ARM Ltd.
110 Fulbourn Road
Cambridge
GB-CB1 9NJ
Great Britain

Industrial Engineering Internship Report

Embedded systems, Internet of Things and Communication

Télécom Paristech mentor:
Samuel Tardieu

Supervisor:
Simon Ford

Author:
Samuel Mokrani

This report is made in the context of the last year engineering internship of Télécom Paristech. I did my internship at ARM, more precisely in the Mbed team from the 4th of July 2011 until the 3rd of February 2012.

January 10, 2012

Acknowledgment

I took immense pleasure in this internship and would like to begin this report by thanking several people.

Firstly, I wish to express my deep sense of gratitude to my manager, Mr Simon Ford. This internship would not have been possible without him. I also will never forget the excellent atmosphere in the mbed team thanks to: Mr Simon Ford, Mr Chris Style, Mr Dan Ros, Mr Emilio Monti and Mr Amit Naran.

I also want to thank two teachers of Télécom Paristech: Mr Alexis Polti and Mr Samuel Tardieu who gave me working method and taught me many things. I will never forget my two month project during which I worked with them on an octocopter. It was a unique experience.

Special thanks to the summer intern Nathan Hutton. We worked on the realtime data streaming from sensors project together, and it was a great pleasure to work with him.

Finally, I would like to express my heartfelt thanks to my parents. They are always by my side. I owe them everything.

Contents

Chapter 1

ARM and Mbed

1.1 ARM

1.1.1 History

ARM Holdings plc is a British multinational semiconductor and software company. The headquarters are based in Cambridge, where it was established. ARM is well known in the processor field, although it also designs, licenses and sells software development tools under the RealView and KEIL brands, systems and platforms, system-on-chip infrastructure and software.

Advanced RISC Machines Ltd (now ARM Ltd) was founded in November 1990. It is the result of a joint venture between Acorn Computers, Apple Computer (now Apple Inc.) and VLSI Technology (now NXP semiconductor). The purpose of this joint venture was to develop a RISC chip originally developed by Acorn Computer involved in an Apple project. ARM became bigger throughout the years by acquiring companies like Micrologic Solutions, a software consulting company based in Cambridge 1999. In 2000, ARM acquired Allant Software, Infinite Designs in Sheffield (UK) and EuroMips in Sophia Antipolis (France). Then, in 2001, it acquired a team specialized in hardware and software debugging based in Blackburn (UK). In 2005, ARM acquired Keil Software, a leader in software development tools for microcontrollers. More recently, ARM joined with Texas Instruments, Samsung, IBM, ST-Ericsson and Freescale Semiconductor in forming an open source engineering company named Linaro. Linaro produces for example ARM tools or linux kernels for ARM based system-on-chips. Today, ARM has offices and design centres all over the world, including the UK, Germany, France, Israel, Sweden, Norway, Slovenia, USA, China, South Korea, Japan, Taiwan and India.

Nowadays, ARM processors are widely used in mobile phones, tablets, personal digital assistants, GPS, digital cameras and digital televions. The main reason for this success is their low electric power consumption, making them suitable for embedded systems. Even if ARM products are widely used, the company itself doesn't manufacture its own CPUs. ARM licenses its technology as Intellectual Property (IP). Companies like Intel, Texas Intrument or Nvidia are making processors based on ARM's IP.

1.1.2 ARM Processor Architecture

The ARM architecture forms the basis around which every ARM processor is built. The ARM architecture is generally described as a Reduced Instruction Set Computer (RISC) architecture. Main RISC features are:

- Load/store architecture: data-processing operates only on register contents
- Uniform and fixed-length instruction fields, to simplify instruction decode and pipelining
- A large register file

Over time the ARM architecture has evolved in order to improve performance, provide new functionalities or meet the demand for new needs.

- **ARMv5**: support for digital signal processing (DSP) algorithms and the Jazelle Java byte code engine to enable hardware execution of Java bytecodes, thus improving performance of applications written in Java
- **ARMv6**: introduction of the SIMD (Single Instruction Multiple Data) technology. This extension is particularly used in audio and video processing. In addition a security extension has been introduced known as TrustZone. The Thumb-2 mode has also been introduced to improve code density and performance

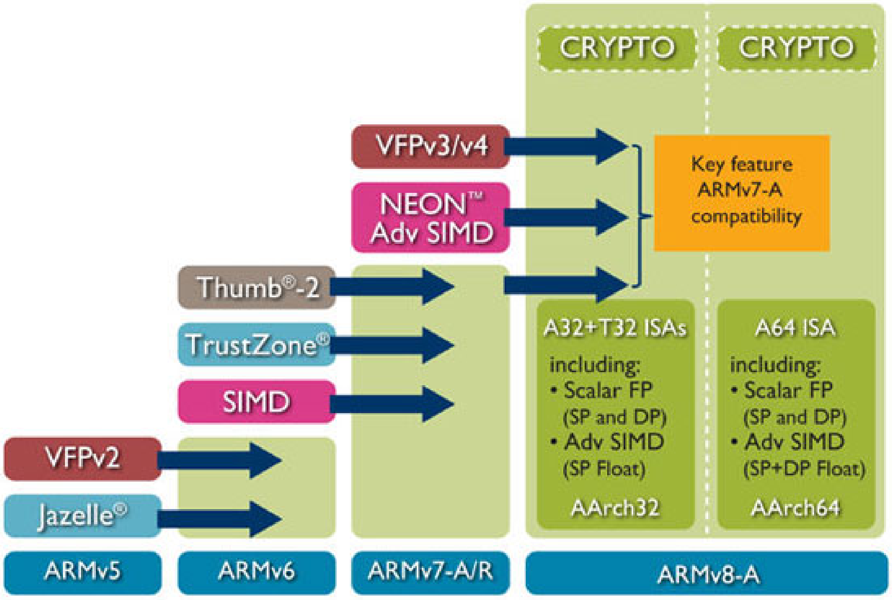


Figure 1.1: Evolution of the ARM architecture

- **ARMv7**: this architecture delivers a set of profiles customized to target applications. All Cortex processors implement the ARMv7 architecture, (except Cortex M0 and Cortex M1 series which implement ARMv6M):
  - **Cortex A**: Application profile implementing a virtual memory system architecture based on an MMU (memory management unit). An optional NEON processing unit for multimedia applications and advanced hardware Floating Point unit can be added
  - **Cortex R**: Realtime profile implementing a protected memory system architecture based on an MPU (memory protection unit)
  - **Cortex M**: Microcontroller profile designed for fast interrupt processing and ideal for cost-sensitive devices requiring highly deterministic behaviour
- **ARMv8**: ARMv8 will introduce a 64-bit architecture support. There will also be an instruction level support for cryptography. This new architecture is in development

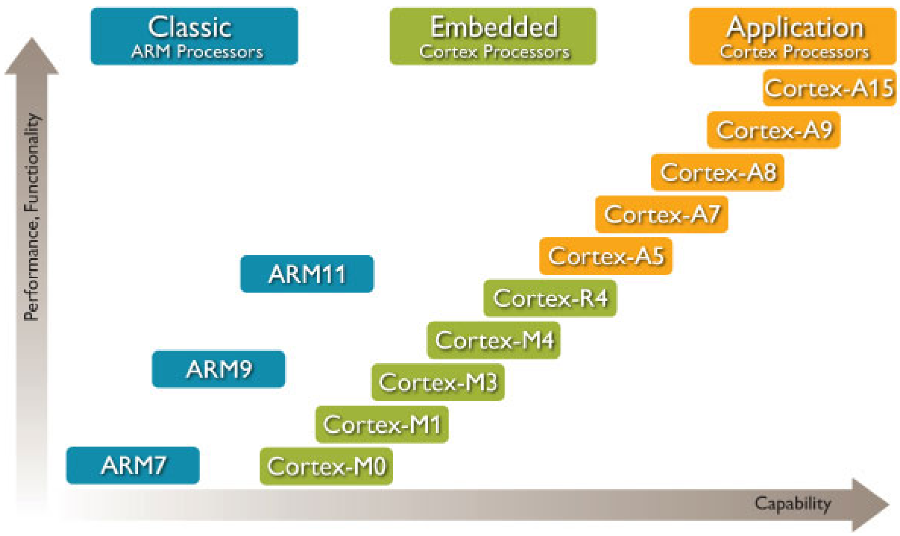


Figure 1.2: ARM processors

1.2 Mbed

Aiming to pursue new areas of expertise, ARM founded Mbed in 2009. Mbed's core area is the development of prototyping boards (called mbed) based on ARM processors (Cortex M0 and Cortex M3). Their purpose is to make simple and easy to setup prototyping solutions using 32-bit processors. Mbeds boards have been designed for quick experimentation; users can try something and see if it is doable in a very quick and easy manner.

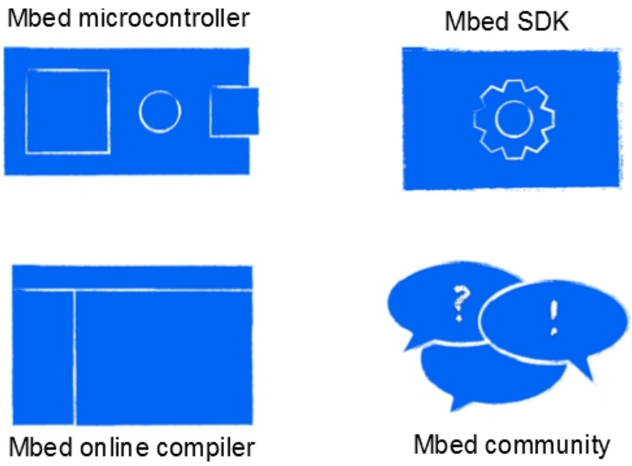
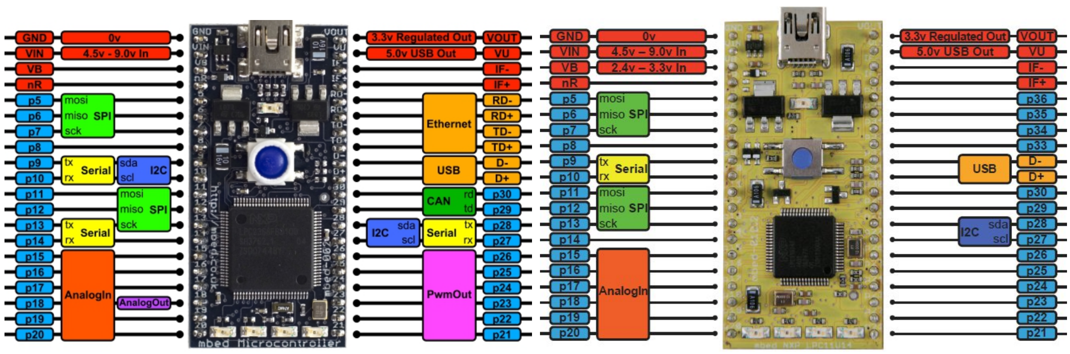


Figure 1.3: Four main elements of Mbed

1.2.1 Mbed boards: LPC1768 and LPC11U24

For the moment, two boards have been designed. The first is based on the LPC1768 microcontroller from NXP which uses a Cortex M3:

- 96 MHz, 32 Kb of RAM, 512 Kb of flash
- 3 UART, 2 SPI, 2 I2C, 6 PWM, 6 ADC, GPIO, Ethernet, USB Host/Device



(a) mbed lpc1768 (Cortex M3)

(b) mbed lpc11U24 (Cortex M0)

Figure 1.4: Mbed boards

More recently, a new board based on a LPC11U24 (Cortex M0) was introduced. This mbed has been designed to prototype USB devices or battery powered applications:

- 48 MHz, 8 Kb of RAM, 32 Kb of flash
- UART, I2C, 2 SPI, 6 ADC, GPIO, USB Device

Both mbed boards have a built-in USB drag and drop FLASH programmer. This feature allows mbed users to drag and drop a binary into the mbed, recognized as a USB mass storage device by a computer. It is very convenient and simple to flash a new program into the mbed:

- create a program (online IDE or offline toolchains)
- copy the binary generated into the mbed disk
- press the reset button

1.2.2 The online compiler

The two main keywords of Mbed are "simple" and "cloud computing". Users can use an online compiler to develop their programs in C++. They compile online and transfer the binary received into the mbed, which is connected to a computer over a USB cable. They just have to press the reset button to see their program running.

