UartRemote

Ste7an

Sep 11, 2022

CONTENTS

This is a library for robust, near real-time communication between two UART devices. We developed it on python 3.9 with LEGO EV3, SPIKE Prime and other MicroPython (ESP/STM32) modules. The library is available on github: UartRemote on GitHub. The library has the following properties:

- It is fast enough to read sensor data at 30-50Hz.
- It is fully symmetrical, so master and slave can have the same import.
- It includes a RAW REPL mode to upload code to a slave module. This means you can develop code for both modules in one file.
- It is implemented in MicroPython and Arduino/C code. With arduino code, much higher sensor reading speeds are possible, but flashing is a bit less user friendly.
- The library has a command loop to wait and listen for calls. That loop is customizable and non-blocking so you can add your own code to it.
- The python-struct-like encoding is included in the payload, so the other side always knows how to decode it.
- Compatable with most RS232-TTL 3.3v/5v converter board to further expand i/o possibilities.
- Remote modiule importing

Usage: you can use all of parts of this library for your own projects. Please give us credits at least. We put a lot spare time in this. You are also welcome to contribute. Please fork and PR.

CHAPTER

ONE

CONTENTS

1.1 UartRemote Library

This library implements a class with methods that help to set up robust communication between instances running MicroPython which are connected over a UART interface.

The library is available on github: UartRemote on GitHub.

1.1.1 Platforms

The following platforms are supported:

- Lego EV3
- Lego Mindstroms Robot Inventor 51515
- Lego SPIKE Prime
- Espressif ESP8266
- Espressif ESP32
- Espressif ESP32-S2
- OpenMV H7 and M7 (STM32 chipset)
- MaixPy (K210 chipset)
- Windows
- Mac OSX

These platforms are automatically detected by querying `sys.platform`. Within `uartremote.py` these platforms are defined by the constants: `_EV3,_SPIKE,_ESP8266,_ESP32,_ESP32_S2,_H7,_K210,_MAC,_WIN32`, respectively. There are small differences in the implementation of MicroPython for these platforms. The `UartRemote` takes these into account based on the platform type.

1.1.2 Constructor

class uartremote.**UartRemote**(*port=0*, *baudrate=115200*, *timeout=1500*, *debug=False*, *rx_pin=18*, *tx_pin=19*)

Construct a UartRemote object on the port given by:

- `port` identifies a port for using the UART.

`port` is board specific:

- SPIKE: port = : has one I2S bus with id=2.
- ESP8266: port = 0.
- ESP32: Use 1 for UART1.

The following keyword arguments are supported on all platforms:

- `port` is the hardware uart port used to by *UartRemote* class; for LEGO it indicated the harware port, i.e. "A"
- `baudrate` is the baudrate at which the UART communicates, defaults to 115200
- `timeout` is the timeout (in ms) for waiting for data coming in on the UART in the `receive_command()`, default to 1500
- `debug` is a Boolean flag to generate debug output, default to False

Only for ESP32 platform:

- `rx_pin` is only used for ESP32 and indicates the pin on which the UART will receive information, default to 18
- `tx_pin` is the pin on which the UART will receive information, default to 19

1.1.3 Methods

UartRemote.**flush**()

Flushes the read buffer, by reading all remaining bytes from the Uart.

UartRemote.**available**()

Return a non zero value if there is a received command available. Note: on the SPIKE prime, you should use the `receive_command` or the `execute_command`, always with the parameter `reply=False`, after using the `available()` method.

UartRemote.**send_command**(*command*, **type_data*)

Sends a command `command`. `*type_data` are a number of argument that consist of a type defintion `t`, followed by one ore more variables of the type corresponding with the paramater `t`.

For example:

```
ur=UartRemote()
ur.send_command('led_color','4B',n,t,g,b)
# will encode a command to remotely calls the function ``led_color``
# where the values of the variables ``n,t,g,b`` are passed to that function.
```

UartRemote.**receive_command**(*wait=True*)

Receives a command and returns a tuple `(<command>, <data>)`. If there is a failure, the `<command>` will be equal to *'err'*. If `wait` is True, the methods waits until it receives a command.

UartRemote.**call**(*command*, **type_data*, ***kwargs*)

Sends a command to a remote host that is waiting for a call and will wait until an answer comes back. Optionally a parameter `timeout=...` for the answer is self.timout, or passable as timeout=...

UartRemote.**process_uart**(*self*, *sleep=-2*)

Processes a remote call if there is any. Upon receiving a remote call, the command is processed and the result is send back by internally calling the `reply_command` method. Sleeps for `sleep` ms after every listen. This method is only used in a loop and is non-blocking (as not for the sleep period).

UartRemote.**reply_command**(*self*, *command*, *value*)

Processes the received command by calling the function with name *command(value)* and passes the arguments as defined in `value`. The result of this function call is send back by calling the `send_command(ack_command, result)` method with with the `ack_command` is the received command prepended with *ack_* and the result (if any) is the return value of the function formatted according to the functions format string.

UartRemote.**loop**()

This is an endless loop around the `process_uart` method, replying on all incoming calls. The slave side instants typically has the following code running:

```
from uartremote import *
ur = UartRemote()
ur.loop() # wait for incoming commands
```

UartRemote.**add_module**(*module_name*)

Sends a command to the other side instructing it to import the module with name `module_name`. The `module_name` argument has type string. After importing the module, the remote side calls the function *<module>.add_commands()*. This is a function that you should add to the modules you want to remotely import. See for usage *Example load module*.

UartRemote.**add_command**(*command_function*, *format=''*, *name=None*)

Adds a command *command* to the dictionary of `UartRemote.commands` together with a function name `command_function`. Optionally, if the `command_function` returns any parameters, the `format_string` describes the type of the returned parameters. If the `command_function` does not return a value, the *format_string* is ommited. The dictionary with commands is used by the `UartRemote.reply_command()` method to call the function as defined upon receiving a specific command. As an argument the `data` that is received is used.

Below is an example of how to use the `add_command` method:

```
def example(a,b):
  # example function receiving two arguments and returning the sum of a and b

  return (a+b)

ur.add_command(example,'f')
```

Here `ur` is the instantiation of the `UartRemote` class and the function `example` will return the sum as type float.

UartRemote.**get_remote_commands**()

Returns an array containing the commands available by the remote uartremote. You will see a number of default built-in commands such as *echo*. This method can be used to query the commands that are added by remotely importing a new module. See for usage *Example load module*.

1.1.4 Helper Methods

The methods below are internally called by the methods listed above. You can use these methods should you like to have more low level control.

`Uartremote.command`

Dictionary with the mapping of command name to corresponding functions.

`UartRemote.encode`(*command*, **typedata*)

Encodes a command `command`. `*type_data` are a number of arguments that consist of a type defintion `t`, followed by one ore more variables of the type corresponding with the paramater `t`.

For example:

```
ur=UartRemote()
ur.encode('led_color','4B',1,2,3,4)

>>> b'\x11\tled_color\x024B\x01\x02\x03\x04'
```

`UartRemote.decode`(*bytestr*)

Decodes an encoded bytestring `bytestr` as a tuple with the command and the parameters. If a command without parameters was encoded, the parameters will be `None`.

