README.md

CSL Serial

If you have developed for the Arduino before, you most likely have used the Serial class to print messages and to send commands for your computer to the Arduino.

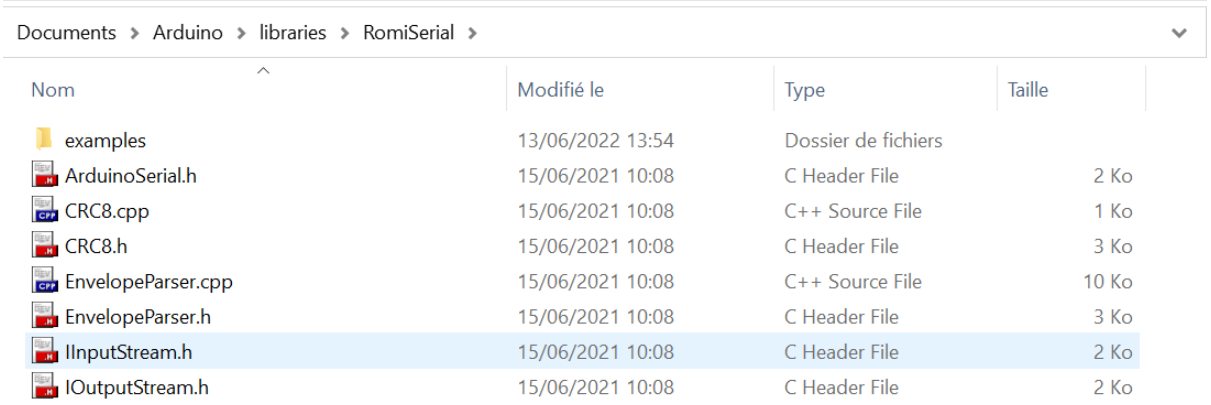
The Serial link has some caveats, though, and Romi Serial tries to address some of those. We will go into detail further below. First, we show some examples on how to use it in your projects. We will show the classic "Blink" and AnalogReadSerial" examples. In the first example we will control the LED from Python and C++. In the second example, we will se how to get data off the Arduino, in Python and C++.

You can find the original code for Blink and AnalogReadSerial online and also in the Arduino IDE in the menu `File > Examples > 01.Basics`.

Install the library

```
git clone XXXXXXXX
cd CSL-serial
python setup.py develop
```

1. Install Arduino IDE
2. If you have never used an Arduino you can start with the tutorial.
3. Find the location where Arduino fetches libraries (usually "Documents/Arduino /libraries on Windows"). Add "libraries" folder if it doesn't exist.
4. In this location, copy the RomiSerial folder of the repository.



To make sure the installation worked, go to (docs/blink)[docs/blink] and open the .ino file. Verify it and upload it on the Arduino by selecting the correct card and com port. Then, run `python blink.py --device=COMXX` (replace XX by the correct port number). You should see the LED in front of port 13 blink every second.

How it works

Blink

Using Python

The complete Python code look as follows:

```
import sys
sys.path.append('../python')

import time
import argparse
from CSLserial import ControlSerial

remoteDevice = False

def setup(device):
    global remoteDevice
    remoteDevice = ControlSerial(device)


def loop():
    global remoteDevice
    remoteDevice.execute('L', 1)
    time.sleep(1)
    remoteDevice.execute('L', 0)
    time.sleep(1)


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--device', type=str, nargs='?', default="COM5",
                        help='The serial device to connect to')
    args = parser.parse_args()

    setup(args.device)
    while True:
        loop()
```

The associated code for the Arduino is as follows:

```
#include <ArduinoSerial.h>
#include <RomiSerial.h>

using namespace romiserial;

void handle_led(IRomiSerial *romiSerial, int16_t *args, const char *string_arg

const static MessageHandler handlers[] = {
        { 'L', 1, false, handle_led },
};

ArduinoSerial serial(Serial);
RomiSerial romiSerial(serial, serial, handlers, sizeof(handlers) / sizeof(Messa
```

```
void setup() {
    pinMode(LED_BUILTIN, OUTPUT);
    Serial.begin(115200);
}

void loop() {
    romiSerial.handle_input();
}

void handle_led(IRomiSerial *romiSerial, int16_t *args, const char *string_arg
{
    if (args[0] == 0) {
        digitalWrite(LED_BUILTIN, LOW);
    } else {
        digitalWrite(LED_BUILTIN, HIGH);
    }
    romiSerial->send_ok();
}
```

Code explanation

We will go over the code above, step by step.