The online IDE is totally independent from the underlying operating system. It contains a lot of interesting features:

- Code editing with syntax highlighting
- Multiple programs
- Import programs from online catalogue of published programs
- Import programs from zip file
- Full output of compile-time messages
- Multiple target support
- Publish your code directly from the compiler
- Export your programs as a .zip file
- Build information including graphical display of code size and RAM usage
- Code formatter
- Import and update of libraries from SVN
- Version control: you can commit, revert, update, merge your programs

1.2.3 Mbed libraries

In addition to the online compiler, almost all drivers have been implemented. Users just have to instantiate an object such as SPI, I2C,... to have access to an API which abstracts all the low-level layers. The figure 1.5 shows on the left hand side all the drivers which has been implemented and are accessible over a C++ class. On the right hand side, a description, some basic examples and the API of a specific class can be found. Users can code using meaningful abstract objects and API calls, so there is no need to learn the microcontroller hardware details to get going.

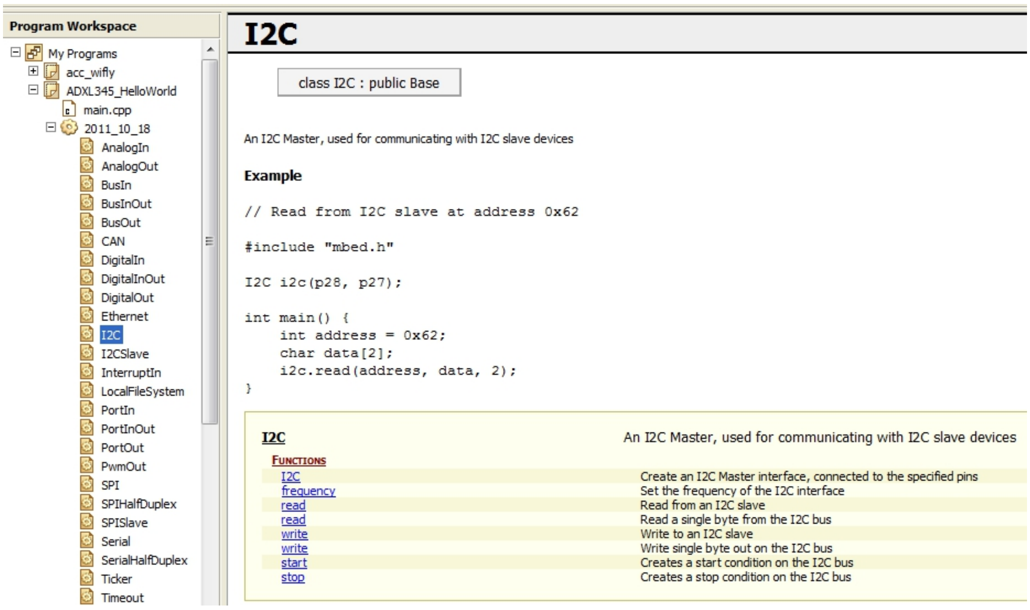


Figure 1.5: mbed libraries

1.2.4 Mbed website: *http://mbed.org/*

Finally, users can access the mbed **forum** to troubleshoot their programs or ask questions. They also have a **handbook** where there is a lot of documentation and examples concerning mbed libraries or the hardware part of the mbed. However, I find that the most important and useful part of the website is the **cookbook**. The cookbook is a great source of examples contributed from, and edited by, other mbed users. You can almost always find a library to use with popular peripherals like accelerometers, pressure sensors... The user is free to contribute to this part of the website by writing articles explaining their project and code.

Chapter 2

Internet Of Things: HTML5 on embedded systems

Nowadays, an increasing number of devices are connected to the Internet. These devices include not only personal computers, but also mobile phones and digital televisions, etc. Cisco predicted in an interview with the BBC that the number of internet connected devices is set to explode in the next four years to over 15 billion, twice the world's population, by 2015. Cisco is not the only company to predict a such boom; VMware's CEO Paul Maritz said during a speech at the 2011 VMworld conference in Las Vegas that:

> "Three years ago over 95 percent of the devices connected to the Internet were personal computers. Three years from now that number will probably be less than 20 percent. More than 80 percent of the devices connected to the Internet will not be Windows-based personal computers."

Thus, the development and progress of technologies connecting different sensors and devices to the Internet is becoming ever more important. I will describe in this chapter two projects concerning the Internet of Things: realtime data streaming from sensors and Remote Procedure Call mechanism. Both of these projects use a new feature of HTML5: a Websocket communication.

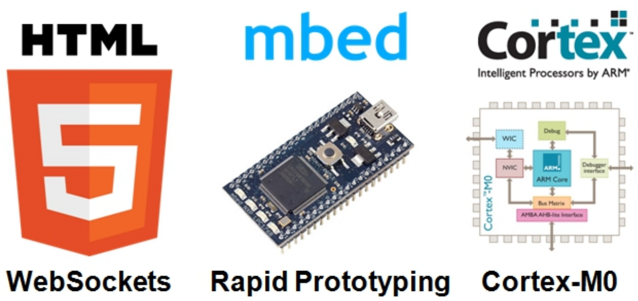


Figure 2.1: HTML5 on embedded systems

2.1 Websockets

2.1.1 Introduction

The WebSocket specification is a new feature of HTML5. It defines a full-duplex single socket connection over which messages can be sent between client and server. The WebSocket standard simplifies much of the complexity around bi-directional web communication and connection management. Furthermore, it reduces polling and unnecessary network throughput overhead.

2.1.2 Why the need for Websockets ?

Traditionally, when a web browser loads a webpage, it follows these basic steps:

- Open a short-lived connection to a web server
- Send a request to the server

- The web server acknowledges this request and sends back the response
- Close the connection to the server

This architecture was designed primarily for document retrieval, where web browsers load static web pages from web servers. A transaction is based on a request/response mechanism. However, more and more frequently web applications require realtime capabilities. Developers wanted a technology where the server can send data to a client without the need of a client request. This technique is called server push. First attempts to provide realtime web applications was based on polling or server push technology:

- **polling**: the browser sends HTTP requests at regular intervals and immediately receives a response. This is an intuitive solution but it's often inefficient because in the case that the server does not have data to send, unnecessary requests will be sent. As a result there is a waste of bandwidth and CPU time on both the client and the server
- **long polling**: a basic example is a XMLHttpRequest object with long polling: The client sends requests through a XMLHttpRequest Javascript object. After each response, the client "rearms" the polling by sending a new request. The server holds client requests open until it has data to send, thus reducing unnecessary requests. However, the request/response mechanism is not avoided. In situations where there is a large volume of messages, long polling results in a continuous loop of immediate polls. Another problem with this technique is that during the reconnection process the data on the page could be out of date and inaccurate
- **streaming**: The browser sends a request and the server sends and maintains an open response that is continuously updated and kept open. This is much more efficient than polling or long polling as no HTTP headers are sent during the streaming period. As a result, the network traffic is reduced. The downside of streaming is that it is still encapsulated in HTTP, so intervening HTTP proxies may choose to buffer the response, increasing the latency of the message delivery. Alternatively, the proxy server may be configured to disconnect HTTP connections that are kept open for a certain amount of time.

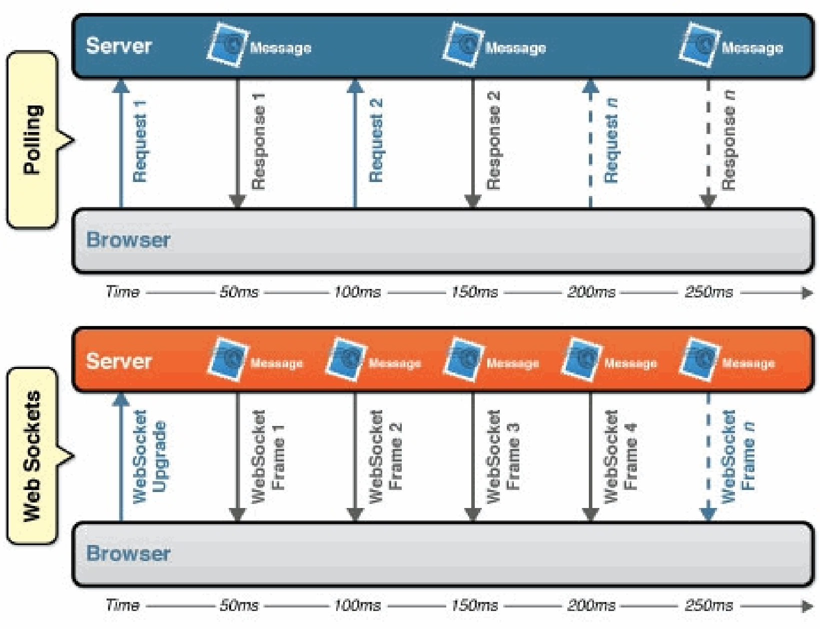


Figure 2.2: Reduction of polling and overhead by Websocket (Image courtesy of Kaazing)

Ultimately, to provide full duplex communication on top of the HTTP half duplex architecture, many of today's solutions use two connections: one for the downstream and one for the upstream. The maintenance and coordination of these two connections introduces significant overhead in terms of resource consumption and increased complexity. WebSocket was developed to solve these issues. The new HTML5 feature provides interesting properties such as:

- a full duplex communication over a single TCP socket
- an overhead reduction
- less traffic: no need for polling
- standard and secure connections over ws:// and wss:// prefixes: A standard websocket URL can be:

ws://example.com

The figure 2.2 shows the reduction in latency. Once the connection is established, messages can flow between server and client. As the connection remains open, there is no need to send another request to the server.

