simpleRPC

Nov 23, 2021

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, Vectors and arbitrary combinations of these.
- Support for reading multidimensional C arrays (e.g., `int**`).
- Support for different types of I/O interfaces via plugins, e.g.,
- Support for the C++ Standard Template Library.
  - Bluetooth.
  - Ethernet (untested).
  - Hardware serial.
  - RS485 serial.
  - Software serial (untested).
  - USB serial.
  - WiFi.
  - Wire (untested).
- Support for using multiple interfaces at the same time.

The Arduino library is independent of any host implementation, a Python API client library is provided 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(Serial, digitalRead, "", digitalWrite, "");
}
```

These functions are now available on the host under names `method0()` and `method1()`.

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

```
interface(
  Serial,
  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 the client reference implementation documentation, contains an example on how this works.

CHAPTER 2

Further reading

Please read *Usage* 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 an I/O device and it uses this to select the appropriated function.

If a function takes parameters, their values need to be read from the I/O device before calling the function. Any return value needs to be written to the I/O 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 where the simpleRPC library comes in, like the implementation above, it only communicates values but has none of the downsides of an *ad hoc* protocol.

2.2 Installation

2.2.1 Arduino IDE

To install this library in the Arduino IDE, please follow these comprehensive installation instructions.

2.2.2 Arduino CLI

The latest version can be installed with the Arduino CLI interface using the following command.

```
arduino-cli lib install simpleRPC
```

2.2.3 Manual installation

Latest release

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

From source

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

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

2.3 Usage

Include the header file to use the simpleRPC library.

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

The library provides the `interface()` function, which is responsible for all communication with the host. To use this function, first initialize the standard `Serial` class instance to enable communication using the hardware serial interface. This is done using the `begin()` method in the `setup()` body.

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

Please see the *Plugins* section for using other I/O interfaces.

2.3.1 Exporting C functions

Standard C functions are exported as RPC methods 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	I/O class instance.
1	Function one.
2	Documentation string of function one.
3	Function two.
4	Documentation string of function 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 RPC 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
	RPC method name.	RPC 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
RPC method name.	`method` followed by a number, e.g., `method0`.
Parameter name.	`arg` followed by a number, e.g., `arg0`.
`return`	`return`

To reduce the memory footprint, the `F()` macro can be used 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 function that sets the brightness of an LED and a function that takes one parameter and returns a value.

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

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

Exporting these functions goes as follows:

```
void loop(void) {
  interface(
    Serial,
    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 Exporting class methods

Class methods are different from ordinary functions in the sense that they always operate on an object. This is why both a function pointer and a class instance need to be provided to the `interface()` function. 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 method `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 method named `setBrightness`.

```
#include "led.h"

LED led(LED_BUILTIN);
```

Exporting this class method goes as follows:

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

2.3.3 Complex objects

In some cases, basic C types and C strings are not sufficient or convenient. This is why simpleRPC supports higher order objects described in detail in the *Tuples* and *Vectors* sections.

Arbitrary combinations of these higher order objects can be made to construct complex objects.

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

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

Vector<Tuple<int, char> > v;

Tuple<int, Vector<int>, Tuple<char, long> > t;
```

These objects can be used for parameters as well as for return values. Note that these objects, like any higher order data structure should be passed by reference.

2.3.4 C arrays

Passing a C array as a parameter is supported, but since in general it is not possible to deduce the size or internal structure of an object it is not possible to return a C array. The closely related Vector should be used in this case.

In the following example, an integer C array is passed to a function.

```
void readArray(int* a) {}
```

Multidimensional arrays are implemented as `NULL` terminated arrays of pointers. This allows for structures that do not have a fixed length in any dimension, e.g., a two-dimensional array `int**` does not have to be rectangular.

2.4 Plugins

The library supports I/O plugins in order to enable RPC communication over a range of interfaces. Currently, the following plugins are implemented.

Table 4: Plugins.

name	description	status
`Serial`	The standard Arduino Serial interface.	working
`SoftwareSerial`	The Arduino SoftwareSerial interface.	untested
`HalfDuplexStream`	RS485 serial interface.	working
`EthernetClient`	Arduino Ethernet interface.	untested
`WiFiClient`	Arduino WiFi interface.	working
`Wire`	I2C / TWI Wire interface.	untested

A plugin inherits from `Stream` and should override the following methods.