The first two lines of the Python code make sure that you can run the example code from within the docs directory. If you installed the CSLserial.py file in your code directory, you will not need this.

```
import sys
sys.path.append('../python')
```

We will use a small utility class, called `ControlSerial`. It wraps the lower-level functions of sending and receiving commands, and handling errors.

```
from CSLserial import ControlSerial
```

We will create one instance of a ControlSerial that we will store in a global variable for simplicity.

```
remoteDevice = None
```

We structured the code to mimick the original Arduino example and wrote two functions, `setup` and `loop`. The `setup` function initializes the remote device. It opens a serial connection to the Arduino to enable the sending of the commands. The function takes as a single argument the name of serial device that it should connect to. You will be able to specify this name on the command line, as we will show below.

```
def setup(device):
    global remoteDevice
    remoteDevice = ControlSerial(device)
```

If you want more debugging information, you can call `remoteDevice.set_debug(True)` in the set-up.

The loop function turns the LED on and off by sending a command to the Arduino. Commands consist of a single character: a lowercase or uppercase ASCII character, or a digit. If the command requires arguments, they can be given as additional parameters to the execute function. In the example below, we pass one argument: whether the LED should be off (the argument is 0) or whether the LED should be on (the argument is 1):

```
def loop():
    global remoteDevice
    remoteDevice.execute('L', 1)
    time.sleep(1)
    remoteDevice.execute('L', 0)
    time.sleep(1)
```

In this example we choose `L` as the opcode of the command. You are free to choose any character but it should correspond to the same character used in your code on the Arduino side (see below).

There are some constraints on the arguments you can pass:

- The number of arguments should be less or equal to 12.
- The arguments should be integers with a value between -32768 and 32767.
- It is possible to one, and only one, string as an argument.

Also, the total length of the message cannot exceed 58 bytes. This is due to the limited size of the buffer on the Arduino.

The number of expected arguments for each opcode will be coded also on the Arduino side. More on that below.

The Arduino code

Let's have a look at the code for the Arduino. To begin with, you have to include the required headers. The RomiSerial classes lives in a namespace of their own, `romiserial`. In the code below we added a `using namespace` statement to simplify the example:

```
#include <ArduinoSerial.h>
#include <RomiSerial.h>

using namespace romiserial;
```

The following line wraps the standard Serial object in one of our classes. The reason is technical. It is so that we can instantiate our classes before the Serial object has completed its initialisation.

```
ArduinoSerial serial(Serial);
```

Here is the most important part of the code. The following code defines all the functions that will handle the commands sent by the user. The first line (`handle_led`) is simply a function declaration so that we can use the name in the following table, `handlers[]`.

This table lists all our commands and the functions that will handle them. Each command definition consists of:

- The opcode. Valid opcodes are (a-z, A-Z, 0-9, ?)
- The number of arguments that are expected
- Whether one of the arguments is a string.
- The function that will handle the commands.

Finally, we create an instance of `RomiSerial`. It takes four arguments:

- The serial object used to read the input.
- The serial object used to write the output (can be the same as the input above).
- The list of command definitions.
- The number of command definitions.

```
void handle_led(IRomiSerial *romiSerial, int16_t *args, const char *string_arg

const static MessageHandler handlers[] = {
        { 'L', 1, false, handle_led },
```

```
};

RomiSerial romiSerial(serial, serial, handlers, sizeof(handlers) / sizeof(Messa
```

