simpleRPC Documentation

Release latest

Apr 20, 2019

Contents:

This library provides a simple way to export Arduino functions as remote procedure calls. The exported method definitions are communicated to the host, which is then able to generate an API interface.

Features:

- For each method, only one line of code is needed for exporting.
- Automatic parameter- and return type inference.
- Support for all native C types and strings.
- Support for arbitrary functions and class methods.
- Optional function and parameter naming and documentation.
- Support for PROGMEM's `F()` macro to reduce memory footprint.
- Support for compound data structures like Tuples, Objects (nested Tuples), Vectors and arbitrary combinations of these.

The Arduino library is independent of any host implementation, we provide a Python API client library as a reference implementation.

Please see ReadTheDocs for the latest documentation.

CHAPTER 1

Quick start

Export any function e.g., `digitalRead()` and `digitalWrite()` using the `interface()` function.

```
#include <simpleRPC.h>

void setup(void) {
  Serial.begin(9600);
}

void loop(void) {
  interface(digitalRead, "", digitalWrite, "");
}
```

These functions are now available on the host under names `method2()` and `method3()`.

The documentation string can be used to name and describe the method.

```
interface(
  digitalRead,
    "digital_read: Read digital pin. @pin: Pin number. @return: Pin value.",
  digitalWrite,
    "digital_write: Write to a digital pin. @pin: Pin number. @value: Pin value.");
```

This is reflected on the host, where the methods are now named `digital_read()` and `digital_write()` and where the provided API documentation is also available. In our client reference implementation documentation, we show an example on how this works.

CHAPTER 2

Further reading

Please read *Library* for more information about exporting normal functions, class member functions and documentation conventions.

If you want to create your own host library implementation for other programming languages, the section *Protocol* should help you on your way.

2.1 Introduction

A remote procedure call to an Arduino device is a common way to read sensor values or to send control signals. This library provides a simple way to export any Arduino function, including API documentation.

2.1.1 Motivation

Suppose we have an number of functions that we want to export as remote procedure calls.

```
int testInt(void) {
  return 1;
}

float testFloat(void) {
  return 1.6180339887;
}

int add(int a, int b) {
  return a + b;
}
```

A common way of making functions available is to map each of the functions to an unique value. The Arduino reads one byte from the serial device and it uses this to call the appropriated function.

If a function takes parameters, their values need to be read from the serial device before calling the function. Any return value needs to be written to the serial device after calling the function.

A typical implementation of such an approach is shown below.

```
void loop(void) {
  int iValue, iParamA, iParamB;
  float fValue;

  if (Serial.available()) {
    switch (Serial.read()) {
      case 0x00:
        iValue = testInt();
        Serial.write((byte *)&iValue, 2);
        break;
      case 0x01:
        fValue = testFloat();
        Serial.write((byte *)&fValue, 4);
        break;
      case 0x02:
        Serial.readBytes((char *)&iParamA, 2);
        Serial.readBytes((char *)&iParamB, 2);
        iValue = add(iParamA, iParamB);
        Serial.write((byte *)&iValue, 2);
        break;
    }
  }
}
```

In this implementation, the methods `Serial.write()` and `Serial.readBytes()` are used to encode and decode values.

On the host, the parameter values need to be packed before sending them to the Arduino, Any return value needs to be unpacked. In the following example, we assume that a serial connection is made using the pySerial library. The functions `pack` and `unpack` are provided by the struct library.

```
# Call the testInt() function.
connection.write(pack('B', 0x00))
print(unpack('<h', connection.read(2))[0])

# Call the testFloat() function.
connection.write(pack('B', 0x01))
print(unpack('<f', connection.read(4))[0])

# Call the add() function.
connection.write(pack('B', 0x02))
connection.write(pack('<h', 1))
connection.write(pack('<h', 2))
print(unpack('<h', connection.read(2))[0])
```

An implementation like the one described above uses very little bandwidth and does not require any heavy external libraries on the Arduino. The downsides of such an approach are clear from the example:

- Quite a bit of boilerplate code is needed.
- Changes have to be made on both the device and the host, keeping the implementations in sync may become difficult.
- A lot of low-level knowledge of the device methods and their types is required.