For example:

```
ur=UartRemote()
ur.decode(b'\x11\tled_color\x024B\x01\x02\x03\x04')

>>> ('led_color', (1, 2, 3, 4))
```

`UartRemote.read_all`()

Returns all bytes that are available in the UART receive buffer.

`UartRemote.force_read`(*self*, *size=1*, *timeout=50*)

Some platforms read too fast from the UART and return 0 or None. This method loops until it receives a valid number of `size` bytes within `timeout` ms.

1.2 Installation

1.2.1 Micropython

The `uartremote.py` needs to be copied to the flash memory of the MicroPython platform. Depending on the platform used, this can be slightly different. Below you will find a description for the installation on the different platforms.

1.2.2 ESP32

We provide an ESP32 firmware image that contains the latest `uartremote.py` module integrated as a frozen module in Ste7an's github repository micropython_ulab_lvgl.

Flash this firmware using the `esptool`:

```
esptool.py --port <serial_port> erase_flash
esptool.py --port <serial_port> --baud 921600 write_flash 0x1000 firmware_ESP32_ULAB_
↪LVGL_SPIRAM_<timestamp>.bin
```

where `<serial_port>` is the port to which the esp32 is connected, and `<timestamp>` the timestamp at which the firmware was build.

For other firmwares, you can manually upload the `uartremote.py` file to the flash memory of the ESP32 module using WebREPL, rshell, or one of the IDE's.

1.2.3 LEGO SPIKE Prime and Robot Inventor 51515

The SPIKE IDE checks the filesystem of the Spike Prime. If it sees any non-system files in the root directory, it triggeres a firmware update. After the firmware update, the non-system files will be deleted. However, files that reside in the `/project` will not be deleted after a firmware update.

Installation of uartremote.py or uartremote.mpy

Open a new Python project in the LEGO Education SPIKE Prime IDE and paste in the content of the file install_uartremote.py. This script is automaically build using a GitHub Action and is committed to the `MicroPython/SPIKE` director of the github repository. Execute the script. Open the console in the IDE. After executing it should show:

```
[10:28:55.389] > writing uartremote.mpy to folder /projects[10:28:55.584] > Finished␣
↪writing uartremote.mpy.
[10:28:55.610] > Checking hash.[10:28:55.686] > Hash generated:  <hash>
[10:28:55.704] > Uartremote library written succesfully. Resetting....
```

The timestanps will be different on your system and <*hash*> will show the sha-256 hash value. Now the uartremote.mpy library is copied to the */project* directory. You can discard the script afterwards.

To use the library include the following line in your python code:

```
from project.uartremote import *
```

Creating the install file

If you have the `mpy-cross` cross compile tool installed, just do this in the SPIKE directory: `./create_install_file.py`

Errors while installing uartremote library

If you see any errors when running the `install_uartremote.py` script, these will be related to wrong hashes. - If the hash of the base64 decoded string is not the same as the hash of the initial uartremote.mpy file you will see

```
Failed hash of base64 input : <hash>
```

- if the hash of the uartremote.mpy file written locally to the hub's filesystem differs from the initial hash you will see:

```
Failed hash of .mpy on SPIKE: <hash>
```

In both cases you can try again by copying the 1`install_uartremote.py`1 again in to an empty Lego Spike project and rerun the code.

1.2.4 LEGO EV3

Copy *uartremote.py* into your project directory or library directory for *uartremote import* *. this also is the same for using in python3 on desktopOS

1.2.5 ESP8266 with rshell

On the ESP8266 copy the ESP8266 library to the board with webREPL or rshell.

Copy the plain .py file or the compiled library .mpy into your device * You can get rshell via:

pip3 install rshell

- If you have a *single device* this will connect

```
rshell -p $(ls /dev/tty.usb*) -b 115200 --editor nano
```

- ...otherwise find your device with:

```
ls /dev/tty.usb*
```

- use your *own* desired modem/serial:

```
rshell -p /dev/tty.usbserial-141230 -b 115200 --editor nano
```

- Your device is now mounted in rshell> as /pyboard
- Use rshell's cp To copy the library:

```
cp Libraries/UartRemote/MicroPython/uartremote.py /pyboard/
```

Alternativly: you can use some IDE's for GUI File managment, such as ThonnyIDE (win/osx/raspi) or Mu IDE if you cannot use rshell command line.

You can edit `/pyboard/boot.py` while you're at it, to configure your wifi connection for LEGO/STM32 using a breakout board.

1.2.6 Arduino

The same UartRemote library is also implemented for Arduino.

1.3 Examples

Basic encoding

1.3.1 Remote module loading

The ESP32 micropython only has a single REPL prompt working properly on UART0 (the primary UART used for uploading firmware). Consequently, the commands in the *UartRemote* library for starting and stopping the remote REPL do not work and we can not use the remote REPL for uploading Python code to the remote micropython instance.

A simple solution would be to initate the code on the ESP32 as *main.py* and have it executing upon reset. This does not allow to change the software running on the ESP32 without reprogramming it.

Therefore, we came up with the following idea. By saving the different programs on the ESP32 as seperate modules, the remote side can choose which module it should load. The loaded module populates the list of remote commands with the functions that it implements. After that the remote commands can be *call*-ed by the remote side. In this way the remote side can decide at run time which commands it can execute on the ESP32.

1.3.2 How to remotely load a module?

We illustrate the way this works by giving an example.

The ESP32 runs the following commands in its `main.py` program:

```
from uartremote import *
ur=UartRemote()          # initialize uartremote on default uart and default uart pins
ur.loop()                # start listing for commands received from the remote instance
```

Furthermore, on the ESP32 we have the following code saved as the module `test.py`:

```
# module test.py

def led(n,r,g,b):
  # code for turning on led n using color (r,g,b)
  # now we only print the received data
  print(n,r,g,b)

def collect_data():
  # code for pulling tuple
  return [('ABC',123),('ABC',123.456)]

def add_commands(ur): # call for adding the functions in this module to UartRemote␣
↪commands
  ur.add_command(led) # does not return any value
  ur.add_command(read_temp,'i') # returns an integer
  ur.add_command(collect_data,'repr') # returns string
```

When the module above is imported, the function `add_commands` will add the two functions that are defined in this module to the current command set of UartRemote. Therefore, this function should be present in your modules that you want to remotely import.