Table 5: Methods.

name	description
	Constructor.
`int available()`	Number of bytes available for reading.
`int read()`	Read a single byte or `-1` upon error.
`int peek()`	Preview the next byte.
`size_t write(uint8_t)`	Write a single byte, return the number of bytes written.

Usually, the I/O plugin is declared as a global object instance in the sketch and initialized in the `setup()` function. See the RS485 sketch for an example that uses a custom I/O plugin.

2.4.1 Multiple I/O interfaces

It is possible to use multiple I/O interfaces at the same time. This can be done by either serving a different set of methods on each interface or by serving the same set of methods on multiple interfaces.

To serve different methods on each interface, the `interface()` function is simply used multiple times.

Example

Suppose we have set up two I/O interfaces named `Serial` and `SerialUSB`, we serve different methods on each of the interfaces as follows.

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

Alternatively, it is possible to serve the same set of methods on multiple interfaces. This can be done by passing an internal Tuple of pointers to the interfaces as the first parameter of the `interface()` function.

Example

Suppose we have set up two I/O interfaces named `Serial` and `SerialUSB`, we serve the same methods on both interfaces by grouping pointers to these interfaces with the `pack()` function as follows.

```
void loop(void) {
  interface(
    pack(&Serial, &SerialUSB),
    inc, F("inc: Increment a value. @a: Value. @return: a + 1."));
}
```

Finally, it is possible to combine both of the strategies described above.

2.5 Protocol

In this section we describe the RPC protocol.

Every exported method defined using the `interface()` function (see the *Usage* 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 I/O device.

2.5.1 Method discovery

Method discovery is initiated by the host by writing one byte with value `0xff` to the I/O 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 string signature. The header format is given in the following table.

Table 6: 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* 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 and Vectors, some more syntax is needed to communicate their structure to the host.

An internal 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`. This type is for internal use only, it is not recommended for use in RPC calls.

A Tuple type is encoded as a compound type like an internal 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 Tuple 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 and Vectors can be made, resulting in type signatures like `[((hB)f)]`, i.e., a Vector of Tuples that contain a Tuple of which the first element is an other Tuple `(hB)` and the second element is a float `f`.

2.5.2 Remote procedure calls

A remote procedure call is initiated by the host by writing one byte to the I/O 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 I/O device. After the parameter values have been received, the device executes the method and writes its return value (if any) back to the I/O device.

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

2.6 API documentation

2.6.1 RPC interface

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

See the *Usage* section for a full description of the RPC interface.

Functions

template <class... *Args*>
void **interface**(Stream &*io*, Args... *args*)

RPC interface.

The `args` parameter pack is a list of pairs (function pointer, documentation). The documentation string can be of type `char const*`, or the PROGMEM `F()` macro can be used to reduce memory footprint.

Parameters

- `io` - Stream.
- `args` - Parameter pairs (function pointer, documentation).

template <class... *Membs*, class... *Args*>
void **interface**(*_Tuple*<Membs...> *t*, Args... *args*)

Multiple RPC interfaces.

Similar to the standard interface , but with support for multiple I/O interfaces, passed as internal *Tuple* `t`.

See *interface*(Stream&, Args...)

Parameters

- `t` - Internal *Tuple* of streams.
- `args` - Parameter pairs (function pointer, documentation).

2.6.2 Tuples

```
#include "tuple.tcc"
```

Tuples

Tuples can be used to group objects of different types. A Tuple is recursively defined as being either:

- Empty.
- A pair (`head`, `tail`), where `head` is of an arbitrary type and `tail` is an other Tuple.

Initialisation of a Tuple can be done with a brace-initializer-list as follows.

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

Element retrieval and assignment is described below in the Helper functions section.

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

Class definitions

template <class... *Membs*>
struct Empty internal *Tuple*.
template <**class** H, class... *Tail*>
struct
class **_Tuple**<H, Tail...>
Non-empty internal *Tuple*.
template <class... *Membs*>
struct *Tuple*.

Inherits from *_Tuple*< *Membs...* >

Helper functions

Elements of a Tuple can be retrieved in two ways, either via the `head` and `tail` member variables, or using 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 additional helper functions available for the creation of Tuples.

The function `pack()` can be used to create a temporary Tuple to be used in a function call.