This is why we developed the simpleRPC library, like the implementation above, it only communicates values but has none of the downsides of an *ad hoc* protocol.

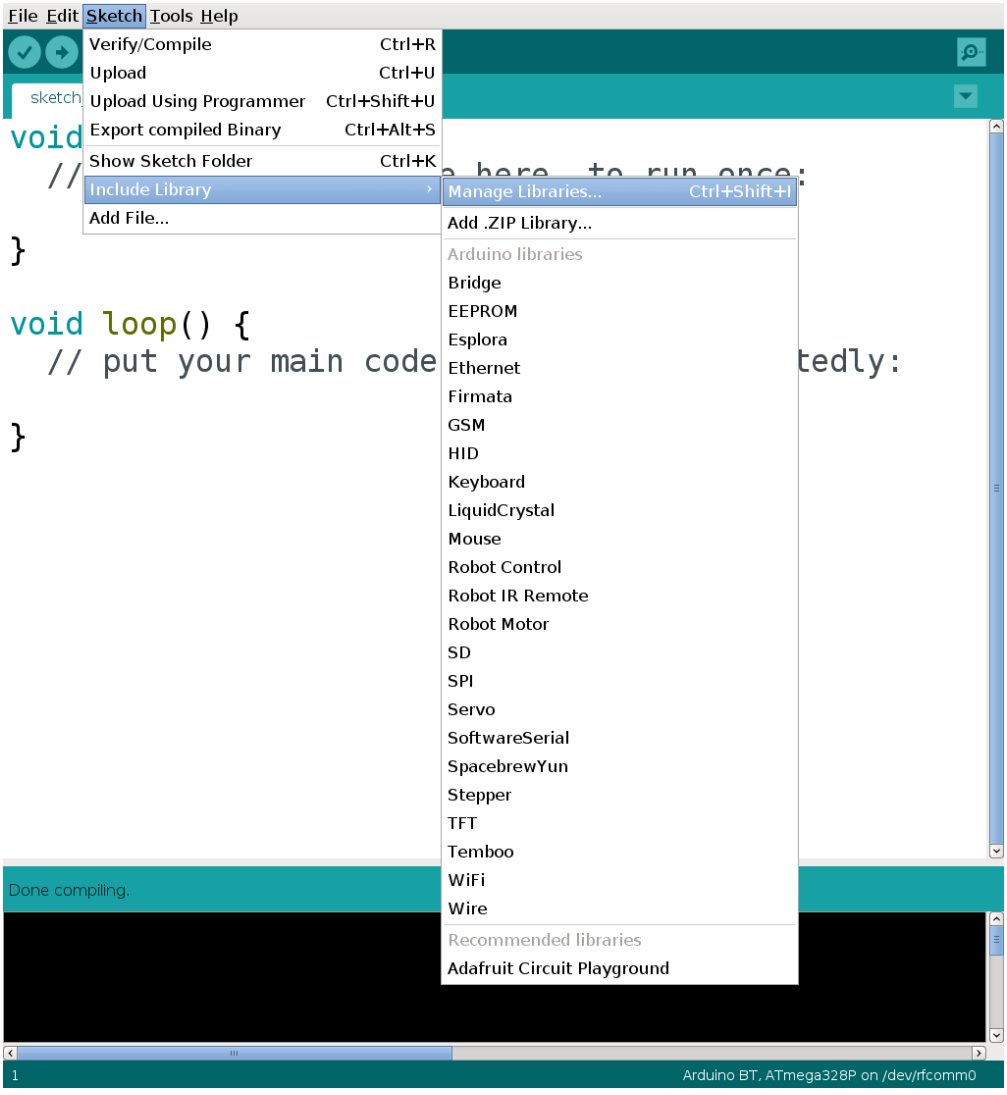
2.2 Installation

In this section we cover retrieval of the latest release or development version of the code and subsequent installation for an Arduino device.