On the master instance (e.g. the Lego robot, where the ESP32 is connected to port 'A'), we use the following code to remotely import the `test` module:

```
# code running on remote instance
from projects.uartremote import *
ur=UartRemote('A')

cmds_before=ur.get_remote_commands()
print('before',cmds_before)

# remotely import module `test`
ur.add_module('test')

cmds_after=ur.get_remote_commands()
print('new commands:',list(set(cmds_after)-set(cmds_before)))

ack,val=ur.call('led','4B',1,100,200,150)

ack,val=ur.call('read_temp')
print('read_temp',val)

ack,val=ur.call('collect_data')
print('collect_data',val)
```

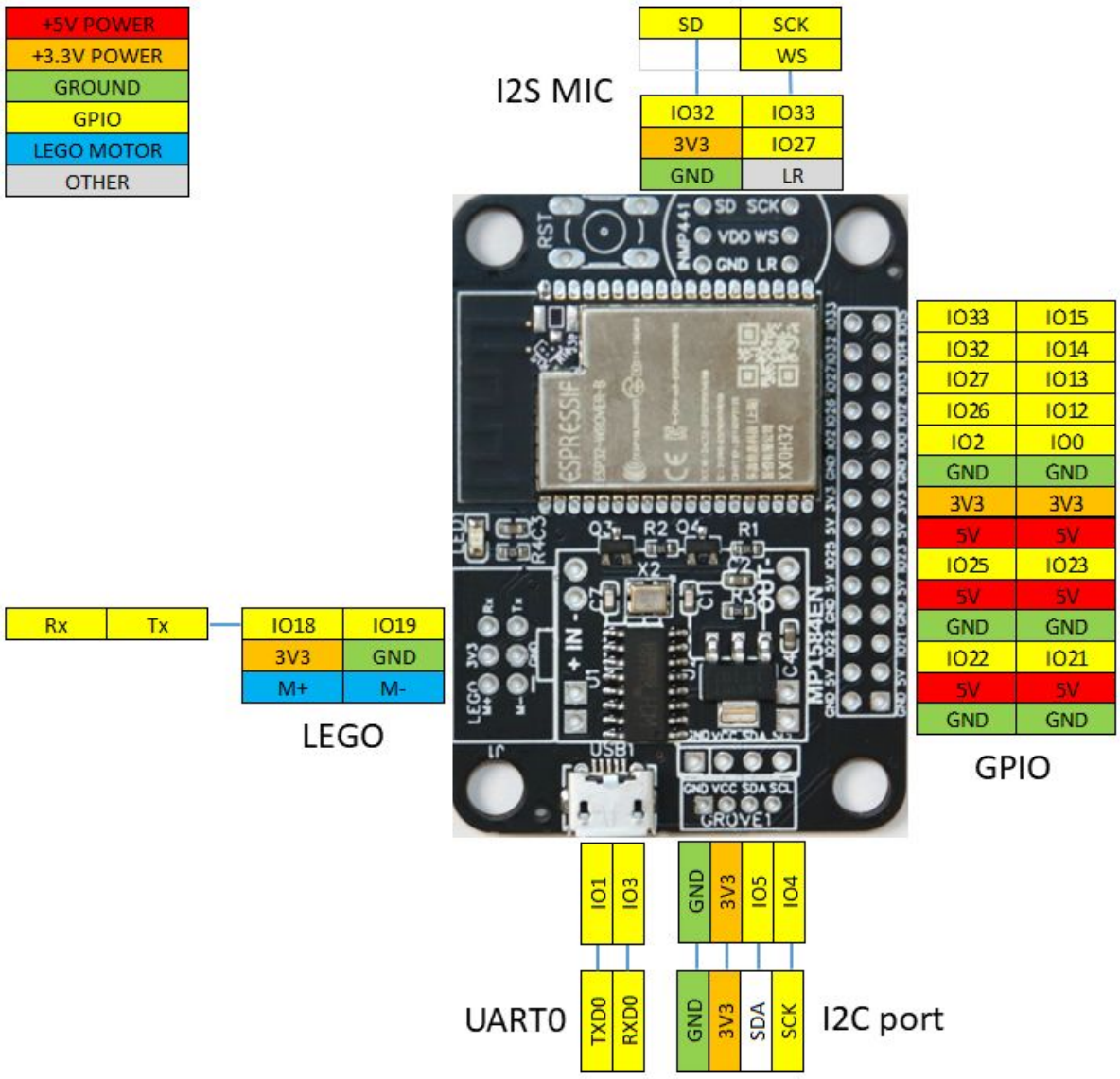
Running this program gives the following output:

```
>>> before ['enable repl', 'disable repl', 'echo', 'raw echo', 'module', 'get_num_
↪commands', 'get_nth_command']
>>> new commands: ['read_temp', 'led']
>>> read_temp 37
>>> collect_data [('ABC, 123'),('ABC',123.456)]
```

and on the ESP32 we see:

```
>>> 1 100 200 150
```

1.4 LMS ESP32 Pinout

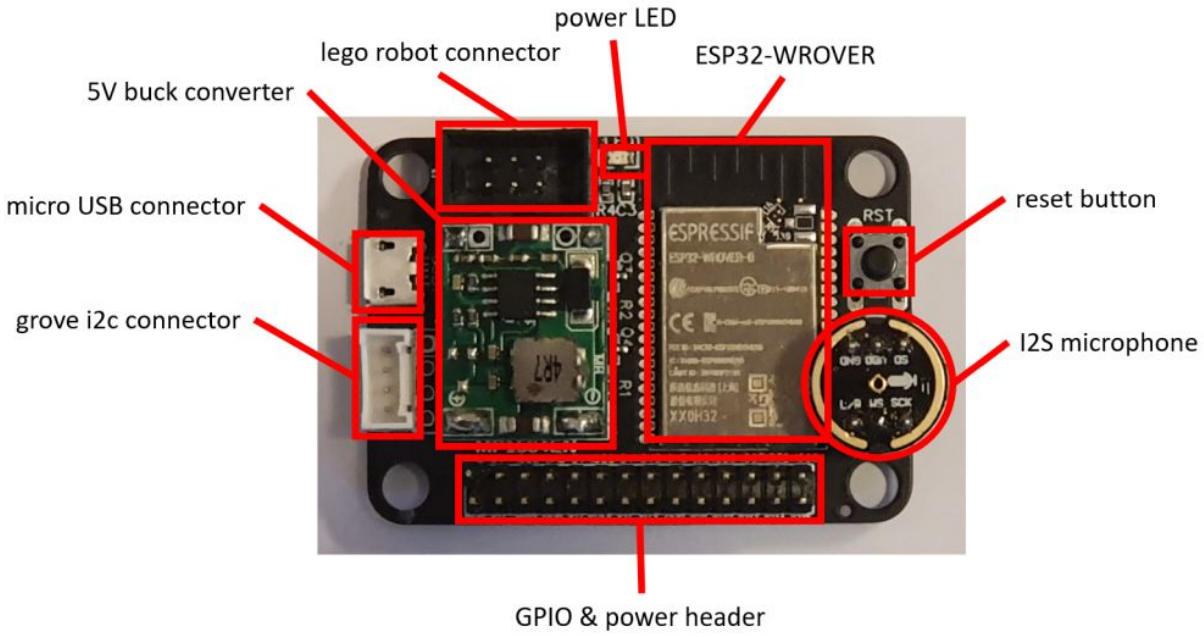


The graph above shows the pinout of the LMS ESP32 expansion board. All ESP32 GPIO pins are indicated. Where they have a specific functions, that is idicated by an extra pin label.

1.5 Hardware

1.5.1 Overview

In the picture below an overviwe of the hardware components of the LMS-ESP32 board is given.