2.1.3 Websocket protocol

To establish a WebSocket connection, the client and server upgrade from the HTTP protocol to the WebSocket protocol during their initial handshake. Another attractive aspect is their HTTP-compatible handshake. This means HTTP servers can share their default HTTP (80) and HTTPS (443) ports with a WebSocket server. After the handshake, a full tcp socket communication with some overhead is available. To establish a Websocket connection, the client initiates a connection to the server as with HTTP, but then requests the server upgrade from the HTTP protocol to the Websocket protocol.

The handshake from the client is as follows:

```
GET /ws HTTP/1.1
Host: example.org
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Upgrade: WebSocket
Origin: http://example.org
Sec-WebSocket-Version: 13
```

The handshake from the server is as follows:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Once the Websocket connection is established, data can be exchanged between client and server. Data can either be text frame encoded as UTF-8 or binary data. Main fields of the data framing represented on the figure 2.3 are:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

Figure 2.3: Websocket data framing

- FIN: Indicates that this is the final fragment in a message.
- RSVx: reserved (0)
- Opcode: meaning of the payload:
  - 0x0: continuation frame
  - 0x1: text frame (the payload is text data encoded as UTF-8)
  - 0x2: binary frame (the payload is binary data)
  - 0x8: connection close
  - 0x9: ping (can be used as keep-alive mechanism)
  - 0xa: pong (can be used as keep-alive mechanism)
- Mask: defines whether the payload data is masked or not

- payload length (in bytes):
  - 0 - 125: payload length
  - 126: the following 2 bytes represent the payload length
  - 127: the following 8 bytes represent the payload length
- Masking-key: used to unmask the payload
- Payload data: data

2.1.4 Architecture of a Websocket communication

Websocket communication involves several clients connected to the same websocket server. All messages from browsers are sent to the server, where the server manages them. It can, for instance, decide to send a message received from one client to another, or to broadcast all messages received to all clients connected.

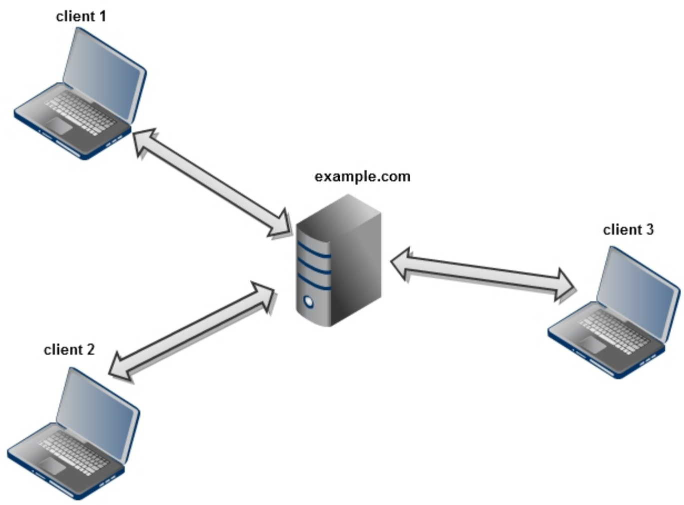


Figure 2.4: Example of websocket communication

For example, let's say that there is an existing websocket server: **example.com** which is listening on port 80. A client can open a connection, receive and send messages to this server with these few lines of javascript:

Listing 2.1: Javascript Websocket Hello World

```
var ws = new WebSocket("ws://example.com");
ws.onopen = function(evt) {
   alert("Connection open");
   ws.send("Hello")};
};
ws.onmessage = function(evt) {
   alert( "Message received: " + evt.data);
};
ws.onclose = function(evt) {
   alert("Connection closed.");
};
```

2.1.5 The mbed websocket library

A websocket client library has been developed in order to exchange data between an mbed and a server over a websocket communication. The library can be used over an ethernet connection, or a wifi connection using a Roving Networks wifly module. The library uses existing libraries such as the wifi module library or the TCP socket library, part of the TCP/IP stack (port of lwIP for mbed). For instance, the method which connects an mbed to a websocket server over wifi is:

Listing 2.2: Connection to a websocket server

```
// open a socket on a specific port
sprintf(cmd, "open %s %s\r\n", ip_domain.c_str(), port.c_str());
wifi->send(cmd, "OPEN");
```

```
//send websocket HTTP header
sprintf(cmd, "GET /%s HTTP/1.1\r\n", path.c_str());
wifi->send(cmd);
sprintf(cmd, "Host: %s:%s\r\n", ip_domain.c_str(), port.c_str());
wifi->send(cmd);
wifi->send("Upgrade: websocket\r\n");
wifi->send("Connection: Upgrade\r\n");
wifi->send("Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==\r\n");
sprintf(cmd, "Origin: http:%s:%s\r\n", ip_domain.c_str(), port.c_str());
wifi->send(cmd);
if (!wifi->send("Sec-WebSocket-Version: 13", "s3pPLMBiTxaQ9kYGzzhZRbK+xOo="))
   return false;
```

2.2 Realtime Data Streaming from sensors

Mbed wanted to demonstrate that it was possible to access sensor data all over the world using an mbed board and HTML5 websockets. For this, two prototype boards has been developed with different sensors:

- **accelerometer board** with a three axis accelerometer
- **environmental board** with:
  - light sensor
  - pressure sensor
  - microphone

The idea behind this project is that the sensor data has to be available everywhere as quickly as possible. That's why we wanted to involve smartphones connected to the Internet over 3G. To achieve this, we used QR codes containing the URL of a webpage which receives the sensor data. Each board has a different QR code. The steps to access data for a user are:

- Place the objects (mbed, sensors, QR code associated to this board) where desired
- Scan the QR code with a smartphone to see its data displayed
- the graph can then be added to a central **dashboard**

2.2.1 Architecture: mbed boards, websocket server, browsers

We can distinguish three different parts:

- a websocket server
- mbed boards which are connected to this websocket server
- browsers which are connected to the same server:
  - **smartphone webpages**: displays a unique graph containing sensor data of a specific board. This webpage is accessed by reading the QR code associated to the board. A user can then decide to **add** or **remove** this graph to or from a central dashboard
  - **dashboard**: central webpage showing graphs of different boards added by users

All mbed boards send messages in JSON format to the websocket server containing sensor data. The websocket server broadcasts to all client connected all messages received, so all browsers connected to this server are able to receive data from sensors. Browsers can then ignore or not the messages received.

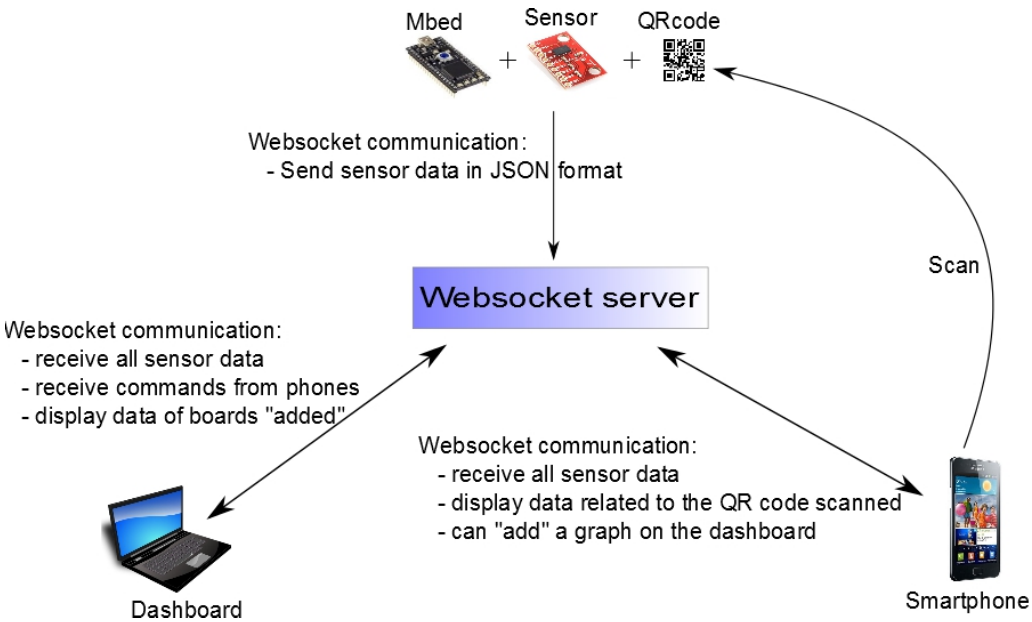


Figure 2.5: Realtime data streaming project architecture

2.2.2 Websocket server: Tornado

The websocket server used for this project is based on the Tornado Framework. This framework is written in Python, and it's a non blocking web server. It uses **epoll** which permits the handling of thousands of connections and so provides realtime features. Some interesting features are, for instance:

- http server
- template language
- mySQL client wrapper
- websocket server

Listing 2.3: Launching of an HTTP server with Tornado

```
application = tornado.web.Application([
    (r'/ws/(.*)/(.*)', WSHandler),
], **settings)

if __name__ == "__main__":
    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(80)
    tornado.ioloop.IOLoop.instance().start()
```

These few lines launch a http server associated with a handler (WSHandler). All requests having a path matching the regular expression **/ws/*/*** will be processed by WSHandler. The first argument of the path represents a channel and the second one the connection mode. This means that to open a connection with this server, a client has to join the server at the address:

ws://sockets.mbed.org/ws/<channel>/<mode>

The WebSocket server is divided into channels. All clients can send and receive messages over the same channel according their connection mode. There are 3 connection modes:

- **write-only (wo)**: the client can write on a certain channel but cannot receive messages
- **read-only (ro)**: the client can read messages on a certain channel but cannot write messages
- **read-write (rw)**: the client can read and write messages over a channel

When the server receives a message from a client in a certain channel which is not in **ro** mode, it will broadcast the message to all clients connected to this channel which are in **rw** or **ro** mode. This mechanism is done in WShandler which inherits from tornado.websocket.WebSocketHandler. The websocket protocol is implemented in the base class. Different methods can be overrided in WShandler, called on different events:

- a connection has been opened
- a message is received
- a connection has been closed

Listing 2.4: Broadcast messages received

```
class WSHandler(tornado.websocket.WebSocketHandler):
    def open(self, chan, mode):
        self.channel = chan
        self.mode = mode
        if mode == 'rw' or mode == 'ro' or mode =='wo':
            if not subscribers.get(self.channel):
                subscribers[self.channel] = []
            subscribers[self.channel].append(self)
            logging.warning("New Subscribers: chan %s, mode: %s"%(chan, mode))

    def on_message(self, message):
        for client in subscribers.get(self.channel, []):
                if client.mode != 'wo':
                  client.write_message(message)
```

2.2.3 Mbed boards

Hardware

The hardware is very simple to setup as mbed has been designed with quick prototyping in mind. The main parts are:

- one mbed
- a wifi module
- sensors

A wifi module has been chosen to connect the mbed to the internet and access the websocket server. I wanted a wifi module easy to use and easy to connect to the mbed. I chose a wifi module from Roving Networks which integrates a full TCP/IP stack. The communication between the mbed and the wifi module is established over a serial port. This module is integrated on a breakout board so that we just need wires to connect with the mbed.