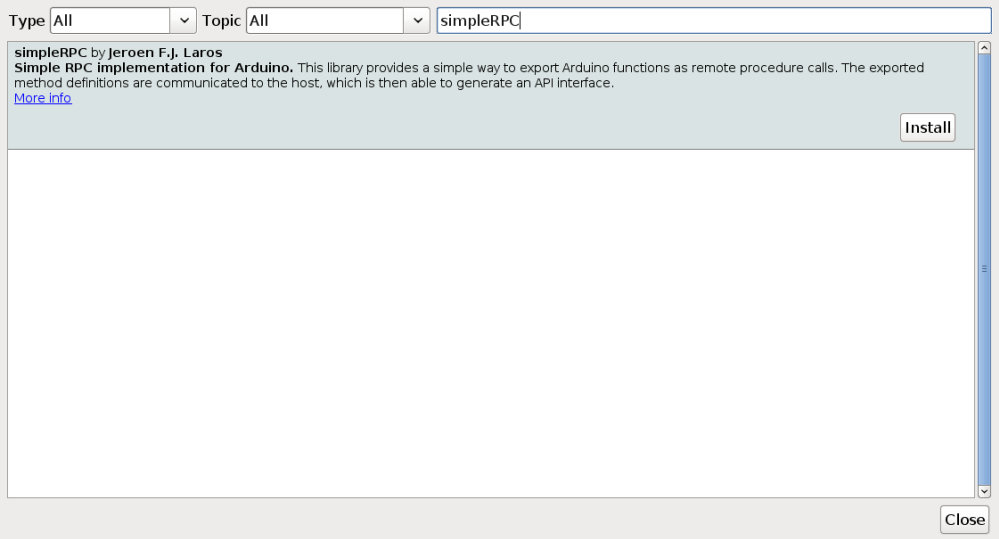
2.2.1 Arduino IDE

Installation via the Arduino IDE is probably the easiest way.

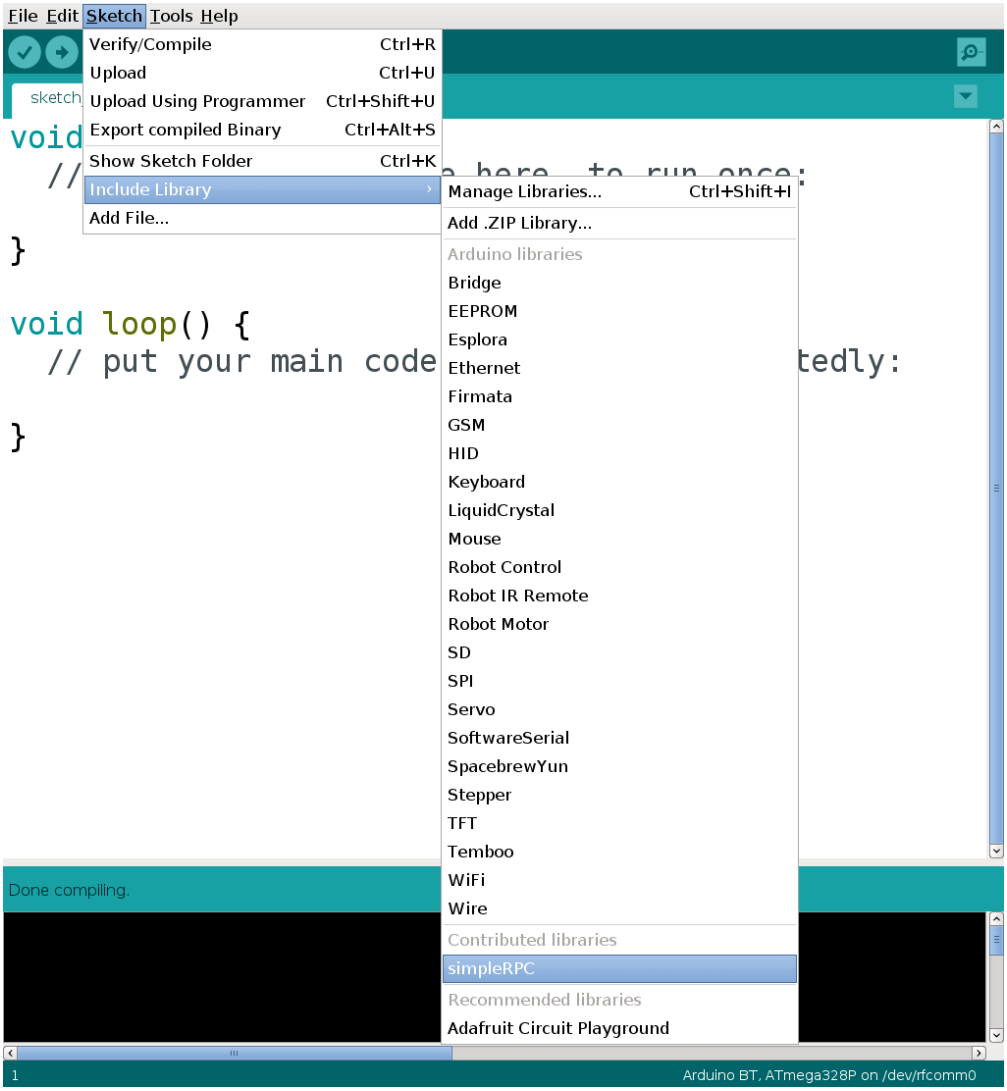
First open the Library Manager.