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

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));
```

Functions

template <size_t *k*, class... *Membs*>
get(*_Tuple*<Membs...> &*t*)

Get the *k*-th element of an internal *Tuple*.

This can be used for both retrieval as well as assignment.

Return Reference to the *k*-th element in `t`.

Parameters

- `t` - An internal *Tuple*.

template <class... *Args*>
_Tuple<Args...> **pack**(Args... *args*)

Make an internal *Tuple* from a parameter pack.

Return Internal *Tuple* containing `args`.

Parameters

- `args` - Values to store in an internal *Tuple*.

template <class... *Membs*, **class** T>
_Tuple<Membs...> **castStruct**(T &*s*)

Cast a `struct` to an internal *Tuple*.

Return Internal *Tuple* representation of `s`.

Parameters

- `s` - Struct.

2.6.3 Vectors

```
#include "vector.tcc"
```

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.

Class definition

template <class T>
class Generic *Vector*.

2.6.4 Input / output

Convenience functions for reading and writing. A template class `I`, is used as an abstraction for I/O devices like serial ports, wire interfaces and network interfaces like ethernet. An overview of the required methods of an I/O plugin is described in the plugins section.

Printing

```
#include "print.tcc"
```

The following functions take care of serialisation of:

- Values of basic types.
- C strings (`char[]`, `char*`, `char const[]`, `char const*`).
- C++ Strings.
- PROGMEM strings (`F()` macro).

Finally, a print function that takes an arbitrary amount of parameters is provided for convenience.

Functions

template <class T>
void **rpcPrint**(Stream &*io*, T *data*)
Print a value to a stream.

Parameters

- `io` - Stream.
- `data` - Data.

void **rpcPrint**(Stream &*io*, char **data*)
Print a value to a stream.

Parameters

- `io` - Stream.
- `data` - Data.

void **rpcPrint**(Stream &*io*, char **const** **data*)
Print a value to a stream.

Parameters

- `io` - Stream.
- `data` - Data.

void **rpcPrint**(Stream &*io*, String &*data*)
Print a value to a stream.

Parameters

- `io` - Stream.
- `data` - Data.

void **rpcPrint**(Stream &*io*, __FlashStringHelper **const** **data*)
Print a value to a stream.

Parameters

- `io` - Stream.
- `data` - Data.

template <class H, class... *Tail*>
void **rpcPrint**(Stream &*io*, H *data*, Tail... *args*)
Print any number of values.

Parameters

- `io` - Stream.
- `data` - Value to be printed.
- `args` - Remaining values.

STL functions

The following functions are only available for cores that have support for the C++ STL. See the *Standard Template Library* section for more information.

void **rpcPrint**(Stream &*io*, std::string &*data*)
Print a value to a stream.

Parameters

- `io` - Stream.
- `data` - Data.

Reading

Read functions for deserialisation.

```
#include "read.tcc"
```

Functions

template <class T>
void **rpcRead**(Stream &*io*, T **data*)
Read a value from an stream.

Parameters

- `io` - Stream.
- `data` - Data.

void **rpcRead**(Stream &*io*, char ***data*)
Read a value from an stream.

Parameters

- `io` - Stream.
- `data` - Data.

void **rpcRead**(Stream &*io*, char **const** **_data_)
Read a value from an stream.

Parameters

- `io` - Stream.
- `data` - Data.

void **rpcRead**(Stream &*io*, String *_data_)
Read a value from an stream.

Parameters

- `io` - Stream.
- `data` - Data.

template <**class** T>
void **rpcRead**(Stream &*io*, *Vector*<T> *_data_)
Read a value from an stream.

Parameters

- `io` - Stream.
- `data` - Data.

template <**class** T>
void **rpcRead**(Stream &*io*, T **_data_)
Read a value from an stream.

Parameters

- `io` - Stream.
- `data` - Data.

template <**class** T>
void **rpcRead**(Stream &*io*, T ***_data_)
Read a value from an stream.

Parameters

- `io` - Stream.
- `data` - Data.

template <class... *Membs*>
void **rpcRead**(Stream &*io*, *_Tuple*<Membs...> *_data_)
Read a value from an stream.

Parameters

- `io` - Stream.
- `data` - Data.

template <class... *Membs*>
void **rpcRead**(Stream &*io*, *Tuple*<Membs...> *_data_)
Read a value from an stream.

Parameters

- `io` - Stream.
- `data` - Data.

STL functions

The following functions are only available for cores that have support for the C++ STL. See the *Standard Template Library* section for more information.

void **rpcRead**(Stream &*io*, std::string **data*)
Read a value from an stream.

Parameters

- `io` - Stream.
- `data` - Data.

Writing

Write functions for serialisation.