Concerning sensors, all are also very easy to use and to connect. The 3-axis accelerometer and the pressure sensor are connected to the mbed over SPI. Libraries for these sensors have already been developed by the mbed community. The light sensor is connected over a GPIO, as is the microphone.

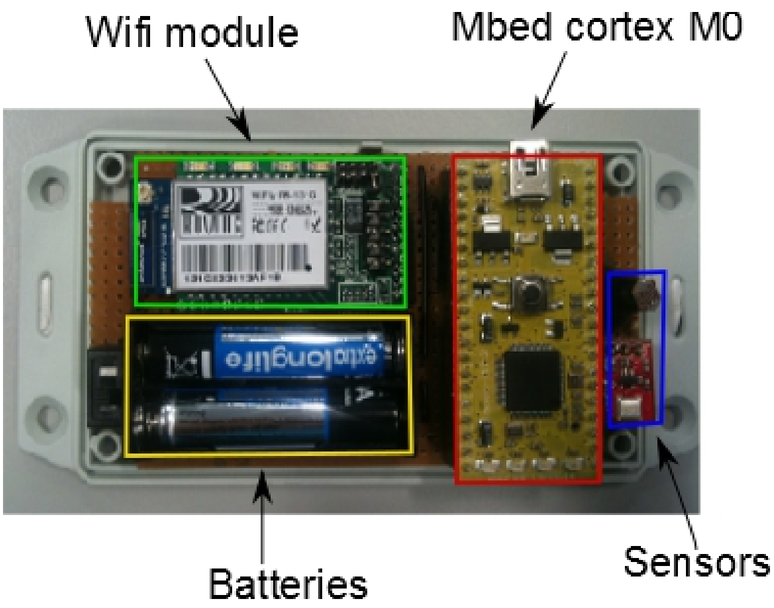


Figure 2.6: Environmental board

Software

The program running on the mbed has to:

- connect to a network
- connect to the websocket server
- in a loop:
  - read sensor data
  - send them over websocket to the server

Listing 2.5: Streaming data code example

```
// wifly instance to connect the wireless network
Wifly wifly(p9, p10, p30, "network", "password", true);

// Websocket instance to send sensor data to the websocket server: we use the channel "sensors" in write-only mode
Websocket ws("ws://sockets.mbed.org/ws/sensors/wo", &wifly);

int main() {
    char json_str[100];
    int press;
    double temp;
    int light;
    unsigned short mic;

    // join the wireless network
    wifly.join();

    // connection to the websocket server
```

```
    ws.connect();

    while (1) {
        wait(0.1);

      //pressure
      press = scp1000.readPressure();

      //temperature
      temp = scp1000.readTemperature();

      //light
      light = light_pin.read_u16()/480;

      //microphone
      mic = readMicrophone();

      //format data in a JSON string and sent it
      sprintf(json_str, "{\"id\":\"wifly_env\",\"p\":\"%d\",\"t\":\"%d\",\"l\":\"%d\",\"m↩
          \":\"%d\"}", (int)press, (int)temp, (int)(140 - light), mic);
      ws.send(json_str);
    }
}
```

2.2.4 Browsers

In the process of developing clean and minimalist webpages for computer and smartphone use, I needed a graph for data that scrolled smoothly in real time. For this, I chose to use another new HTML5 feature, the canvas element. Javascript code can access this drawable region through a complete API. The canvas element can be for instance used to display realtime graphs or to build games and animations.

There are two different webpages:

- **smartphone webpages**: display only the graphed data from the corresponding devices, added via QR code. Two buttons can be used to add or remove a specific graph from the dashboard webpage
- **dashboard**: display all graphs added by users

Realtime graphs

I used Smoothie Charts which is a javascript library to stream data using a canvas element. Smoothie Charts is a small charting library designed for live streaming data. The idea is to display in realtime sensor data in a chart refreshed everytime that we receive a new data.

A QR code associated to each mbed board

A smartphone has to scan a QR code to access sensor data of a specific board. The difference between all mbed boards is made over a variable passed to a PHP webpage:

- **accelerometer board**: sensors.php?id=wifly_acc
- **environmental board**: sensors.php?id=wifly_env

According to the parameter, a smartphone webpage will then listen only messages from a specific board.

Websocket and JSON messages

Several messages are exchanged between clients and server.

- from mbed boards:

{"id"="wifly_acc", "ax"="x_axis_value", "ay"="y_axis_value", "az"="z_axis_value"}
{"id"="wifly_env", "l"="light_value", "t"="temperature_value", "p"="pressure_value", "m"="microphone_value"}

- from smartphones:

{"id"="acc_add"}
{"id"="env_add"}
{"id"="acc_clean"}
{"id"="env_clean"}

Messages sent by mbed boards contain sensor data. When a client receives such a message, it can refresh a realtime graph showing the sensor data against time.
Messages sent by smartphones control graphs on the dashboard webpage. For instance, if a user scans the QR code of the accelerometer board and presses the add button, the message

{"id"="acc_add"}

will be sent. The dashboard will then display the corresponding graph.

Architecture of webpages

The architecture of smartphone webpages is quite simple:

- Connection to the websocket server
- When a message is received:
  - check the validity of the message
  - if the message is a request to add a graph on a dashboard:
    * ignore it
  - if the message is coming from a board:
    * if the **id field** matches the argument passed to the PHP webpage, refresh the graph
    * otherwise, ignore the message

The architecture of the dashboard is a little more complicated because it has to take into account messages from boards and from smartphones:

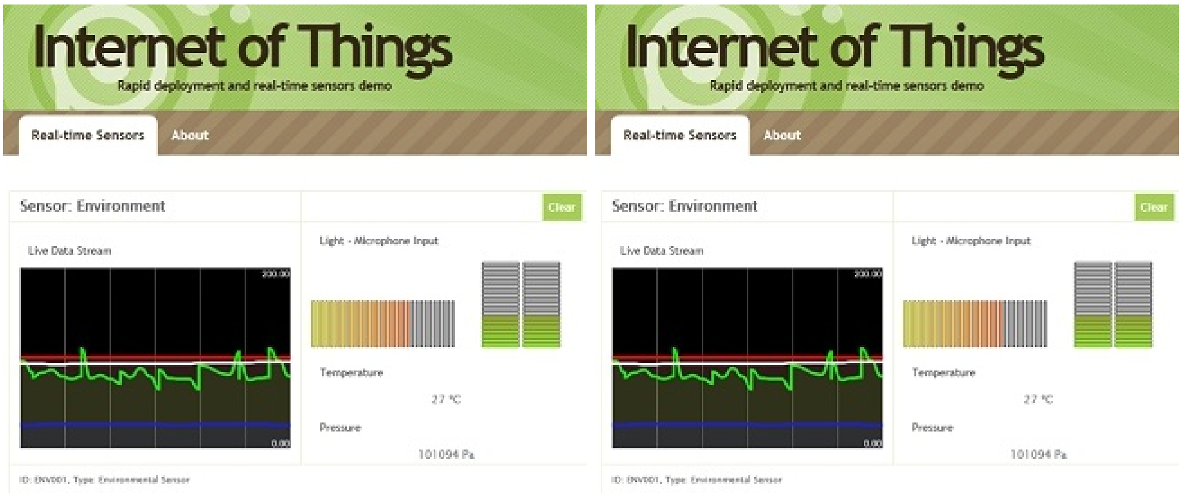
- Connection to the websocket server
- When a message is received:
  - check the validity of the message
    * if the message is a request to add a graph on a dashboard:
      · update the Document Object Model (DOM) using the jQuery javascript library
      · display the graph associated to this request
    * if the message is coming from a board:
      · if the board has been previously added, refreshes a specific graph according to the **id field**

Listing 2.6: Manipulation of websocket messages by the dashboard

```
    //Websocket instance connected on the same subnetwork as the mbed boards
websocket = new WebSocket('ws://sockets.mbed.org/ws/sensors/rw');

websocket.onmessage = function (evt) {
        var json_sensor = jQuery.parseJSON(evt.data.toString());
        if(json_sensor.id == "wifly_acc")
        {
            acc_x.append(new Date().getTime(), json_sensor.ax);
            acc_y.append(new Date().getTime(), json_sensor.ay);
            acc_z.append(new Date().getTime(), json_sensor.az);

            refreshBall();
        }
        else if(json_sensor.id == "wifly_env")
        {
            light.append(new Date().getTime(), json_sensor.l);
            temp.append(new Date().getTime(), json_sensor.t);
            press.append(new Date().getTime(), json_sensor.p / 1000);
            mic.append(new Date().getTime(), json_sensor.m);

            refreshEnv();
        }
        else
            manipulateDOM(json_sensor.id);
    };
```

As expected, when a websocket message is received the corresponding graph is updated. If the "id" field on the message is different from wifly_acc or wifly_env, it's probably a message coming from a smartphone indicating to the dashboard to display or not a specific graph.

Results



(a) Dashboard webpage (b) Smartphone webpage

Figure 2.7: Webpages which access sensor data

On the dashboard, we can see that the environmental board has been added:

- realtime graph available (10 Hz): light, pressure, temperature and microphone data
- more information are available on the right: other representations of sensor data

On the smartphone webpage, we can observe the live graph containing sensor data. On this webpage, we can manage the dashboard with the "add" and "clear" buttons.

2.3 Remote Procedure Call over Websockets

After the realtime data streaming from sensors project demonstrated the power of HTML5 in embedded systems, mbed wanted to use websockets to design an RPC (Remote Procedure Call) mechanism. The idea is to enable users to call a specific method on a specific mbed. Both of these mbeds are connected to the Internet.