In the photo shown above, the major components and headers are shown. Below we briefly discuss each of these:

- ESP32-WROVER

  This is the microcontroller we use on the board. It is an Esspressif dual core MCU whith the following specifications. - dual 240MHZ core - 520kB SRAM - 4MByte flash memory - 4Mbyte PS-RAM

- power LED

  This LED is connected to the 3.3V power ad will light up when the board is powered.

- micro USB connector

  Used for connecting the board through the build-in usb-uart convertor and for powering the board with 5V

- GPIO power header

  This is the header where you connect your hardware such as servo motors , displays and sensors. There are 5V and 3.3V power pins present on the header

- Groves I2C header

  This is a grove's connector that can be used to connect I2C devices to the board. The connector uses 3.3V power supply.

- I2S microphone

  We provide two 3-pin headers for connecting an *INMP441 MEMS microphone*.

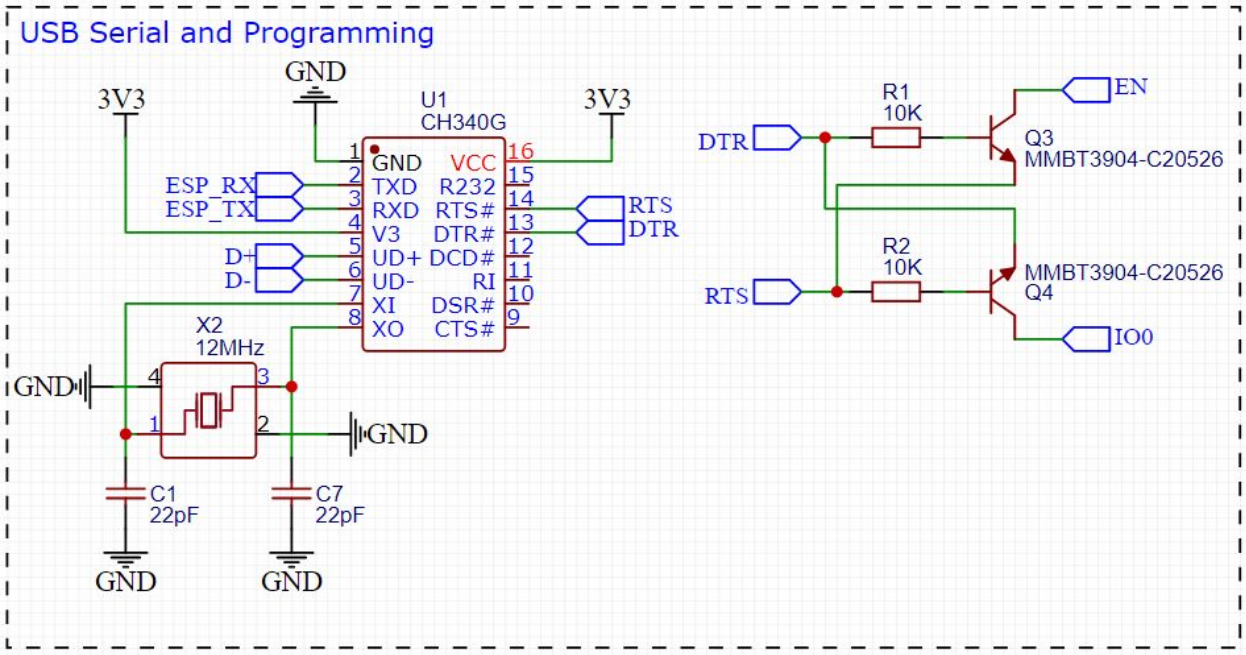
Note: The Microphone is connected using the GPIO pins 27,32 and 33. These Pins are also present on the GPIO header.

- Lego robot connector This connector is used to connect the board to your Lego robot. It connects to the 3.3V power, the UART pins and the M+ pin of the robot.
- Reset button This button is connected to the reset signal of the ESP32

Note: The pinout and maping on ESP32 GPIO pins can be found in the *Pinout*.

1.5.2 USB connector

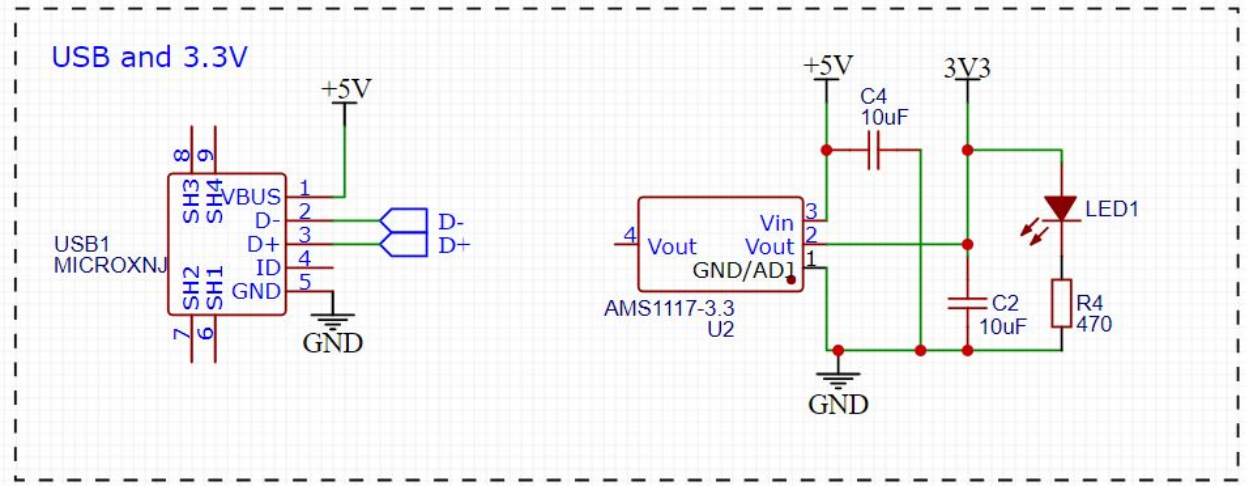
We provide a micro USB connector for powering the board and for communicating with UART0 of the ESP32 using a CH340 Uart USB transceiver. The additional circuit with the two transistors Q3 and Q4 allows the programming tool to switch the ESP32 in program mode using the EN and IO0 pins.



1.5.3 Power supply of the board

When connected to the Lego robot, the board is powered from the robot's 3.3V power supply. When the board is connected to the USB port of a PC, it will get 5V power from the USB port. The 5V is fed to an LMS1117 power regulator with a 3.3V output used for powering the ESP32.