This example only shows one command. To add more, simple add more commands, as much as you need. Below the command `x` takes three arguments and will be handled by the function `handle_another_command`:

```
void handle_led(IRomiSerial *romiSerial, int16_t *args, const char *string_arg
void handle_another_command(IRomiSerial *romiSerial, int16_t *args, const char

const static MessageHandler handlers[] = {
        { 'L', 1, false, handle_led },
        { 'x', 3, false, handle_another_command },
};
```

Once this has been set up, you should call the method `romiSerial.handle_input()` regularly. The best way to do this is to put it in Arduino's `loop()` function. However, if you have some functions in your code that require a lot of time to complete, it may be worth considering calling `handle_input` in those functions as well, to assure that the Arduino remains responsive.

```
void loop() {
    romiSerial.handle_input();
}
```

Finally, you have to define the function handlers that we declared previously. In our example, the `handle_led` function expects one integer argument, and based on its value, it turns the LED on or off:

```
void handle_led(IRomiSerial *romiSerial, int16_t *args, const char *string_arg
{
    if (args[0] == 0) {
        digitalWrite(LED_BUILTIN, LOW);
    } else {
        digitalWrite(LED_BUILTIN, HIGH);
    }
    romiSerial->send_ok();
}
```

When all went well, the handler should call `send_ok()`. If an error occured, you should call `send_error(error_number, "Error message")`. The error number is any positive integer of your choosing. The code on the Python-side will receive it. Similarly for the error message.

That's it!

Running the example

To run the example, you first have to upload the Arduino code using the Arduino IDE. The following page explains in detail how to do this.

Then, you execute the `blink.py` Python script that you can find in this docs directory. If you're not sure how to do that, there are many tutorials on the web that explain this. Two examples are this page and this page.

In the Linux console you can type:

```
$ python3 blink.py --device /dev/ttyACM0
```

On Windows, using the MS-DOS `cmd` application:

```
C:> python3 blink.py --device COM5
```

The port that is given as argument (/dev/ttyACM0, COM5) corresponds to the serial device to which the Arduino is attached (more here).

Using C++

The C++ code is also quite straightforward. Here is the full listing:

```
#include <memory>
#include <unistd.h>
#include <RomiSerialClient.h>
#include <RSerial.h>
#include <Console.h>

using namespace romiserial;

void delay(size_t milliseconds)
{
        usleep(milliseconds * 1000);
}

int main(int argc, char **argv)
{
        if (argc < 2) {
                throw std::runtime_error("Usage: blink <serial-device>");
        }

        std::string device = argv[1];
```

```
        auto log = std::make_shared<Console>();
        auto serial = std::make_shared<RSerial>(device, 115200, true, log);

        RomiSerialClient romiClient(serial, serial, log, 0, "blink");
        nlohmann::json response;

        while (true) {
                romiClient.send("L[1]", response);
                delay(1000);
                romiClient.send("L[0]", response);
                delay(1000);
        }
}
```

This docs directory contains a minimal Makefile to compile the code. If the compilation is successful, there will be two executable binaries, `blink_app` and `analogread_app` (see below):

```
$ make
...
$ ./blink_app
```

AnalogRead

The Arduino code

The Blink example showed how to set the LED lights, but it didn't show how you can get data back from the Arduino. If you have some sensor attached to the Arduino, you may want to read the value of the sensor from your Python application. The rewrite of the AnalogRead example, below, will show you how to do it.

The original Arduino code looks somethings like this:

```
void setup() {
  Serial.begin(115200);
}

void loop() {
  int sensorValue = analogRead(A0);
  Serial.println(sensorValue);
  delay(1);
}
```

We are going to replace this with the following version, based on RomiSerial:

```
#include <ArduinoSerial.h>
#include <RomiSerial.h>

using namespace romiserial;

ArduinoSerial serial(Serial);

void read_sensor(IRomiSerial *romiSerial, int16_t *args, const char *string_ar

const static MessageHandler handlers[] = {
        { 'A', 0, false, read_sensor },
};

RomiSerial romiSerial(serial, serial, handlers, sizeof(handlers) / sizeof(Mess

void setup() {
    Serial.begin(115200);
}

void loop() {
    romiSerial.handle_input();
}

void read_sensor(IRomiSerial *romiSerial, int16_t *args, const char *string_ar
{
    char reply[16];
    int sensorValue = analogRead(A0);
    snprintf(reply, sizeof(reply), "[0,%d]", sensorValue);
    romiSerial->send(reply);
}
```