2.3.1 Architecture: mbed boards, websocket server

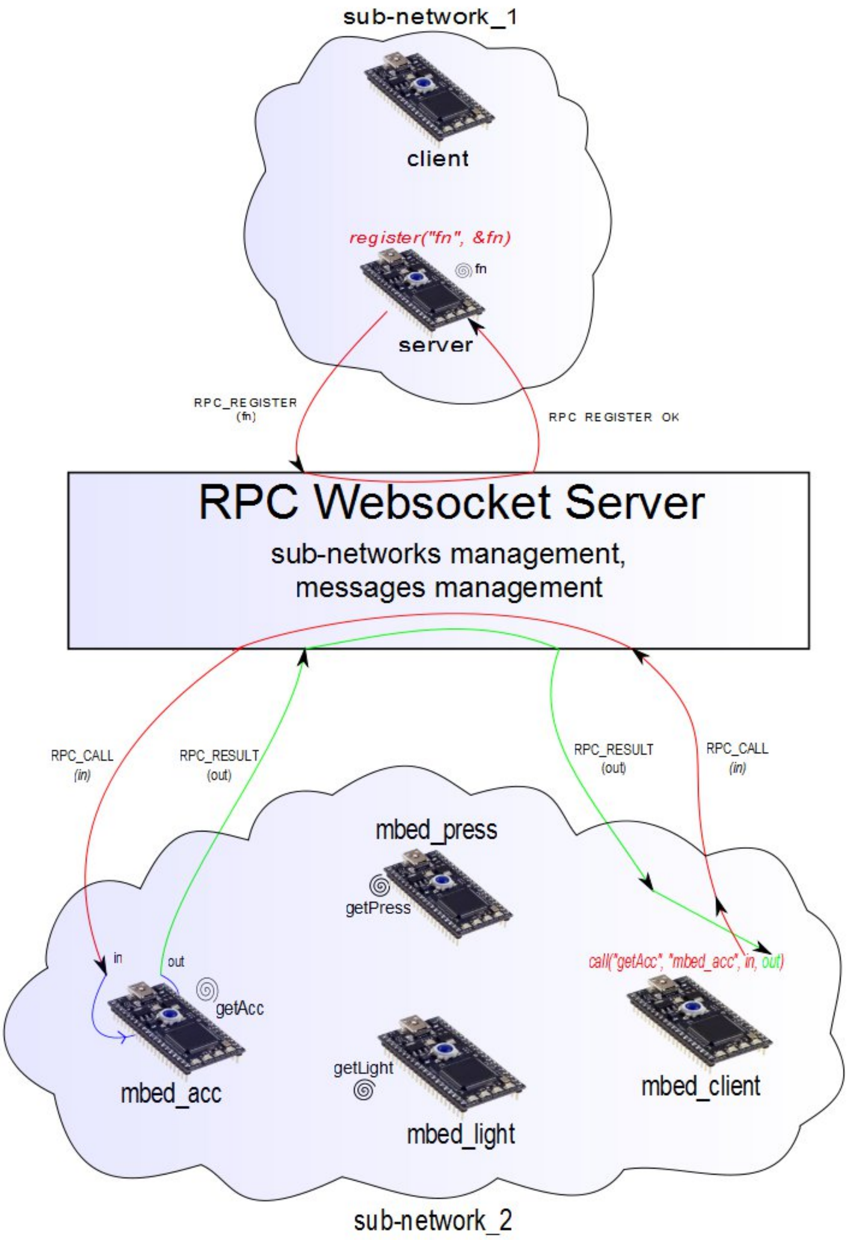


Figure 2.8: RPC architecture

On the previous diagram, two main elements can be distinguished:

- Two different sub-networks. On each sub-network, all mbeds can share methods or execute a non-local one
- An RPC Websocket server which is responsible for managing all sub-networks and all messages exchanged. This server is written in Python and uses Tornado

As with the realtime data streaming from sensors project, each mbed will belong to a specific subnetwork and will be identified by a specific name. This identification is made over the URL used to connect the websocket server as shown in the figure 2.9.

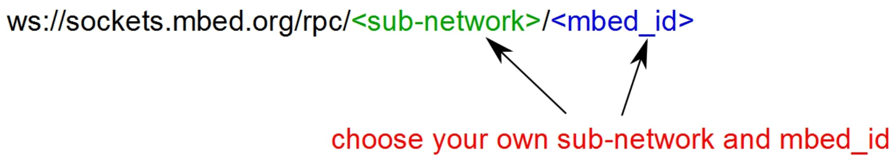


Figure 2.9: RPC: mbed identification on a subnetwork

Different steps are required to execute a distant method:

- connect several mbeds to the same subnetwork over a websocket communication
- all mbeds can register methods: an mbed has to register on the server one or several methods before than others can call them
- an mbed connected over the same subnetwork as the previous mbed can call the registered method
- the distant mbed will execute the method and return the results
- the local mbed can then handle the result of the distant method

2.3.2 Protocol

The RPC mechanism relies on messages exchanged. On these messages, we need to specify:

- the source of the message
- the destination of the message
- message name:
  - CALL: to call a distant function
  - RESULT: this message contains the result of a previous CALL
  - REGISTER: used to register a specific method to the websocket server
  - INFO_METHODS: a client can have access to all methods registered from a specific client
  - ERROR: an error has been detected
- message ID: random number used to identify a transaction. For a valid RPC transaction, the request and the answer have to have the same message ID.

These previous fields are common to all messages exchanged. After for each message, other fields can be present:

- CALL:
  - method name: distant method which will be called
  - params: parameters for this method
- RESULT:
  - result: result of the function called remotely
- REGISTER:
  - fn: on the network, a distant method will be identified by this field
- ERROR:
  - cause: cause of the error:
    - JSON_PARSE_ERROR: error in the json format
    - JSON_RPC_ERROR: error in the rpc message format
    - METHOD_NOT_FOUND: the distant method called has not been registered
    - CLIENT_NOT_CONNECTED: the client which must execute the distant method is not connected

2.3.3 Signature of methods handled

A user wants to be able to register and then call a function of his choice. So a function can take all kinds of input arguments and return whatever type. As messages are exchanged in JSON format, I decided to use this signature for all methods handled by the RPC mechanism:

void fn(MbedJSONValue& in, MbedJSONValue& out)

A MbedJSONValue object is the object associated with a string in JSON format. This class can handle parameters such as:

- booleans
- integers
- doubles
- strings
- arrays of MbedJSONValue
- MbedJSONValue object inside another one

This class has been designed in order to simplifies at maximum the creation of such objects:

Listing 2.7: MbedJSONValue manipulation

```
#include "mbed.h"
#include "MbedJSONValue.h"
#include <string>

int main() {
   MbedJSONValue demo;

   const  char * json = "{\"my_array\": [\"demo_string\", 10], \"my_boolean\": true}";

   //parse the previous string and fill the object demo
   parse(demo, json);

   std::string my_str;
   int my_int;
   bool my_bool;

   my_str = demo["my_array"][0].get<std::string>();
   my_int = demo["my_array"][1].get<int>();
   my_bool = demo["my_boolean"].get<bool>();

   printf("my_str: %s\r\n", my_str.c_str());
   printf("my_int: %d\r\n", my_int);
   printf("my_bool: %s\r\n", my_bool ? "true" : "false");
}
```

2.3.4 Core of the RPC mechanism: MbedJSONRpc

Register a method

To register a method, a user has to invoke:

Listing 2.8: Register a method

```
template<typename T> RPC_TYPE registerMethod(const char * public_name, T * obj_ptr, void (T↩
    ::*fn)(MbedJSONValue& val, MbedJSONValue& res))
```

This method is responsible to register on the rpc websocket server the specified method:

- Send a websocket message to the server containing a MSG_REGISTER request:
  - from: mbed_id
  - to: server
  - msg: REGISTER
  - fn: public_name
- Wait a response from the server (if the request is successful: REGISTER_OK)
- Fill two local arrays:
  - callback: contains function pointers of methods registered
  - name: contains identifiers of methods registered

After this step, a distant mbed will be able to call the method registered identified by "public_name".

Call a registered method

To call a distant method, a user has to invoke:

Listing 2.9: Call a distant method

```
RPC_TYPE MbedJSONRpc::call(const char * fn, const char * dest, MbedJSONValue& in, ↩
    MbedJSONValue& out)
```

This method, in order to call a distant method, requires several steps:

- Serialize the in MbedJSONValue object in order to be sent over websockets
- Send a websocket message to the server containing a MSG_CALL request
  - from: mbed_id
  - to: dest
  - msg: CALL
  - fn: fn
  - params: "in" serialized
- Wait a MSG_RESULT message from the server containing the result of the distant method
- Convert the JSON string contained in the "result" parameter into a MbedJSONValue
- Fill the "out" reference with the previous MbedJSONValue

After this step, a user can handle the result of the distant method contained in the "out" reference.

Listen for incoming request

To listen incoming requests, a user has to invoke:

Listing 2.10: Listen incoming requests

```
void MbedJSONRpc::work()
```

This method does the following steps:

- Tries to read a websocket message
- If a message is available:
  - Check the validity of the message
  - Convert the "params" field in a MbedJSONValue object
  - Execute the registered method with the previous in argument. At the same time, the method will fill the "out" parameter containing the result
  - Serialize the result of the method executed
  - Send a MSG_RESULT websocket message to the server;
    * from: mbed_id
    * to: msg["from"]
    * msg: RESULT
    * res: "out" parameter serialized

2.3.5 Mbed boards

Hardware

The only requirement for the hardware is to have access to the internet so a wifi module or ethernet jack is required.

TODO: photo

Software: mbed which registers a method

I will demonstrate the main program running on an mbed which has an accelerometer connected. The idea is to remotely access this accelerometer's values from another mbed.

Listing 2.11: Client which registers a method

```

// accelerometer
ADXL345 accelerometer(p5, p6, p7, p8);

//wifi module
Wifly wifly(p9, p10, p17, "network", "password", true);

//websocket: configuration with sub-network = sensor and mbed_id = mbed_acc
```

```
Websocket webs("ws://sockets.mbed.org/rpc/sensor/mbed_acc",&wifly);

//RPC object attached to the websocket server
MbedJSONRpc rpc(&webs);

//Acc class.
class Acc {
public:
    Acc() {};
    void getAcc(MbedJSONValue& in, MbedJSONValue& out) {
        int readings[3] = {0, 0, 0};
        accelerometer.getOutput(readings);
        out[0] = readings[0];
        out[1] = readings[1];
        out[2] = readings[2];
    }
};

//Instance of Acc. The method getAcc of this object will be registered
Acc acc;

int main() {
    RPC_TYPE t;

    //Accelerometers init
    accelerometer.init();

    //join the network
    wifly.join();

    //connect the websocket server
    webs.connect();

    //register the acc method and wait for incoming methods
    if((t = rpc.registerMethod("getAcc", &acc, &Acc::getAcc)) == REGISTER_OK) {
            //wait for incoming CALL requests
            rpc.work();
    }

}
```

Software: mbed which calls a distant method

Listing 2.12: Client which calls a distant method

```
//websocket over ethernet: configuration with sub-network = samux and mbed_id = mbed_acc
Websocket webs("ws://sockets.mbed.org/rpc/sensor/mbed_client");

//RPC object attached to the websocket server
MbedJSONRpc rpc(&webs);

int main() {
    RPC_TYPE t;

    //in: argument for the distant method (here empty)
    //out results of the distant method (accelerometers values)
    MbedJSONValue in, out;

        //connect the websocket server
    webs.connect();

    //CALL getAcc on mbed_acc
    if ((t = rpc.call("getAcc", "mbed_acc", in, out)) == CALL_OK) {
        printf("acc_x: %d\r\n", out[0].get<int>());
        printf("acc_y: %d\r\n", out[1].get<int>());
        printf("acc_z: %d\r\n", out[2].get<int>());
    }
}
```

2.3.6 Websocket server: Tornado

The server uses the same framework used for the realtime streaming data project: Tornado. The server is responsible for managing all subnetworks and messages exchanged over a same subnetwork.