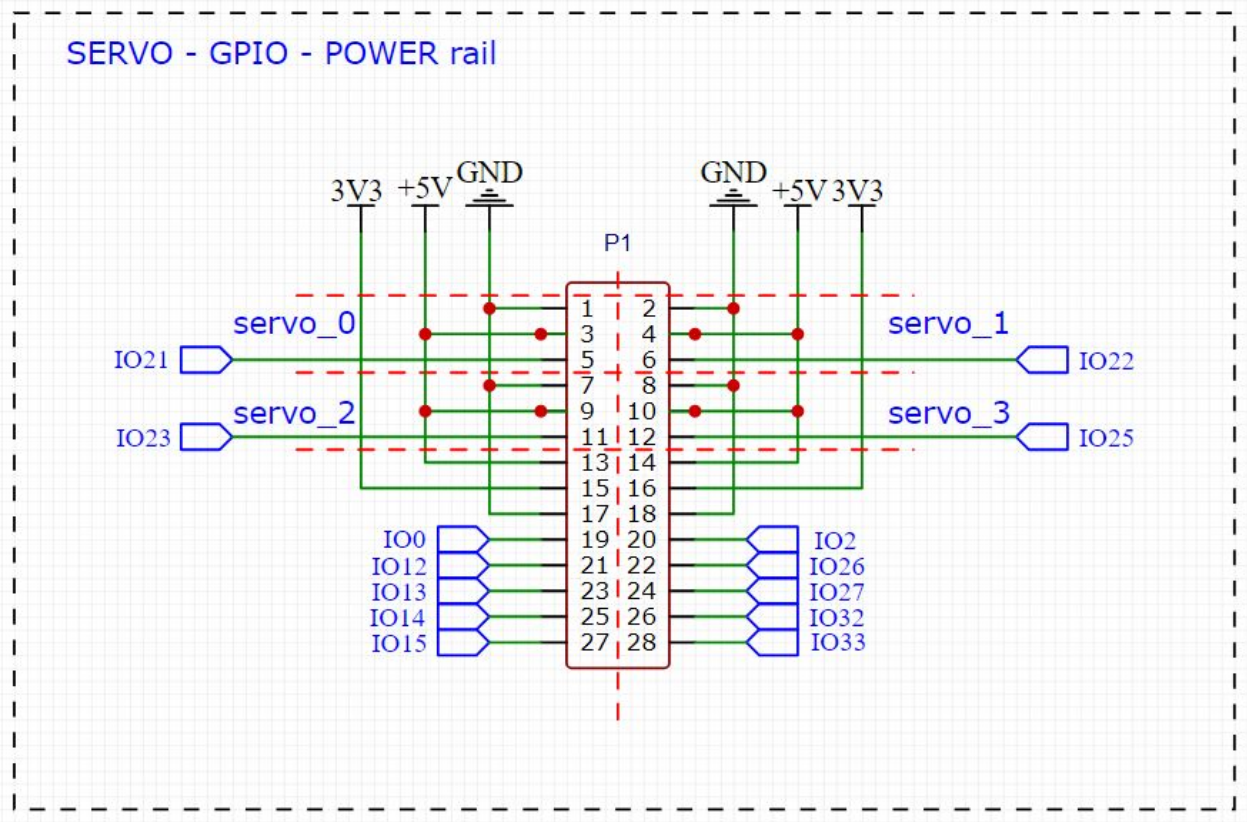
When the robot and the USB port are both connected simultaneously, the 3.3V rail will get its power from both the Lego robot as well as from the output of the LMS1117. Because there is only a small difference in these two voltages, there will be no harm done to the board or the lego robot.



For applications that need 5V, such as LED strips or servo motors, we provide 5V outputs on the 14x2 header. This 5V is drawn from hte USB connector when that is connected. For obtaining a stable 5V when the board is only connected to the lego robot, we provide a buck converter that can be added on the board. This buck converts the voltage supplied from the lego robot motor output (which is 8V when the motor power is set to 100%) to 5V in a very efficient way. When powering the board from the robot motor output simultaneously with the USB power, we get two 5V sources: one from the USB connector and one from the output of the buck converter. We estimate the because the voltages are almost the same, no harm willl be done to the board, the lego robot or the USB port.

1.5.4 GPIO header

The GPIO header provides access to 14 GPIO pins of the ESP32 and provides a number of 5V and 3.3V power pins.

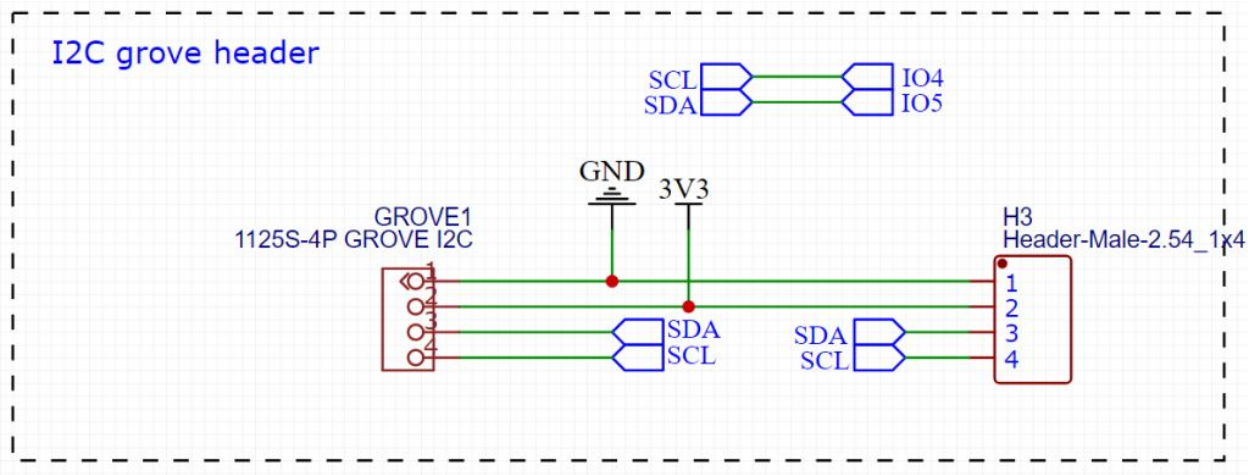


Whe grouped the pins in 3 different regions. The yellow region for connecting servo's or LED strips pwered with 5V (keep in mind that all the logic pins of the ESP32 are only 3.3V tollerant), the red region for power connections and the green region for access to the GPIO pins.

Note: The GPIO Pins are directly connected to the ESP32 and are not 5V tollerant, so use only 3.3V logic signals.

