendDisconnected();” function, which will pass the “0” command to the robot so it plays the “*I can’t hear you, please come closer*” MP3 file.

Since we declared this function before to accept the “void\*” as a parameter (due to the timer) we will have to pass “something” along, else C++ will complain that we didn’t pass a parameter to a function that expects one. So we are sending along “NULL” to bypass this.

We then start a new timer task and assign this to the bluetoothFeedbackTimer variable so we are allowed to cancel the task (as explained before) when the bluetooth reconnects.

The timer will trigger the “sendDisconnected();” every 20000ms (20s).

After this is done, we set the “isConnected” variable to false (since we are no longer connected) and use a return to stop the function from executing the rest of the code.

Text

Description automatically generated

The “voiceCommandSetup()” function runs when the ESP32 starts.

It will begin the SirSA\_BT instance (used to talk to the arduino) using “SirSA\_BT.begin(9600)”. 9600 is the BAUD, which is essentially a port we use to communicate through to talk to the arduino.

We also begin the Bluetooth using “voiceBT.begin(BT\_DEVICE\_NAME)”, where BT\_DEVICE\_NAME is the variable we defined at the top fo the file. This will start the bluetooth connection and allows use to connect with it.

We then add an “even” callback to the “voiceBT” instance using “voiceBT.register\_callback(setConnection);”. This will trigger the “setConnection()” function every single time an event happens on the bluetooth device.

In our case we use it for the connected and disconnected events, but more events exist. Please refer to the link shared before where the “setConnection()” was explained.