The main data structure in order to represent the state of the whole network is a dictionnary:

- Each key represents a specific subnetwork
- A value associated to a subnetwork is another dictionnary:
  - Each key represents one mbed connected
  - Each value is a dictionnary containing
    * a reference on the mbed
    * an array containing methods registered by this mbed

```
{ Subnetwork1: {  mbed_id1:  { id: ref_object, methods: [fn11, fn12,...]},
                  mbed_id2:  { id: ref_object, methods: [fn21, fn22,...]}
                                         ⋮
                                         },
  Subnetwork2: { ........................................... },
                                         ⋮
}
```

Figure 2.10: Data structure for the RPC websocket server

The websocket server does the following:

- If a new mbed connects a specific subnetwork:
  - Update the dictionnary containing the state of the network by adding an entry on the subnetwork dictionnary
- If the server receives a MSG_REGISTER request:
  - Update the dictionnary containing the state of the network by adding the new distant method in the array associated to the mbed
- If the server receives a MSG_CALL request:
  - Check that the request comes from an mbed on the same subnetwork as the distant mbed
  - Check that the method called is registered in the array associated to the called mbed
  - If the two previous steps are sucessful, send a MSG_CALL message to the distant mbed
- If the server receives a MSG_RESULT message:
  - Check that the mbed which will receive the result of the distant method is on the same subnetwork
  - Forward the message to the mbed which called the distant method

2.4 Conclusion

For Mbed the Internet of Things is a very important field. We feel that in the future more and more devices will be connected to the cloud. Examples of applications could include:

- smart buildings
- medical devices
- factories

The first project shows that it was possible to stream data sensor on realtime graphs accessible all over the world. The Remote Procedure Call project provides a means to execute a method on a distant mbed. This can be useful for instance in critical places where people cannot go frequently. The common point of these two projects is the use of a websocket communication, a new feature of HTML5. This new way of communication enables realtime capabilities with a bidirectionnal communication between client and server. It was quite interesting to develop projects using websocket which the protocol was in development. For instance, at the beginning, the data framing was not at all the same as now. It was much more simple: data frames started with 0x00 and ended with 0xff. The overhead was reduced but the final protocol provides more capabilities such as: raw binary can be sent, a ping/pong mechanism, a packet can be segmented. This change in the protocol led to tricky situations where the browser has been updated with the new protocol and not the websocket server. So that the browser and the server were not compatible.

After having worked on the Internet of Things, Mbed wanted to explore USB capabilities on the two mbed boards.

Chapter 3

Universal Serial Bus

3.1 Inside the USB bus

3.1.1 USB overview

The Universal Serial Bus (USB) is the most widely used bus in today's computer. USB has particularly been designed to standardize connections between the computer and peripherals. For instance, keyboards, printers, scanners, disk drives or cameras can use the same bus to exchange data with a computer. USB has effectively replaced a variety of earlier interfaces, such as serial or parallel ports. The USB bus provides several benefits such as the same interface for many device, the hot pluggable capability which allows a user to connect and disconnect a USB device whenever he wants, or the automatic configuration which is the capacity of the operating system to load a specific driver according to the device connected. Another useful benefit is that the USB interface provides power supply (5V) that can be used by a device so long as it doesn't require more than 500 mA.

USB version 1.0 supported two speeds, a full speed mode of 12Mbits/s and a low speed mode of 1.5Mbits/s. USB 2.0, which is the most widely version of USB, can reach 480Mbits/s. The 480Mbits/s is known as High Speed mode. USB version 3.0 specifies a maximum transmission speed of up to 5Gbits/s (known as SuperSpeed), but few products support USB 3.0 at present.

USB 1.0	low-speed full-speed	1.5Mbits/s 12Mbits/s
USB 2.0	high-speed	480Mbits/s
USB 3.0	super-speed	5Gbits/s

Table 3.1: USB speeds

The Universal Serial Bus is host controlled. However, there can only be one host per bus and the host is responsible for undertaking all transactions. Considering this restriction, two devices cannot exchange information without one being the host. Nevertheless, the On-The-Go specification, which is part of the USB 2.0 standard, has introduced a Host Negotiation Protocol, allowing two devices to negotiate for the role of host. With this specification, we can imagine a camera exchanging data with a printer without the need for a computer.

3.1.2 Topology

The physical bus topology defines how USB devices are connected to the host. The USB network is implemented as a tiered star network with one host (master) and several devices (slaves). The topology looks like a tree. In order to increase the number of devices connected, a hub needs to be connected to the root port. This special hub and the root port are the first tier of the network. Furthermore, the USB network can support up to 127 external nodes but the number of tiers cannot exceed 7. The following diagram represents a possible connection architecture on a USB bus.

3.1.3 Endpoints and type of transfers

Devices have a series of buffers to communicate with the host. Each buffer will belong to an endpoint. An endpoint is a uniquely identifiable entity on a USB device, which is the source or terminus of the data that

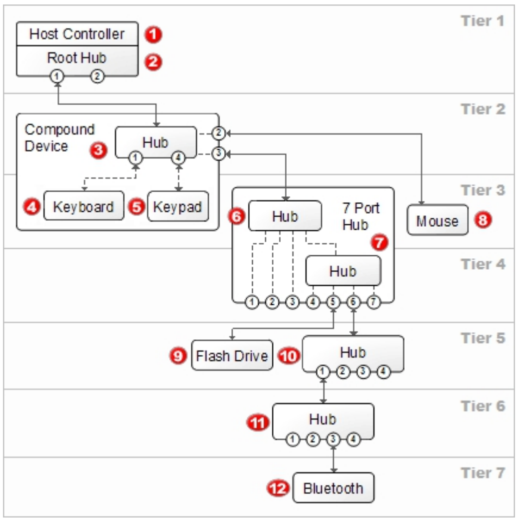


Figure 3.1: USB physical topology [1]

flows from or to the device. For instance, the mbed based on a NXP LPC1768 microcontroller has 32 physical endpoints whereas the mbed based on the NXP LPC11U24, has 10 physical endpoints. One logical endpoint represents two physical endpoints. Each physical endpoint has a specific direction: either OUT to receive data from the host or IN to send data to the host.

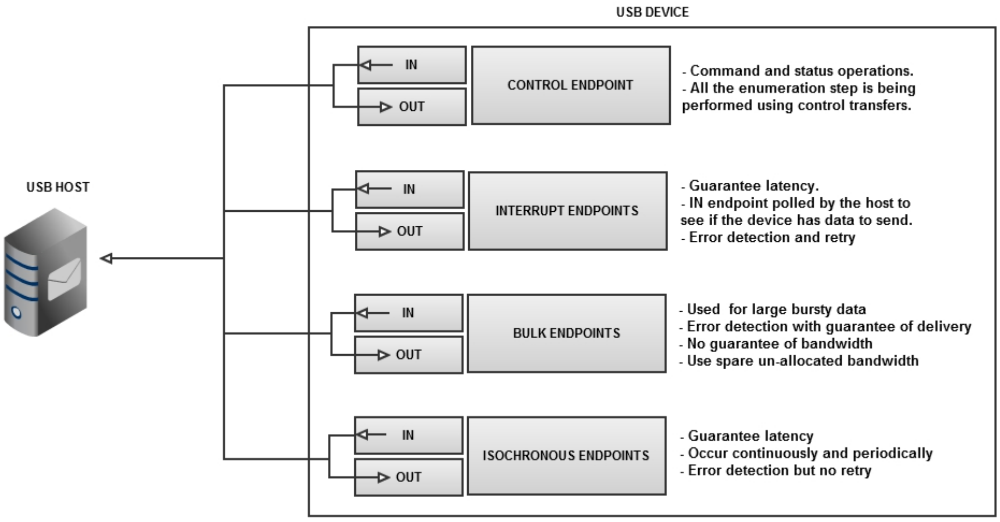


Figure 3.2: Different types of endpoints

There are four types of endpoints which correspond to different requirements according to the device:

- **Control endpoint**: All USB devices once connected perform an *enumeration step* wherein the host requests all capabilities of the device. Control transfers are used to enumerate a device
- **Interrupt endpoints**: Interrupts transfers are used when a device requires responsiveness. Typical applications would include keyboard and mouse. Users don't want a noticeable delay between pressing a key or moving a mouse and seeing the result on the screen. An interrupt transfer only occurs when the host polls the device. There is a guarantee that the host will request a data within a specified time interval
- **Bulk endpoints**: Bulk transfers are typically used to transfer large amount of data like files to a printer. A bulk transfer uses spare un-allocated bandwidth, so time is not critical with bulk transfers. Typical applications include printers, scanners or mass storage devices
- **Isochronous endpoints**: Isochronous tranfers are used for streaming and realtime data. For instance, audio and video streaming devices use isochronous transfers. Such devices need a guaranteed delivery rate for data but if an error occurs, the data is not re-transmitted

[1] reference: http://www.usblyzer.com

3.1.4 Packets exchanged

Common fields in a USB packet

Field	Length	Meaning
SYNC	low and full speed: 8bits	A packet starts with a SYNC pattern to allow the receiver bit clock to synchronise with the data.
	high speed: 32bits	
EOP		A packet ends with an End of Packet (EOP) field
PID	8bits	Packet ID
ADDR	7bits	The address field specifies which device the packet is designated for
ENDP	4bits	The endpoint number which the packet is designed for
CRC	5 or 16bits	Cyclic Redundancy Checks are performed on the data within the packet payload

Table 3.2: Fields in a USB packet

Packets

- **Token Packet**: To indicate the type of transaction to follow, USB uses token packets:

SYNC	PID	ADDR	ENDP	CRC5	EOP
8/32bits	8bits	7bits	4bits	5bits	

  There are three types of token packets:
  - IN: The host wants to read information.
  - OUT: The host wants to send information.
  - SETUP: Used to begin control transfers.

- **Data Packet**: Packets which contain the payload

SYNC	PID	DATA	CRC16	EOP
8/32bits	8bits	(0 - 1024) * 8bits	16bits	

- **Handshake Packet**: After a data stage, handshake packets are used to acknowledge data or to report errors

SYNC	PID	EOP
8/32bits	8bits	

  There are four types of handshake packets:
  - ACK: Packet successfully received.
  - NAK: The device temporary cannot send or received data
  - STALL: Endpoint is halted, or control pipe request is not supported.
  - NYET: No response yet from receiver (high speed only)

- **Start Of Frame Packet**: USB manages time in units called "frames" (USB 2.0 further added "microframes"), and uses frames/microframes to realize the concept of time. Each "frame" represents 1 ms, and each microframe represents 125 microseconds. When performing control, bulk or interrupt transfers with a peripheral device, there is little need to pay attention to the frames/microframes. However, when performing isochronous transfers, frames/microframes may need to be taken into consideration for proper synchronization with the system. For this reason, the host issues SOF (Start Of Frame) packets to the bus to indicate the starting point of each frame/microframe

SYNC	PID	Frame number	CRC5	EOP
8/32bits	8bits	11bits	5bits	

3.1.5 Enumeration

The host hub port is able to detect the attachment of the USB device and makes the host controller aware of the same. The host controller then starts communicating with the USB device (which could be a mouse, keyboard, flash drive etc.). This initial communication between the host and the device is termed as bus enumeration. Bus enumeration is the process through which the host learns about the capabilities of the device. Since any USB device can be connected to the host hub port at any time, bus enumeration becomes the essential first step of USB communication. This step is performed on the control endpoint (endpoint 0). So, all devices must at least have the control endpoint enabled. During this step, the host will assign a unique address to the device. All devices capabilities are transmitted in data structures called **descriptors**. Descriptors are arrays containing information like the device class, the number of endpoints used, the maximum length of these endpoints... There are several types of descriptors. Standards descriptors are: device descriptor, configuration descriptor, interface descriptor and endpoint descriptor. But, for each USB class, other descriptors can be found such as the HID (Human Interface Device) descriptor and the report descriptor for HID devices.