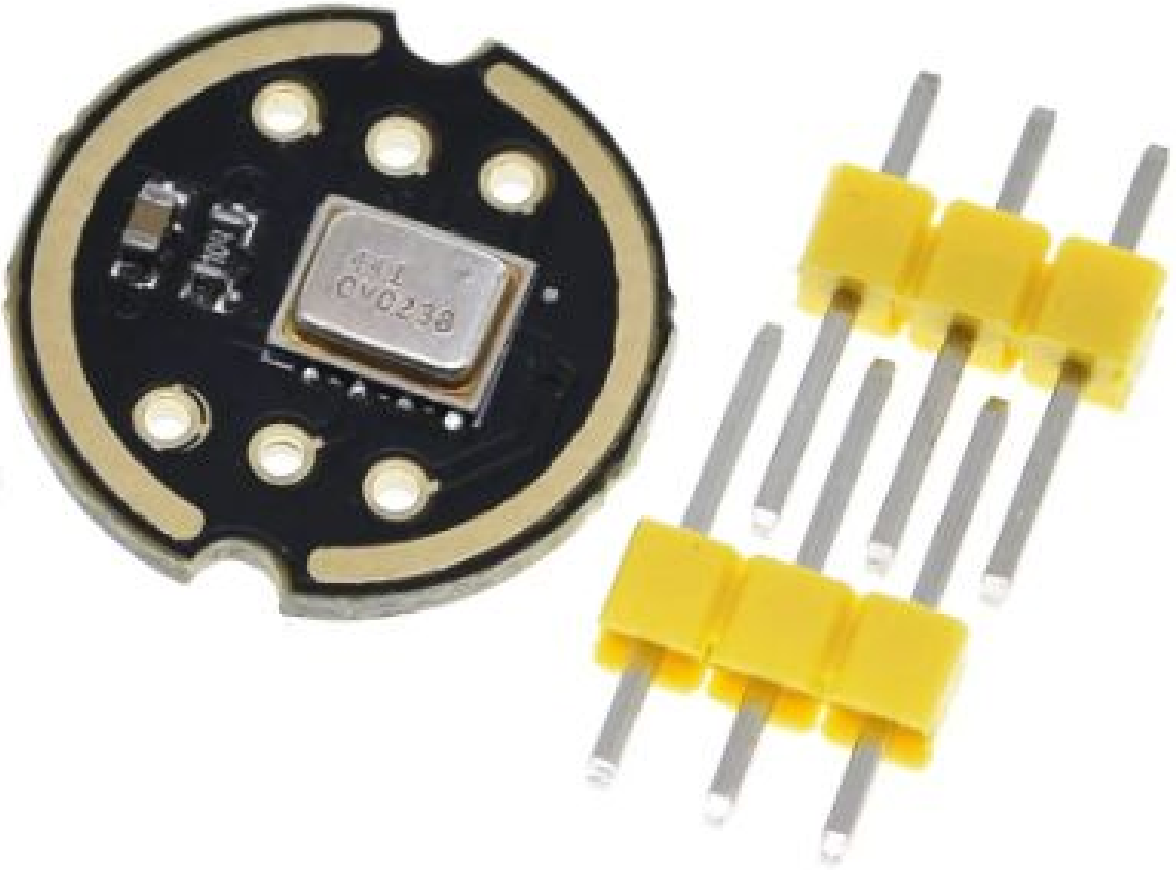
1.5.5 I2C Grove connector

We use the standard Grove connector that is used on many i2c sensors. M5Stack Units use them and can directly be connected to the board. Note that the power supply on this port is 3.3V.

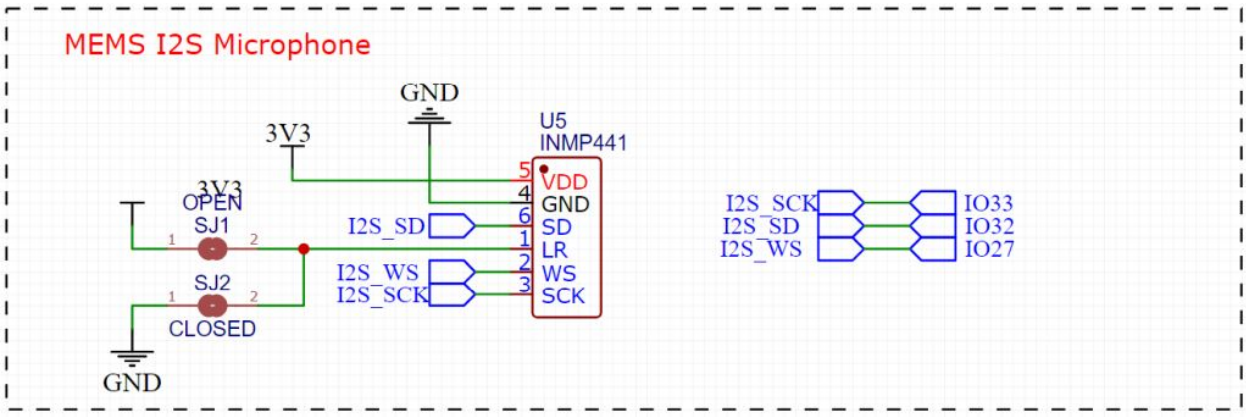


1.5.6 I2S Microphone

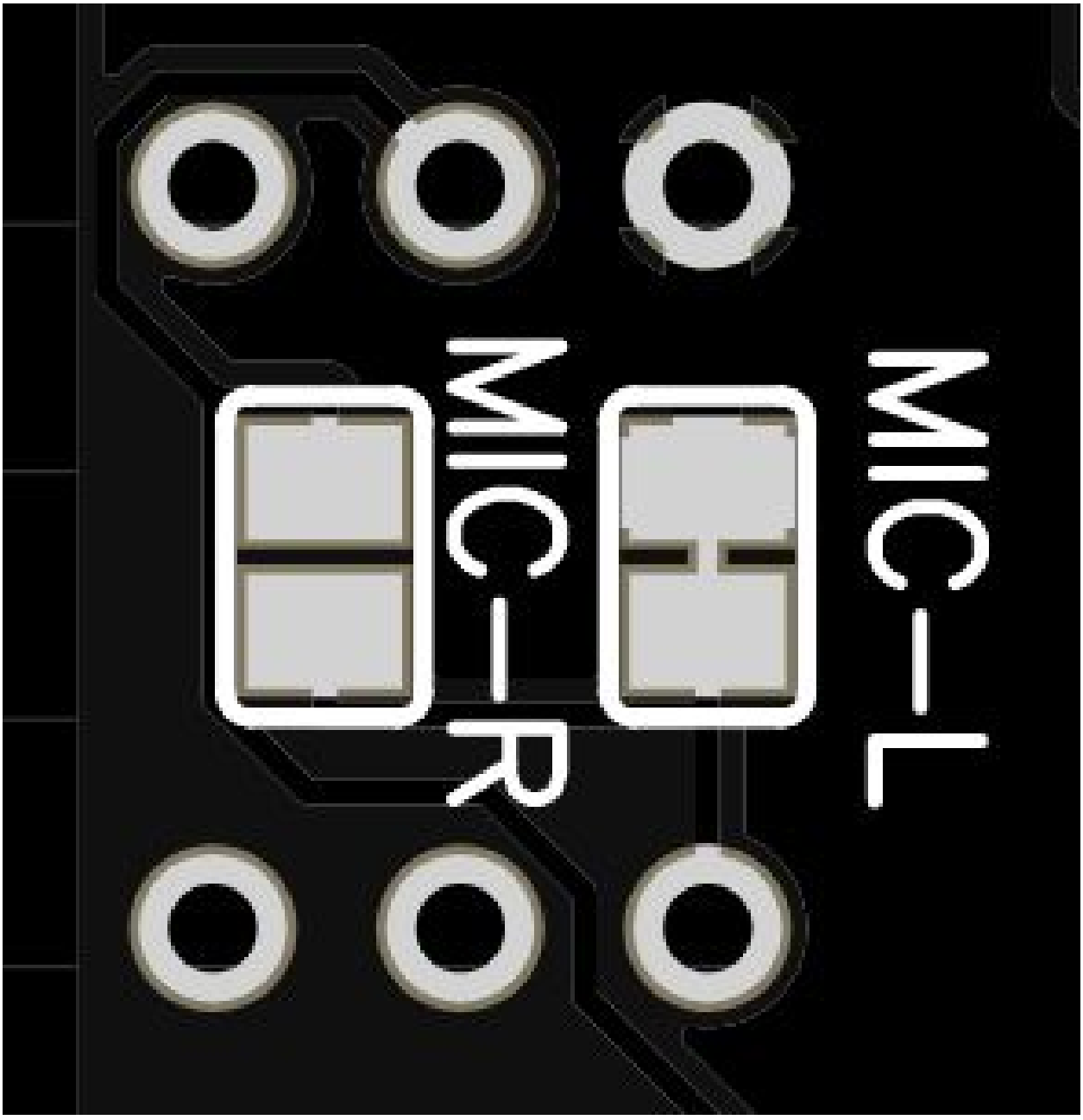
We provide two headers for connecting an INMP441 I2S microphone to the board. The esp-idf (Espressif IoT Development Framework) supports natevely the usage of I2S audio devices.