```
#include "read.tcc"
```

Functions

template <class T>
void **rpcWrite**(Stream &*io*, T **data*)
Write a value to an stream.

Parameters

- `io` - Stream.
- `data` - Data.

void **rpcWrite**(Stream &*io*, char ****data*)
Write a value to an stream.

Parameters

- `io` - Stream.
- `data` - Data.

void **rpcWrite**(Stream &*io*, char **const** ****data*)
Write a value to an stream.

Parameters

- `io` - Stream.
- `data` - Data.

void **rpcWrite**(Stream &*io*, String **data*)
Write a value to an stream.

Parameters

- `io` - Stream.
- `data` - Data.

template <**class** T>
void **rpcWrite**(Stream &*io*, *Vector*<T> **data*)
Write a value to an stream.

Parameters

- `io` - Stream.
- `data` - Data.

template <class... *Membs*>
void **rpcWrite**(Stream &*io*, *_Tuple*<Membs...> **data*)
Write a value to an stream.

Parameters

- `io` - Stream.
- `data` - Data.

template <class... *Membs*>
void **rpcWrite**(Stream &*io*, *Tuple*<Membs...> **data*)
Write a value to an stream.

Parameters

- `io` - Stream.
- `data` - Data.

STL functions

The following functions are only available for cores that have support for the C++ STL. See the *Standard Template Library* section for more information.

void **rpcWrite**(Stream &*io*, std::string **data*)
Write a value to an stream.

Parameters

- `io` - Stream.
- `data` - Data.

2.6.5 Types

```
#include "types.tcc"
```

Get a struct formatted representation of the type of a value.

Table 7: Type encoding examples.

type	encoding	description
`bool`	`?`	Boolean.
`unsigned long int`	`L`	Unsigned integer.
`char const*`	`s`	String.
`Tuple<Tuple<char, int>, unsigned long>`	`((ci)L)`	Tuple.
`Vector<int>`	`[i]`	Vector.

Functions

void **rpcTypeOf**(Stream &*io*, bool)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, char)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, signed char)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, unsigned char)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, short int)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, unsigned short int)
Type encoding.

Parameters

- io - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, long int)
Type encoding.

Parameters

- io - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, unsigned long int)
Type encoding.

Parameters

- io - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, long long int)
Type encoding.

Parameters

- io - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, unsigned long long int)
Type encoding.

Parameters

- io - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, float)
Type encoding.

Parameters

- io - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, String&)
Type encoding.

Parameters

- io - Stream.

- – - Value.

void **rpcTypeOf**(Stream &*io*, char *)
Type encoding.

Parameters

- io - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, char **const** *)
Type encoding.

Parameters

- io - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, int)
Type encoding.

Parameters

- io - Stream.
- – - Value.

void **rpcTypeOf**(Stream &*io*, double)
Type encoding.

Parameters

- io - Stream.
- – - Value.

template <class... *Membs*>
void **rpcTypeOf**(Stream &*io*, *_Tuple*<Membs...> &*t*)
Get the types of all members of an intnernal *Tuple*.

Parameters

- io - Stream.
- t - Internal *Tuple*.

template <class... *Membs*>
void **rpcTypeOf**(Stream &*io*, *Tuple*<Membs...> &*o*)
Get the types of all members of a *Tuple*.

Parameters

- io - Stream.
- t - *Tuple*.

template <**class** T>
void **rpcTypeOf**(Stream &*io*, *Vector*<T>&)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

template <class T>
void **rpcTypeOf**(Stream &*io*, T *)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

void **hardwareDefs**(Stream &*io*)
Determine endianness and type of `size_t`.

Parameters

- `io` - Stream.

STL functions

The following functions are only available for cores that have support for the C++ STL. See the *Standard Template Library* section for more information.

void **rpcTypeOf**(Stream &*io*, std::string &*t*)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

template <class... *Membs*>
void **rpcTypeOf**(Stream &*io*, std::tuple<Membs...> &*t*)
Get the types of all members of a *Tuple*.

Parameters

- `io` - Stream.
- `t` - *Tuple*.

template <class T>
void **rpcTypeOf**(Stream &*io*, std::vector<T> &*t*)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

template <class T, std::size_t *N*>
void **rpcTypeOf**(Stream &*io*, std::array<T, N> &*t*)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

template <class T>
void **rpcTypeOf**(Stream &*io*, std::list<T> &*t*)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

template <class T>
void **rpcTypeOf**(Stream &*io*, std::forward_list<T> &*t*)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

template <class T>
void **rpcTypeOf**(Stream &*io*, std::set<T> &*t*)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

template <class T>
void **rpcTypeOf**(Stream &*io*, std::unordered_set<T> &*t*)
Type encoding.

Parameters

- `io` - Stream.
- – - Value.

2.6.6 Function Signatures