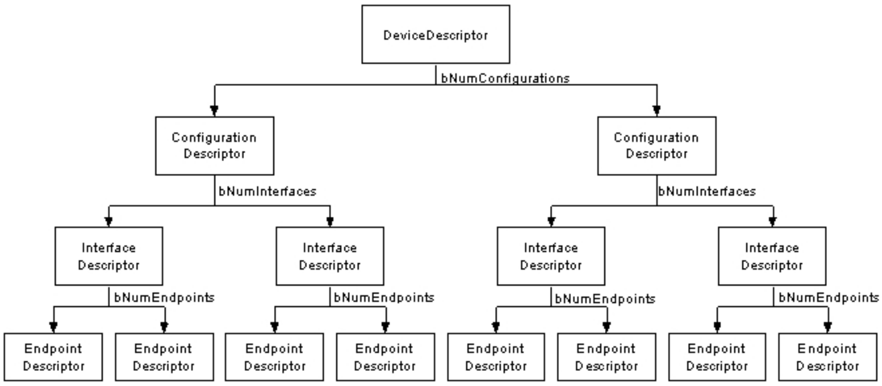


Figure 3.3: Descriptor architecture

A typical enumeration follows:

- **GET_DEVICE_DESCRIPTOR**: The host sends a get device descriptor request. The device replies with its device descriptor to report its attributes (Device Class, maximum packet size for endpoint zero)
- **SET_ADDRESS**: A USB device uses the default address (0) after reset until the host assigns a unique address using the set address request. The firmware writes the device address assigned by the host
- **GET_CONFIGURATION_DESCRIPTOR**: The host sends a get configuration request. The device replies with its configuration descriptor, interface descriptor and endpoint descriptor. The configuration descriptor describes the number of interfaces provided by the configuration, the power source (Bus or Self powered) and the maximum power consumption of the USB device from the bus. The Interface descriptor describes the number of endpoints used by this interface. The Endpoint descriptor describes the transfer type supported and the bandwidth requirements
- **SET_CONFIGURATION**: The host assigns a configuration value to the device based on the configuration information. The device is now configured and ready to be used

3.1.6 USB 2.0 transactions

A transfer consists of 1 or more transactions. Each transaction contains a token packet and may contain a data and/or handshake packet.

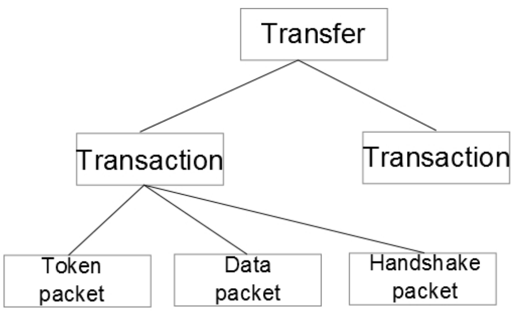


Figure 3.4: USB 2.0 transaction

I will illustrate a whole transfer with a control read transfer. This type of transfer is used by the host to request descriptors to a device on the control endpoint:

- **Setup transaction**:

  - SETUP token packet sent by the host
  - Data packet: the host sends a request concerning specific descriptor
  - Handshake packet: the device returns ACK
- **One or more data transaction(s)**:
  - IN token packet sent by the host
  - Data packet: the device sends the descriptor requested
  - Handshake packet sent by the host
- **Status transaction**:
  - OUT token packet sent by the host
  - Data packet: 0 length data
  - Handshake packet: the device returns the status

Concerning bulk or interrupts endpoints, a whole OUT transfer can be:

- **OUT transaction**:
  - OUT token packet sent by the host
  - Data packet: the host sends data
  - Handshake packet: the device returns the status (ACK, NAK or STALL)

Isochronous transfers are the same as bulk or interrupt transfers without the handshake packet. Occasional errors must be acceptable on isochronous transfers.

3.2 USB Device stack

A USB device stack has been developed to allow mbed users to design their own USB device or to use their mbed as USB peripheral such as a keyboard or a mouse. USB defines class code information that is used to identify a device's functionality. This class code is parsed by the USB host stack to load an appropriate device driver. Several classes have been implemented:

- HID (Human Interface Device)
- MSD (Mass Storage Device)
- Audio
- MIDI
- a subset of CDC (Communication Device Class) to provide a virtual serial port

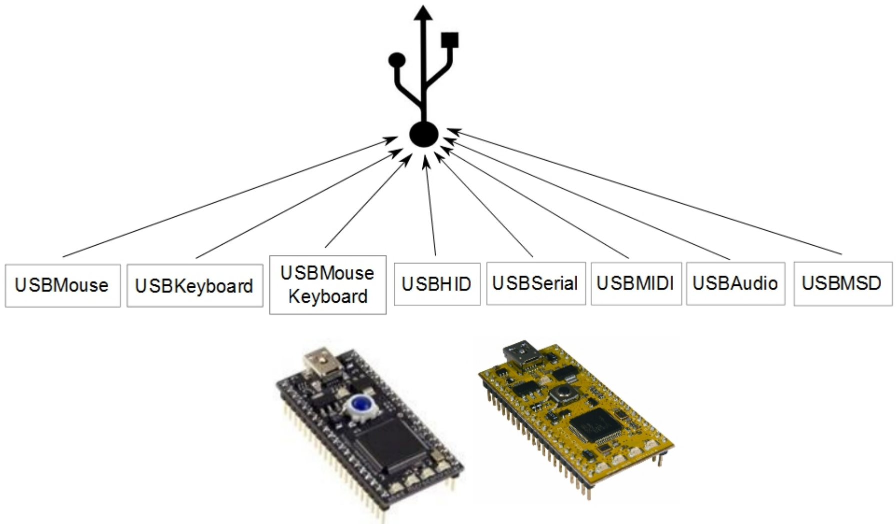


Figure 3.5: USB device stack capabilities

I will describe in this section the USB stack architecture and some USB classes such as HID, MSD, Audio and CDC.

3.2.1 USB Device stack architecture

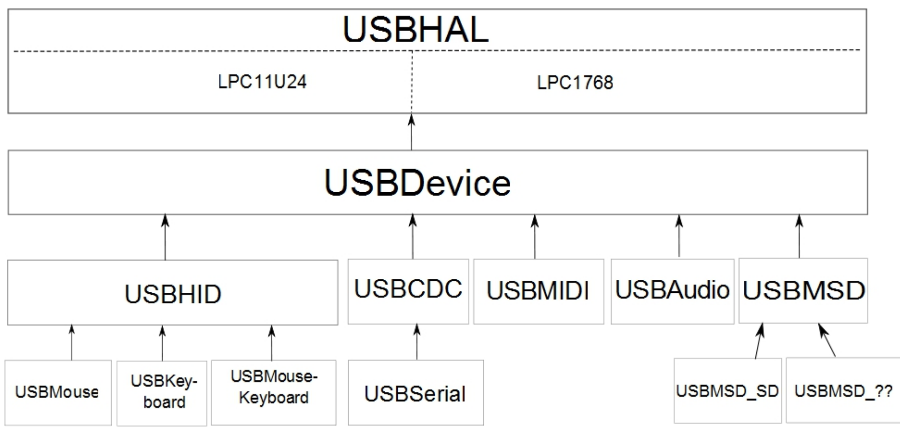


Figure 3.6: USB device stack architecture

- **USBHAL**: The USB hardware layer for the LPC11U24 and for the LPC1768. In this class, all low level methods are defined. There are USBHAL_LPC11U24.cpp and USBHAL_LPC1768.cpp which define functions defined in USBHAL.h. The right .cpp file is chosen according to a macro defined by the compiler: if a user is compiling a program for the LPC1768, the macro TARGET_LPC1768 is defined. If a user wants to compile a program for the LPC11U24, the macro TARGET_LPC11U24 is defined. Virtual functions are called on specific events to be treated by subclasses

- **USBDevice**: This layer is in charge of abstracting the hardware. At this level, no differences are made between the LPC11U24 and the LPC1768 concerning the target. USBDevice is in charge of performing the setup packet treatment (enumeration step is performed in this class) and provide an abstraction to handle the USB interface
- **USB class layer**:
  - USBHID: implements standard requests of the HID class specification. When a USBHID object is instantiated, the mbed is enumerated as a generic HID device so that raw data can be sent and received to and from a custom program running on the host side
  - USBCDC: implements a subset of the CDC class specification to allow the mbed to be recognized as a virtual serial port
  - USBMIDI: enables the mbed to send and receive MIDI messages to and from a computer
  - USBAudio: implements standard requests of the USB Audio class. The mbed is enumerated as a microphone and a speaker on the same device
  - USBMSD: implements the mass storage specification. This class is generic: a subclass has to implement some pure virtual functions defined in USBMSD to access a storage chip
- **USB device layer**:
  - USBMouse: used to emulate a mouse
  - USBKeyboard: used to emulate a keyboard
  - USBMouseKeyboard: used to emulate a mouse and a keyboard at the same time
  - USBSerial: used to emulate a virtual serial port
  - USBMSD_??: All users can implement their own class which inherits from USBMSD in order to access their storage chip

3.2.2 USBHAL: USB Hardware abstraction layer for the LPC11U24

This section describes the hardware layer implemented for the mbed LPC11U24. This microcontroller has a built-in USB 2.0 device controller. It also has 2 kB of RAM dedicated for USB operations.