The I2S connections use GPIO 27, 32 and 33 for word clock line (I2S_WS), the data line (I2S_SD) and the bit clock line (I2S_SCK). Note that this microphone is a PCM version.

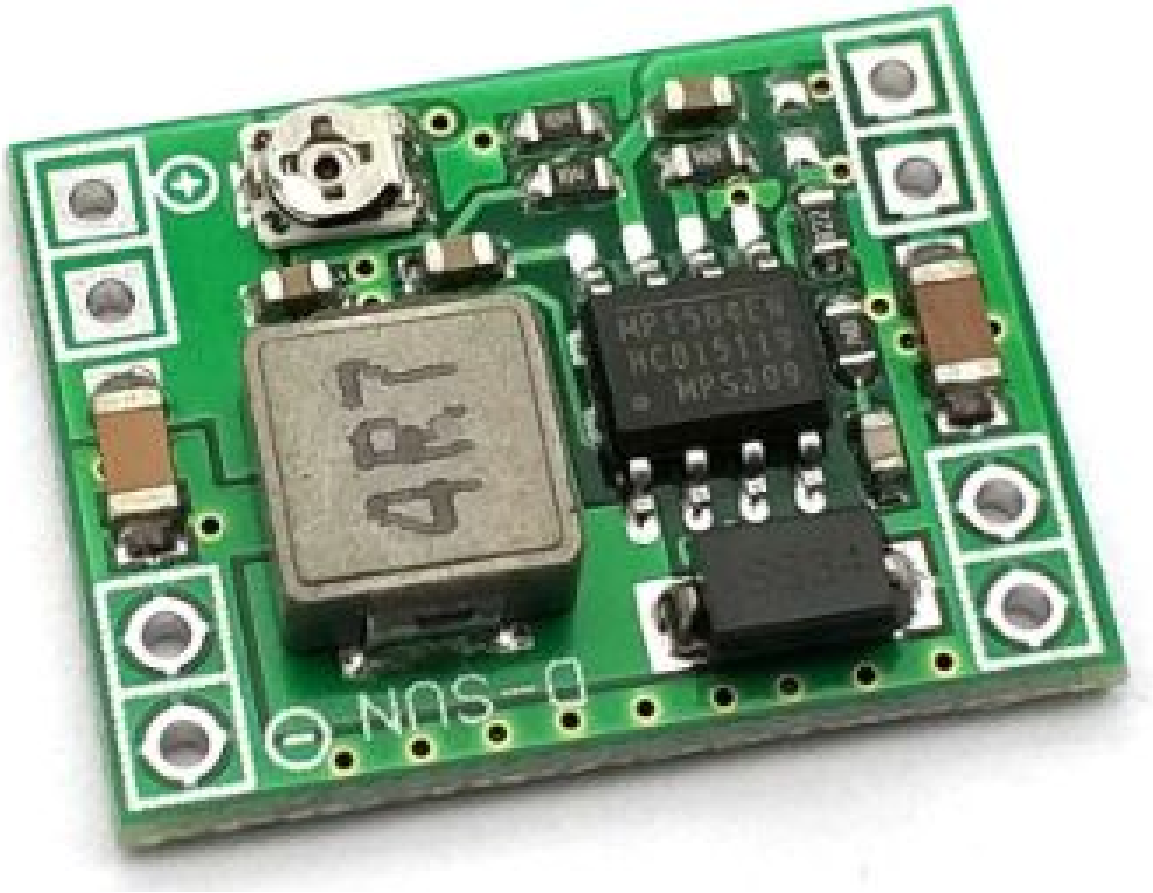


The solder pads on the back of the board can be used to select in which channel the digital output of the microphone is collected. By default the pad for selecting the left channel is shorted. Should you need to change the channel, than the trace between the MIC-L pads should be removed and the pads MIC-R should be shorted. Take care not to short both pads at the same time, as that leads to a power shortage.



1.5.7 Buck converter

We provide solder pads for a MP1584 buck converter. The buck converters input is connected to the M+ pin of the Lego robot's motor interface. When seeting the Lego motor to 100% power, the M+ pin will provide 8V DC. The buck converter brings this voltage down to a well regulated 5V DC.



The following code can be executed on the SPIKE Prime or Robot Inventor hub. Here we use Port A to connect the LMS-ESP32:

```
import hub
# set motor power to +100%, resulting in 8V on M+ and 0V on M-
hub.port.A.pwm(100)
```

1.6 Protocol

The communication between different UartRemote instances uses a self-described packet format. This section describes the packet format in detail.

1.6.1 Packet format

When a command with its accompanying values is transmitted over the Uart, the following packet format is used:

start	total len	command len	command	format len	format	data	end
<	ln	lc	cmd	lf	f	data	>

with

- `ln` the length of the total packet encoded as a single byte,
- `lc` the length of the command string <*cmd*> as a single byte,
- `cmd` the command specified as a string,
- `lf` the length of the format string
- `f` the Format encapsulation to pack the values; This can be `repr` for encapsulating arbitrary objects, `raw` for no encapsulation, or a Python struct format string.
- `data` a number of values encapsulated according to `f`.

The whole message is sandwiched between a *start* and *end* delimitter < and >.

When the method:

```
ur.send_command('test','repr',[1,2,3,4,5])
```

is used, the following packet will be transmitted over the line:

```
b'<b'\x1c\x04test\x04repr([1, 2, 3, 4, 5],)>'
```

Format Option 1: python struct.pack

This option interpretes the Format string *f* as the format string of the *struct.pack/unpack* method (see https://docs.python.org/3/library/struct.html), for example:

```
send_command('test_struct','3b3s1f',1,2,3,"ale",1.3).
```

This is the fastest method (1ms) but is limited to c-types, like int, unsigned int etc…

Format Option 2: repr/pickle

This uses the string representation of data, `repr()` to encode it. Then `eval()` is used on the receiving end.

```
ur.encode('test_command', 'repr', [[1,2],[3,4]])
```

will be encoded as:

```
b'%\x0ctest_command\x04repr([[1, 2], [3, 4]],)'
```

Here's the power of repr:

```
ur.encode('test_command','repr',[[1,2],[3,str],[len,True],[2+3]])
```

becomes:

```
b"W\x0ctest_command\x04repr([[1, 2], [3, <class 'str'>], [<built-in function len>, True],
↪ [5]],)"
```

This is slower (7ms) and incompatible with Arduino but it is more flexible.