In the search bar, type “simpleRPC”, click install and close the Library Manager.



The simpleRPC library is now available in the "Contributed libraries" section.



2.2.2 Manual installation

Download

Latest release

Navigate to the latest release and either download the `.zip` or the `.tar.gz` file.

Unpack the downloaded archive.

From source

The source is hosted on GitHub, to install the latest development version, use the following command.

```
git clone https://github.com/jfjlaros/simpleRPC.git
```

Installation

Arduino IDE

In the Arduino IDE, a library can be added to the list of standard libraries by clicking through the following menu options.

- Sketch
- Import Library
- Add Library

To add the library, navigate to the downloaded folder and select the subfolder named `src`.

- Click OK.

Now the library can be added to any new project by clicking through the following menu options.

- Sketch
- Import Library
- simpleRPC

Ino

Ino is an easy way of working with Arduino hardware from the command line. Adding libraries is also easy, simply place the library in the `lib` subdirectory.

```
cd lib
git clone https://github.com/jfjlaros/simpleRPC.git
```

2.3 Library

Include the header file to use the device library.

```
#include <simpleRPC.h>
```

The library provides the `interface()` function, which is responsible for all serial communication with the host. To use this function, first enable serial communication at 9600 baud in the `setup()` body.

```
void setup(void) {
  Serial.begin(9600);
}
```

2.3.1 Standard methods

Methods are exported by calling the `interface()` function from the `loop()` body. This function accepts (function, documentation) pairs as parameters.

Table 1: Interface function parameters.

parameter	description
0	Method one.
1	Documentation string of method one.
2	Method two.
3	Documentation string of method two.
...	...

A documentation string consists of a list of key-value pairs in the form `key: value` delimited by the `@` character. The first pair in this list is reserved for the method name and its description, all subsequent pairs are used to name and describe parameters or to describe a return value.

Table 2: Documentation string.

field prefix	key	value
	Method name.	Method description.
`@`	Parameter name.	Parameter description.
`@`	`return`	Return value description.

The documentation string may be incomplete or empty. The following defaults are used for missing keys. All descriptions may be empty.

Table 3: Default names.

key	default
Method name.	`method` followed by a number, e.g., `method2`.
Parameter name.	`arg` followed by a number, e.g., `arg0`.
`return`	`return`

To reduce the memory footprint, the use of the `F()` macro is allowed in the `interface()` function. This stores the documentation string in program memory instead of SRAM. For more information, see the progmem documentation.