Most of the code above is similar to the Blink example discussed earlier. The command handler is now called `read_sensor`. As you can see from the command definition, the handler does not take any arguments (second value is zero):

```
const static MessageHandler handlers[] = {
        { 'A', 0, false, read_sensor },
};
```

The main difference is in the `read_sensor` handler. Instead of calling `romiSerial->send_ok()` we send back a string. The string should contain an array ([]) with a list of values. The first value should be zero because it indicates whether an error occured. Following that, you can insert as many values as you want, including strings in double quotes.

```
void read_sensor(IRomiSerial *romiSerial, int16_t *args, const char *string_ar
```

```
{
    char reply[16];
    int sensorValue = analogRead(A0);
    snprintf(reply, sizeof(reply), "[0,%d]", sensorValue);
    romiSerial->send(reply);
}
```

The Python code

On the Python side, there is not so much change neither. Here's the full listing:

```
import sys
sys.path.append('../python')

import time
import argparse
from CSLserial import ControlSerial

remoteDevice = None

def setup(device):
    global remoteDevice
    remoteDevice = ControlSerial(device)


def loop():
    global remoteDevice
    response = remoteDevice.execute('A')
    print(f'Sensor value: {response[1]}')
    time.sleep(1)


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--device', type=str, nargs='?', default="COM5",
                        help='The serial device to connect to')
    args = parser.parse_args()

    setup(args.device)
    while True:
        loop()
```

The main change is that we store the result of the call `remoteDevice.execute` into the variable `reponse`:

```
response = remoteDevice.execute('A')
print(f'Sensor value: {response[1]}')
```

`response` is an array containing the values that were sent back by the Arduino, starting with the zero error code and followed by the value of the sensor. If you print `response` you should look something like `[0,123]` where 123 (or `response[1]`) is the value of the sensor you are looking for.

Using C++

Below is the C++ version. The code is similar to the Blink example. The response object is a full-featured JSON object, in this case an array. Similar to the Python implementation, the first element of the array represents the error code, and the second element the sensor value returned by the Arduino.

```
#include <memory>
#include <iostream>
#include <unistd.h>
#include <RomiSerialClient.h>
#include <RSerial.h>
#include <Console.h>

using namespace romiserial;

void delay(size_t milliseconds)
{
        usleep(milliseconds * 1000);
}

int main(int argc, char **argv)
{
        if (argc < 2) {
                throw std::runtime_error("Usage: analogread_app <serial-device
        }

        std::string device = argv[1];
        auto log = std::make_shared<Console>();
        auto serial = std::make_shared<RSerial>(device, 115200, true, log);

        RomiSerialClient romiClient(serial, serial, log, 0, "analogread");
        nlohmann::json response;

        while (true) {
                romiClient.send("A", response);
                std::cout << "Sensor value: " << response[1] << std::endl;
                delay(1000);
        }
```

```
}
```

More in depth

The Romi Serial library helps improve the reliability of the communication between a computer and an Arduino over the serial connection.

It was developed for the Romi Rover because it communicates with several microcontrollers over a serial bus. The proposed library should help improve the reliability of these serial communications. It aims to address the following issues:

- When the serial connection is used without proper synchronisation, the Arduino Uno may loose data without a warning because its internal serial buffer overflows. This issue is addressed by limiting the maximum size of a message to 64 bytes and by using a synchronous request-response pattern.
- Without a well-defined timing between the request and the response, the host may wait indefinitely for a response in case the controller fails for some reason. The interaction therefore defines a maximum timeout for both the request and response handling.
- It provides a better error handling:
    - The error code is now explicitly part of the response.
    - Each request-response pair has an ID to assure that a reponse corresponds to the request that was sent.
    - A CRC code is appended to the messages to assure that the messages are not corrupted or incomplete.
- It distinguish more clearly between the following three layers:
    - The serial link to read and write bytes
    - The protocol layer to send requests and read responses
    - The application layer that defines the semantics of the commands

This proposition decribes the protocol layer that should be shared by all devices for which we control the firmware. A clearly defined protocol should simplify the documentation and debugging. However, this document does not specify the opcodes and expected arguments of the different firmwares. These opcodes are part of the application layer. Some normalisation of the opcodes could be the subject of a later specification.