Format Option 3: raw bytes

This is the fastest option of all, but you'll have to do your own decoding/encoding.

```
ur.encode('test_command','raw',b'abcd')
```

is encoded as:

```
b'\x15\x0ctest_command\x03rawabcd'
```

The format string

The type of <*data*> is given according to the [struct Format characters](https://docs.python.org/3/library/struct.html), of which the most commonly used are shown below:

Format character	type	number of bytes
b	*byte*	1
B	*unsigned byte*	1
i	*int*	4
I	*unsigned int*	4
f	*float*	4
d	*double*	8
s	*string[]*	one per char

example:

```
ur.call('mycommand','bb3sb',-3,-2,"aha",120)
```

Note that struct DOES NOT decode utf-8. You will always get a bytestring on the other side. It uses about 1ms to encode/decode.

Special format strings for other encoding types

- `repr`: use for a pickle-like serialized string encoding/decoding
- `raw` : skip encoding altogether and just pas one raw byte string.

example:

```
ur.call('mycommand','repr',[[12,34],[56,78]],"tekst",(1,2,3))
```

This will get all the python types across, but uses about 7ms to encode/decode.

```
ur.call('mycommand','raw',b"Raw byte string")
```

Note: If the encoder fails it resorts to raw bytes by default.

1.7 UartRemote for Arduino

This library is from a protocol point of view compatible with the MicroPython UartRemote library. It can be used in any Arduino project by adding this whole directory to the Arduino library directory. After importing the *UartRemote* library, an example can be selected form the examples in the Arduino IDE. Because this library is written in pure C++, it is faster by a factor of approximately 100 compared to the MicroPython implementation.

1.7.1 Differences compared to MicroPython implementation

Because C++ lacks the possibility to generate a function call with a variable number of parameters, a conversion function `unpack` was introduced.

A typical definition of a user defined function that will be called upon receiving a command with its accompanying parameters is shown below:

```
void led(Arguments args) {
  int r,g,b,n;
  unpack(args,&r,&g,&b,&n);
  Serial.printf("LED on: %d, %d, %d, %d\n", r, g, b, n);
  uartremote.send_command("ledack","B",0);
}
```

Here, the user function takes always a single parameters of type `Arguments`. To obtain the encoded variables, the `unpack` function is called.

The user defined function must always return an acknowledgement. In this case a dummy variable of type `byte` is returned.

The following example shows how to return one or more values back.

```
void add(Arguments args) {
  int a,b;
  unpack(args,&a,&b);
  Serial.printf("sum on: %d, %d\n", a, b);
  int c=a+b;
  uartremote.send_command("imuack","i",c);
}
```

In this example the sum of `a` and `b` is returned as an integer (`i`).

1.7.2 API

Arguments `UartRemote::receive_command`(char *cmd);

> Waits for an incomming command and return the received command in `cmd` and returns the format and buffer in the struct `Arguments`. The struct member `.error` has value 0 when no error occur.

void `UartRemote::send_command`(const char *cmd, const char *format, ...);

Send a command `cmd` over the UART where the vaiavle arguments are formatted according to the `format` string.

Arguments `UartRemote::call`(const char *cmd, const char *format, ...);

Calls remotely the function specified by `cmd` and returns the result in a struct of type `Argument`. The struct member `.error` equals 1 if an error occured.

int UartRemote::**receive_execute**()

Receives a command and executes the corresponding local function with the parameters as received from the command. This is a combination of `receive_command` and `command`. Returns 0 if no errors occurred.

void UartRemote::**add_command**(const char *cmd, void (*func)(Arguments));

Adds a function to the list of commands. In the example below, the function `tst` is added:

```
def tst(Arguments a) {
  ...
}

uartremote.add_command("tst",&tst);
```

void UartRemote::**command**(const char *cmd, Arguments rcvunpack);

Executes the function by looking up the `cmd` string in the internal `cmds` array that is filled using the `add_command` method and maps function names as sring to the actual function pointers.

int UartRemote::**available**();

Checks whether a character is available in the UART receive buffer. This can be used for a non-blocking implementation of UartRemote in your own loop.

void UartRemote::**flush**();

Flushes the UART receive buffer. This can be used if an error is suspected.

Private methods:

Arguments UartRemote::**pack**(unsigned char *buf, const char *format, ...);

Packs the variatic list of arguments according to the `format` string in the buffer `buf`.

unsigned char UartRemote::**readserial1**()

Reads a single byte from the UART receive buffer.

1.7.3 Arguments struct

We use a struct `Arguments` to store the format string together with the buffer with the unpacked data. The `friend unpack` function takes care for the proper unpacking of the buffer according to the format string. The `.error` struct member is used for passing error status back.

```
struct Arguments {
  void* buf;
  const char* fmt;
  int error;
  template<typename... Args> friend void unpack(const Arguments& a, Args... args) {
      struct_unpack(a.buf,a.fmt, args...);
  }
};
```

1.7.4 Examples

Below are some code snippets showing to use the Arduino side as a master and as a slave with its counter part written in Python.

```
char cmd[32]; // global temporary storage for command names

UartRemote uartremote;

void setup() {
  ...
  uartremote.add_command("led", &led);
  uartremote.add_command("add", &add);
  ...
}

void loop() {
  int error = uartremote.receive_execute();
  if (error==1) {
    printf("error in receiving command\n");
  }
}
```

With on the Python side the following code:

```
from uartremote import *
from utime import sleep_ms
ur=UartRemote()

ur.flush()

while True:
  ack,s=ur.call('add','2i',1,2)
  print("sum = ",s)
  sleep_ms(500)
  ur.call('led','4i',1,2,3,4)
  sleep_ms(500)
```

The other way round it would look like:

```
char cmd[32]; // global temporary storage for command names

UartRemote uartremote;

void setup() {
  ...
}
int i=0;
int s=0;

void loop() {
  i+=1;
  i%=100;
  args=uartremote.call("led","4i",i+1,i+2,i+3,i+4);
```

(continues on next page)

(continued from previous page)

```
  if (args.error==0) {
    printf("received ledack: %s\n",cmd);
  } else { printf("error from call led\n");}
  delay(1000);
  args=uartremote.call("add","2i",i+1,i+2);
  if (args.error==0) {
    unpack(args,&s);
    printf("Received sumack: %s, sum=%d\n",cmd,s);
  } else { printf("error from call sum\n");}
 delay(1000);
}
```

On the Pyhton side we have the following code:

```
from uartremote import *
ur=UartRemote()

  def led(n,r,g,b):
  print('led',n,r,g,b)

def add(a,b):
  print('adding',a,b)
  return(a+b)

ur.add_command(led)
ur.add_command(add,'i')

ur.loop()
```

1.7.5 Struct library

We use the `struct` library with implemenations of `pack` and `unpack` supporting Python compatible format strings. The code can be found on github.com/svperbeast/struct.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THREE

STATUS

INDEX