Example

Suppose we want to export a method that sets the brightness of an LED and a method that takes one parameter and returns a value.

```
void setLed(byte brightness) {
  analogWrite(13, brightness);
}

int inc(int a) {
  return a + 1;
}
```

Exporting these methods in the `loop()` body looks as follows:

```
void loop(void) {
  interface(
    inc, "inc: Increment a value. @a: Value. @return: a + 1.",
    setLed, "set_led: Set LED brightness. @brightness: Brightness.");
}
```

We can now build and upload the sketch.

The client reference documentation includes an example on how these methods can be accessed from the host.

2.3.2 Class methods

Class member functions are different from ordinary functions in the sense that they always operate on an object. This is why it is not possible to simply pass a function pointer, but to also provide a class instance for the function to operate on. To facilitate this, the `pack()` function can be used to combine a class instance and a function pointer before passing them to `interface()`.

For a class instance `c` of class `C`, the class member function `f()` can be packed as follows:

```
pack(&c, &C::f)
```

The result can be passed to `interface()`.

Example

Suppose we have a library named *led* which provides the class `LED`. This class has a member function named `setBrightness`.

```
#include "led.h"

LED led(13);
```

Exporting this class method as a remote call goes as follows:

```
void loop(void) {
  interface(
    pack(&led, &LED::setBrightness),
      "set_led: Set LED brightness. @brightness: Brightness.");
}
```

2.3.3 Tuples

Tuples can be used to group multiple objects of different types together. A Tuple has two members, `head` and `tail`, where `head` is of any type, and `tail` is an other Tuple.

Tuples can be initialised with a brace-initializer-list as follows.

```
Tuple <int, char>t = {10, 'c'};
```

Elements of a Tuple can be retrieved in two ways, either via its `head` and `tail` member variables, or with the `get<>()` helper function.

```
int i = t.head;
char c = t.tail.head;

int j = get<0>(t);
char d = get<1>(t)';
```

Likewise, assignment of an element can be done via its member variables or with the `get<>()` helper function.

```
t.head = 11;
t.tail.head = 'd';

get<0>(t) = 11;
get<1>(t) = 'd';
```

There are two additional helper functions available for Tuples: `pack()` and `castStruct().pack()` can be used to create a temporary tuple to be used in a function call.

```
function(pack('a', 'b', 10));
```

Likewise, the `castStruct()` function can be used to convert a C `struct` to a Tuple.

```
struct S {
  int i;
  char c;
};

S s;
function(castStruct<int, char>(s));
```

Note that a Tuple, like any higher order data structure should be passed by reference.

2.3.4 Objects

Objects behave much like Tuples, but they are serialised differently (see the *Protocol* section).

Objects can be initialised via a constructor as follows.

```
Object <int, char>o(10, 'c');
```

Element retrieval and assignment is identical to that of Tuples.

Note that an Object, like any higher order data structure should be passed by reference.

2.3.5 Vectors

A Vector is a sequence container that implements storage of data elements. The type of the vector is given at initialisation time via a template parameter, e.g., `int`.

```
Vector <int>v;
Vector <int>u(12);
```

In this example, Vector `v` is of size 0 and `u` is of size 12. A Vector can also be initialised with a pointer to an allocated block of memory.

```
Vector <int>v(12, data);
```

The memory block is freed when the Vector is destroyed. If this is not desirable, an additional flag `destroy` can be passed to the constructor as follows.

```
Vector <int>v(12, data, false);
```

This behaviour can also be changed by manipulating the `destroy` member variable.

A Vector can be resized using the `resize` method.

```
v.resize(20);
```

The `size` member variable contains the current size of the Vector.

Element retrieval and assignment is done in the usual way.

```
int i = v[10];

v[11] = 9;
```

Note that a Vector, like any higher order data structure should be passed by reference.

2.3.6 Complex objects

Arbitrary combinations of Tuples, Objects and Vectors can be made to construct complex objects.