```
#include "signature.tcc"
```

Table 8: Function signature examples.

signature	encoding	description
`short int f(char, float)`	`h: c f`	Function that returns a value.
`void f(char, float)`	`: c f`	Function that does not return a value.
`Tuple<int, char> f(float)`	`(ic): f`	Returning an Tuple.
`int f(Vector<signed char>&, int)`	`i: [b] i`	A Vector parameter.

Functions

template <class R, class... *FArgs*>
void **signature**(Stream &*io*, R (**f*)) FArgs...
Get the signature of a function.

Return Function signature.

Parameters

- `io` - Stream.
- `f` - Function pointer.

template <class R, **class** C, class... *FArgs*>
void signature(Stream & io, R(C::*)(FArgs...) f)
Get the signature of a function.

Return Function signature.

Parameters

- `io` - Stream.
- `f` - Function pointer.

template <class... *FArgs*>
void **signature**(Stream &*io*, void (**f*)) FArgs...
Get the signature of a function.

Return Function signature.

Parameters

- `io` - Stream.
- `f` - Function pointer.

template <class C, class... *FArgs*>
void signature(Stream & io, void(C::*)(FArgs...) f)
Get the signature of a function.

Return Function signature.

Parameters

- `io` - Stream.
- `f` - Function pointer.

2.6.7 RPC function calls

```
#include "rpcCall.tcc"
```

Functions

template <class R, class... *FArgs*>
void **rpcCall**(Stream &*io*, R (**f*)) FArgs...
Call a function.

Parameter values for `f` are read from `io`, after which `f` is called. Any return value is written back to `io`.

Parameters

- `io` - Stream.
- `f` - Function pointer.

template <class C, **class** P, **class** R, class... *FArgs*>
void **rpcCall**(Stream &*io*, *_Tuple*<C *, R (P::*)) FArgs...
> *t*Call a class method.

See *rpcCall*(Stream&, R (*)(FArgs...))

Parameters

- `io` - Stream.
- `t` - Internal *Tuple* consisting of a pointer to a class instance and a pointer to a class method.

2.6.8 Deleting objects

```
#include "del.tcc"
```

Functions

template <class T>
void **rpcDel**(T **data*)
Delete a value.

Parameters

- `data` - Data.

template <class T>
void **rpcDel**(T ***data*)
Delete a value.

Parameters

- `data` - Data.

template <class T>
void **rpcDel**(T **const** ***data*)
Delete a value.

Parameters

- `data` - Data.

template <class T>
void **rpcDel**(T ****data*)
Delete a value.

Parameters

- `data` - Data.

template <class T>
void **rpcDel**(T **const** ****data*)
Delete a value.

Parameters

- `data` - Data.

template <class... *Membs*>
void **rpcDel**(*_Tuple*<Membs...> **data*)
Delete a value.

Parameters

- `data` - Data.

template <class... *Membs*>
void **rpcDel**(*Tuple*<Membs...> **data*)
Delete a value.

Parameters

- `data` - Data.

2.6.9 Standard Template Library

Some cores like the SAM core and the ESP32 core have support for the C++ Standard Template Library which provides some useful STL containers. Currently the following STL containers can be used in RPC calls.

- Arrays.
- Strings.
- Tuples.
- Vectors.

Installation

The ESP32 core supports the STL by default. For the SAM core, some modifications need to be made to the platform configuration file.

First locate your arduino configuration folder, navigate to `packages/arduino/hardware/sam/x.y.z` (where `x.y.z` is a version string e.g., `1.6.12`).

Open the file `platform.txt` and search for the line starting with `recipe.c.combine.pattern`, add `-lstdc++ -specs=nano.specs` to the end of this line.

Usage

Include the following header file instead of the standard one.

```
#include <simpleRPC_STL.h>
```

Example

See the esp32_stl sketch for an example.

2.7 Contributors

- Jeroen F.J. Laros <jlaros@fixedpoint.nl> (Original author, maintainer)
- Chris Flesher <chris.flesher@stoneaerospace.com> - Support for non-AVR architectures. - Ethernet support. - RS485 serial support. - USB serial support. - WiFi support.

Find out who contributed:

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

Index

Symbols

C

G

H

I

P

R

S

T

V