Unless specified otherwise, the serial connections should be set to: 115200 baudrate, 8 data bits, no parity, and 1 stop bit (115200 8N1).

Message formats

The exchange always follows a request-response pattern. The formats of the request and the response are detailed below. However, for debugging purposes, the controller may also send log messages asynchronously. These should be handle by the host, and the log text treated in whatever way that is most appropriate, for example, writing them to a log file.

Log messages

At any time, the controller may send log messages in the following form:

```
'#!' TEXT ':xxxx\r\n'
```

The TEXT is a string of variable length.

Requests

A request is a string that consists, in summary, of a one-character opcode followed by zero or more arguments, an ID and a cyclic redundancy check (CRC). The precise format is as follows:

```
'#' <opcode> ':' <id> <crc> '\r\n'

'#' <opcode> '[' <arg1>, <arg2>, ... ']' ':' <id> <crc> '\r\n'
```

- the opcode consists of a single character from the following set: {a-z, A-Z, 0-9, ?}
- arg1, arg2 are integer numbers in the range [-32768,32767] (signed 16 bits), or a string with a maximum length of 32 characters (see more below).
- id is an integer number in the range [0,255]. It is encoded as a two-character hexadecimal.
- crc is the 8-bit CRC code of the request, encoded as a two-character hexadecimal
- the carriage return and line feed characters `\r\n` signal the end of the message.

The hashtag indicates the start of a message. For this reason, it should be avoided in strings passed as argument. Similarly, the carriage return and line feed characters (CRLF) indicate the end of a message. They should be avoided in strings.

The maximum number of integer arguments that can be given is 12.

A string argument should be surrounded by double-quotes ('"'). There can only be one string per request. The maximum length of the string is 32 characters.

The CRC code is computed on the string representation of the request, starting with the hashtag until the ID (included). See below for an implementation of thr CRC-8 algorithm.

Requests sents from software code should always add a valid ID and CRC. The 8-bit CRC code is formatted as a hexadecimal string of two characters. It must use the lowercase letters a-f. Example a CRC with a value of 0 is formatted as "00"; a CRC of 255 results in the string "ff".

The total length of the request, including the hashtag, ID, the CRC, and CRLF, must not be longer than 64 bytes. This is the size of the internal buffer used by the Arduino Uno.

Manual input

The trailing colon, ID and CRC code are obligatory. To simplify sending commands manually from a terminal it is possible to replace the four characters of yje ID and the CRC by `x` characters, as follows.

```
'#' COMMAND ':xxxx\r\n'
```

For example:

```
#e[0]:xxxx\r\n
```

You should verify that your terminal ends lines with CR + LF when you press enter. In the Arduino terminal window, select the "Both NL & CR" line ending in the pop-up menu at the bottom of the window. In the `picocom` terminal, use the `--omap crcrlf` command line option. For other terminal applications, check their documentation.

Response

The reponse is formatted as follows:

```
'#' <opcode> '[' 0, <value1>, <value2> ... ']' ':' <id> <crc> '\r\n'

'#' <opcode> '[' errorcode, <message> ']' ':' <id> <crc> '\r\n'
```

- the opcode consists of a single character. It mirrors the opcode of the request.
- value1, value2 are number or strings formatted compatible with the JSON standard.

- the errorcode is an integer (more below).
- message is a user-readable string in double-quotes (optional).
- id is an hexadecimal number in the range [0,255]. It mirrors the id of the request.
- crc is the CRC code of the textual representation of the response up to and including the ID.
- the firmware will finish the message with a carriage return and a linefeed. Both are sent because it works more nicely with terminal applications.