In the following example, we create a 2-dimensional matrix of integers, a Vector of Tuples and an Object containing an integer, a Vector and an other Object respectively.

```
Vector <Vector <int> >matrix;

Vector <Tuple <int, char> >v;

Object <int, Vector <int>, Object <char, long> >o;
```

2.4 Protocol

In this section we describe the serial protocol.

Every exported method defined using the `interface()` function (see the usage_device section) is assigned a number between 0 and 254 in order of appearance. The number 0 maps to the first method, the number 1 maps to the second method, etc.

There are two types of calls to the device: the method discovery call and a remote procedure call. In both cases, communication is initiated by the host by writing one byte to the serial device.

2.4.1 Method discovery

Method discovery is initiated by the host by writing one byte with value `0xff` to the serial device.

The device will respond with a header and a list of method descriptions delimited by an end of string signature (`\0`). The list is terminated by an additional end of line signature. The header format is given in the following table.

Table 4: Header format.

size	delimiter	value	description
	`\0`	`simpleRPC`	Protocol identifier.
3		`\3\0\0` (example)	Protocol version (major, minor, patch).
1		< or >	Endianness, < for little-endian, > for big-endian.
1		`H` (example)	Type of `size_t`, needed for indexing vectors.

Each method description consists of a struct formatted function signature and a documentation string separated by a `;`. The function signature starts with a struct formatted return type (if any), followed by a `:` and a space delimited list of struct formatted parameter types. The format of the documentation string is described in the usage_device section.

For our example, the response for the method discovery request will look as follows.

```
h: h;inc: Increment a value. @a: Value. @return: a + 1.\0
: B;set_led: Set LED brightness. @brightness: Brightness.\0
\0
```

For more complex objects, like Tuples, Objects and Vectors, some more syntax is needed to communicate their structure to the host.

A Tuple type is encoded as a compound type, e.g., `hB` (a 16-bit integer and a byte). It can be recognised by the absence of a space between the type signatures. Note that a concatenated or nested Tuple type can not be recognised from its signature, e.g., `hB` concatenated with `ff` is indistinguishable from `hBff`

An Object type is encoded as a compound type like a Tuple, but its type signature is enclosed in parentheses `(` and `)`, which makes it possible to communicate its structure to the host, e.g., the concatenation of `(hB)` and `(ff)` is `(hB)(ff)` and the type signature of a nested Object may look like this `((hB)(ff))`.

A Vector type signature is enclosed in brackets `[` and `]`. So a vector of 16-bit integers will have as type signature `[h]`.

Finally, any arbitrary combination of Tuples, Objects and Vectors can be made, resulting in type signatures like `[((hB)f)]`, i.e., a Vector of Objects that contain a Tuple of which the first element is an other Object `(hB)` and the second element is a float `f`.

2.4.2 Remote procedure calls

A remote procedure call is initiated by the host by writing one byte to the serial device of which the value maps to one of the exported methods (i.e., 0 maps to the first method, 1 to the second, etc.). If this method takes any parameters, their values are written to the serial device. After the parameter values have been received, the device executes the method and writes its return value (if any) back to the serial device.

All native C types (`int`, `float`, `double`, etc.), Tuples, Objects, Vectors and any combination of these are currently supported. The host is responsible for packing and unpacking of the values.

There is currently one built-in function.

Table 5: Built-in functions.

index	name	description	parameters	returns
0	ping	Echo a value.	int data: Value.	int: Value of data.

2.4.3 Overhead

There does not seem to be any noticeable overhead of the library. We tested the throughput by calling the `ping()` function repeatedly at baud rates ranging from 600 baud to 115200 baud.

Table 6: Throughput statistics.

baud rate	calls per second
600	30
1200	60
2400	120
4800	239
9600	480
14400	718
19200	959
28800	1447
31250	1559
38400	1920
57600	2924
115200	5838

The number of calls per second scales linearly with the baud rate, even at the highest speed, there is no measurable overhead.

2.5 Contributors

- Jeroen F.J. Laros <jlaros@fixedpoint.nl> (Original author, maintainer)

Find out who contributed:

```
git shortlog -s -e
```