The reponse of the controller will also start with a hashtag. The controller will then repeat the opcode of the request. The first argument is always an error code. An error code of zero means that the request was successful. If the error is not 0 than the second value may be an additional short, user-readable message.

The ID mirrors the id of the original request. If none was given, it will be zero (in case of manual commands from a terminal). The controller will always return an ID and CRC code. The CRC code is computed on the complete response, starting with the hashtag until and including the ID.

Examples

Let's look at a couple of simple examples. The first example is a request with the opcode 'e' but without any arguments, ID, or CRC. The string ot the request is as follows:

```
#e\r\n
```

If all goes well (error code is 0) and controller does not return any extra values, then the response by the controller is:

```
#e[0]:0092\r\n
```

The controller has returned an ID of zero because none was given in the request. The CRC-8 value of the string "#e[0]:00" is 0x92, so "92" is appended to the response.

In the next example, the host sends a request with an ID of 123 (0x7b in hexaecimal). The host must add a CRC code, too. The full request string looks like the string below. The CRC-8 of '#e:7b' is 0x04:

```
#e:7b04\r\n
```

The response of the controller will now also include the ID:

```
#e[0]:7b40\r\n
```

(crc8 of #e[0]:7b is 0x40)

Here's an example in which the host sends a request with an additional arguments (the ID is still 123):

```
#M[16,"Shutdown"]:7bba\r\n
```

(crc8 of '#M[16,"Shutdown"]:7b' is 0xba)

We will assume that the request can't be completed and that the controller returns an error message. The first value that is returned is the error code (1 in this case) and the second an optinal error message:

```
#M[1,"Out of boundary"]:7ba7\r\n
```

(the CRC-8 of '#M[1,"Out of boundary"]:7b' is 0xa7.)

Host: Time outs and message IDs

All communication is synchronous. When the host sends a request, the controller must send a response within less than one second. The host should therefore set a time-out when reading the response. This avoids the risk that the host may get stuck indefinitely while waiting for a response.

Note: The controller must reply in less than one second. However, the host should use a time-out value that is a slightly longer than one second.

The host may receive log messages while waiting for the reponse. In that case, the host must treat the log message and then reattempt to read the response. This next attempt should again use a timeout of one second or more.

The request IDs must be incremented by one after each cycle. When the ID reaches 255, the next ID start again at 0.

In case the controller *does* miss the one second deadline (nothing is perfect): in that case controller may send the response *after* the host timed out. This delayed response will then be read by the host in a subsequent request-response cycle. The host must therefore be prepared to handle this. It should always check the ID of the response. If the ID of the request is different than the ID of the request, the host should ignore the response and try reading the next response message with a timeout of 1 second.

A call to the host's request/reponse function should never take more than 2 seconds to complete. The host should therefore keep track of the total time spent even when at re-attempts to read a response after receiving a log message or a stale message.

Controller

The controller must respond to requests within one second. Vice versa, the host should assure that a request is completely sent within less than a second. If the host sends the request too slowly or if data is lost on the connection and the complete message takes more than one second to parse, the controller must do the following: ignore the partially read request, return a time-out error (see below), and wait for the beginning of a new request (initiated by the hashtag).

The error codes that are returned by the controller fall in two categories. Errors raised by the protocol layer and errors returned by the application. The protocol layer will only use negative error codes. They are discussed below. The application can freely use positive error codes.

Error codes

The complete list of error codes can be found in the RomiSerialErrors.h file.

Notes on the CRC-8

There exists several varieties of the CRC8 algorithm (see this https://reveng.sourceforge.io/crc-catalogue/1-15.htm#crc.cat.crc-8). Romi Serial uses the CRC-8/SMBUS, or "plain" CRC-8 implementation. In the following online CRC calculator, https://crccalc.com/, it's the first one in the table (CRC-8).

Please check the code in the CRC8.h file.