USB EP Command/Status FIFO start

31	30	29	28	27	26	25 24 23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	Offset
A	R	S	TR	TV	R	EP0 OUT Buffer NBytes	EP0 OUT Buffer Address Offset	0x00
R	R	R	R	R	R	Reserved	SETUP bytes Buffer Address Offset	0x04
A	R	S	TR	TV	R	EP0 IN Buffer NBytes	EP0 IN Buffer Address Offset	0x08
R	R	R	R	R	R	Reserved	Reserved	0x0C
A	D	S	TR	RF TV	T	EP1 OUT Buffer 0 NBytes	EP1 OUT Buffer 0 Address Offset	0x10
A	D	S	TR	RF TV	T	EP1 OUT Buffer 1 NBytes	EP1 OUT Buffer 1 Address Offset	0x14
A	D	S	TR	RF TV	T	EP1 IN Buffer 0 NBytes	EP1 IN Buffer 0 Address Offset	0x18
A	D	S	TR	RF TV	T	EP1 IN Buffer 1 NBytes	EP1 IN Buffer 1 Address Offset	0x1C
A	D	S	TR	RF TV	T	EP2 OUT Buffer 0 NBytes	EP2 OUT Buffer 0 Address Offset	0x20
A	D	S	TR	RF TV	T	EP2 OUT Buffer 1 NBytes	EP2 OUT Buffer 1 Address Offset	0x24
A	D	S	TR	RF TV	T	EP2 IN Buffer 0 NBytes	EP2 IN Buffer 0 Address Offset	0x28
A	D	S	TR	RF TV	T	EP2 IN Buffer 1 NBytes	EP2 IN Buffer 1 Address Offset	0x2C
...								
A	D	S	TR	RF TV	T	EP4 OUT Buffer 0 NBytes	EP4 OUT Buffer 0 Address Offset	0x40
A	D	S	TR	RF TV	T	EP4 OUT Buffer 1 NBytes	EP4 OUT Buffer 1 Address Offset	0x44
A	D	S	TR	RF TV	T	EP4 IN Buffer 0 NBytes	EP4 IN Buffer 0 Address Offset	0x48
A	D	S	TR	RF TV	T	EP4 IN Buffer 1 NBytes	EP4 IN Buffer 1 Address Offset	0x4C

Figure 3.7: USB endpoints status array

To describe the state of each endpoints available, we need to allocate space in USB RAM as specified in the previous figure. Main fields of endpoints status array are:

- A: the endpoint is active or not
- NBytes: For OUT endpoints this is the number of bytes that can be received in this buffer. For IN endpoints this is the number of bytes that must be transmitted

- Address offset: represents the end of the address where will be stored data. The beginning of the address is specified in DATABUFSTART register

Main methods for USBHAL class ARE:

- Memory allocation for endpoints status list and endpoint 0 buffers. These initializations are done in the constructor of USBHAL::USBHAL
- To add more endpoints, space allocation has to be done. This is done in the method: USBHAL::realiseEndpoint
- To read a specific endpoint:
  - Fill the endpoint status list (active bit, address offset and NBytes). After this step, the endpoint is ready to receive data. This is done in USBHAL::endpointRead
  - When a data is received on the specific endpoint, an interrupt is raised. Software can read data in the buffer of this OUT endpoint. This is done in USBHAL::endpointReadResult
- To write on a specific endpoint:
  - Fill the endpoint status list (active bit, address offset and NBytes). Copy data in the buffer of the IN endpoint. After this step, the endpoint is ready to send data. This is done in USBHAL::endpointWrite
  - Wait until the writing has been effectively done. This is done in USBHAL::endpoint-WriteResult

When a specific event occurs, USBHAL calls virtual functions which can be overrided in a subclass to perform a custom treatment. Main callback functions are:

- SOF(int frameNumber): called on each Start Of Frame event (each millisecond)
- EP0setupCallback(): called when a setup packet is received
- EPx_OUT_callback(): called a data has been received on a specific endpoint
- EPx_IN_callback(): called when a data has been sent on a specific endpoint

These callback methods are called from the USB interrupt handler:

Listing 3.1: Main part of the USB interrupt handler

```
void USBHAL::usbisr(void) {
    // Start of frame
    if (LPC_USB->INTSTAT & FRAME_INT) {
        // Clear SOF interrupt
        LPC_USB->INTSTAT = FRAME_INT;

        // SOF event, read frame number
        SOF(FRAME_NR(LPC_USB->INFO));
    }

    // Endpoint 0
    if (LPC_USB->INTSTAT & EP(EP0OUT)) {
        // Clear EP0OUT/SETUP interrupt
        LPC_USB->INTSTAT = EP(EP0OUT);

        // Check if SETUP
        if (LPC_USB->DEVCMDSTAT & SETUP) {

            // Clear EP0IN interrupt
            LPC_USB->INTSTAT = EP(EP0IN);

            // Clear SETUP (and INTONNAK_CI/O) in device status register
            LPC_USB->DEVCMDSTAT = devCmdStat | SETUP;

            // EP0 SETUP event (SETUP data received)
            EP0setupCallback();
        } else {
            // EP0OUT ACK event (OUT data received)
            EP0_OUT_callback();
        }
    }

    if (LPC_USB->INTSTAT & EP(EP0IN)) {
        // Clear EP0IN interrupt
        LPC_USB->INTSTAT = EP(EP0IN);

        // EP0IN ACK event (IN data sent)
```

```
        EP0_IN_callback();
    }

    if (LPC_USB->INTSTAT & EP(EP1IN)) {
        // Clear EP1IN interrupt
        LPC_USB->INTSTAT = EP(EP1IN);
        epComplete |= EP(EP1IN);
        if (EP1_IN_callback())
            epComplete &= ~EP(EP1IN);
    }

    if (LPC_USB->INTSTAT & EP(EP1OUT)) {
        // Clear EP1OUT interrupt
        LPC_USB->INTSTAT = EP(EP1OUT);
        epComplete |= EP(EP1OUT);
        if (EP1_OUT_callback())
            epComplete &= ~EP(EP1OUT);
    }

    // Same for all following endpoints
    .
    .
    .
}
```

3.2.3 USBDevice: target abstraction and setup packet processing

Target abstraction

One of the purpose of this class is to provide an API in order to handle the USB interface very easily:

- init() to initialize the USB controller
- connect() to connect a device
- disconnect() to disconnect a device
- addEndpoint(int endpoint) to add a specific endpoint
- readEP(int endpoint) to read a specific endpoint
- writeEP(int endpoint) to write a specific endpoint

Setup packet processing

Two data structures has been defined in order to perform the control endpoint processing. The enumeration step is part of this treatment:

- SETUP_PACKET: to describe a setup packet
- CONTROL_TRANSFER: to describe a transfer on the control endpoint

Listing 3.2: Data structures for control endpoint processing

```
typedef struct {
    struct {
        uint8_t dataTransferDirection;
        uint8_t Type;
        uint8_t Recipient;
    } bmRequestType;
    uint8_t  bRequest;
    uint16_t wValue;
    uint16_t wIndex;
    uint16_t wLength;
} SETUP_PACKET;

typedef struct {
    SETUP_PACKET setup;
    uint8_t * ptr;        // pointer on a buffer to be sent
    uint32_t remaining;   // remaining bytes on the transfer
    uint8_t direction;    // direction: host > device or device > host
    bool zlp;             // zero length packet
    bool notify;
} CONTROL_TRANSFER;
```

For instance, a GET_DEVICE_DESCRIPTOR request is processed according to this scheme:

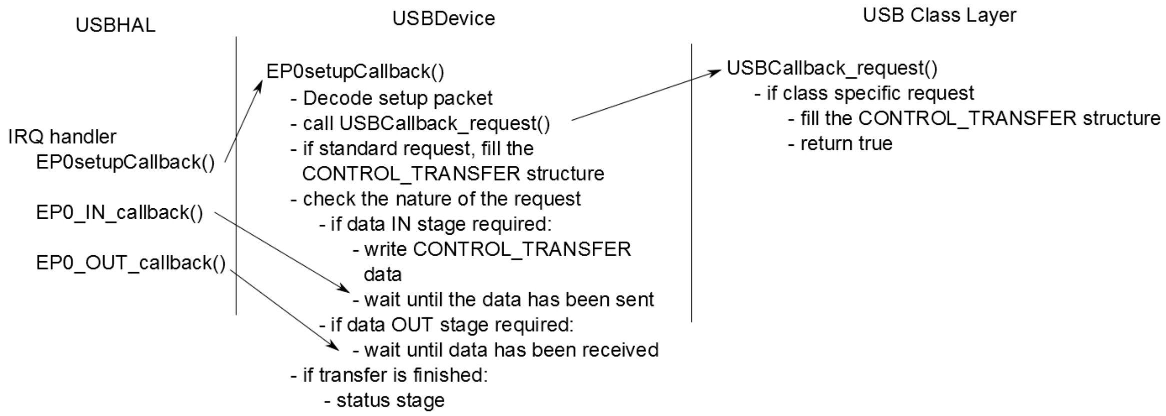


Figure 3.8: Setup packets processing

- Reception of a setup packet
- Decoding the setup in the SETUP_PACKET structure
- This is not a class specific request so the treatment is done in USBDevice class
- The request is GET_DEVICE_DESCRIPTOR:
  - Fill the CONTROL_TRANSFER structure containing in particular a pointer to the device descriptor
  - This request needs DATA IN stages (the device needs to send the descriptor)
  - Send data until CONTROL_TRANSFER.remaining == 0
  - Read the control endpoint for the status transaction

A basic device descriptor is quite common for USB devices. Importants fields are:

- bcdUSB which is the USB specification release number
- bDeviceClass specifies the device class
- bMaxPacketSize0 specifies the maximum packet size for endpoint 0

Listing 3.3: Device Descriptor

```
uint8_t * USBDevice::deviceDesc() {
    static uint8_t deviceDescriptor[] = {
        DEVICE_DESCRIPTOR_LENGTH,       // bLength
        DEVICE_DESCRIPTOR,              // bDescriptorType
        LSB(USB_VERSION_2_0),           // bcdUSB (LSB)
        MSB(USB_VERSION_2_0),           // bcdUSB (MSB)
        0x00,                           // bDeviceClass
        0x00,                           // bDeviceSubClass
        0x00,                           // bDeviceprotocol
        MAX_PACKET_SIZE_EP0,            // bMaxPacketSize0
        LSB(VENDOR_ID),                 // idVendor (LSB)
        MSB(VENDOR_ID),                 // idVendor (MSB)
        LSB(PRODUCT_ID),                // idProduct (LSB)
        MSB(PRODUCT_ID),                // idProduct (MSB)
        LSB(PRODUCT_RELEASE),           // bcdDevice (LSB)
        MSB(PRODUCT_RELEASE),           // bcdDevice (MSB)
        STRING_OFFSET_IMANUFACTURER,    // iManufacturer
        STRING_OFFSET_IPRODUCT,         // iProduct
        STRING_OFFSET_ISERIAL,          // iSerialNumber
        0x01                            // bNumConfigurations
    };
    return deviceDescriptor;
}
```

3.2.4 USBHID class

Introduction

HID is one of the most frequently used USB classes. The HID class was primarily defined for devices that are used by humans to control the operation of computer systems. Typical examples of HID devices include keyboards, mice, trackballs, and joysticks. HID devices may have knobs, switches, buttons, sliders, etc. The HID class is also frequently used for devices that may not require human interaction but provide data in a similar format to HID class devices. Since all operating systems have a built-in HID driver, it's easy to design a USB HID device which doesn't require any specific driver.

Data exchanged between the host and device resides in a structure called reports. The report format is flexible and can handle any format of data, but each report has a fixed size. This size is specified in a HID descriptor which is transmitted as part of the configuration descriptor. HID devices are required to provide a Report Descriptor which enumerates all data fields of a report. A HID device can have at most one interrupt IN endpoint and one interrupt OUT endpoint (and the control endpoint to perform the enumeration step).

For each field in the report, the Report Descriptor defines how many bits the field consumes, how often this data is repeated, and which is the type of the data field. Thus, report descriptors are specific to a device.

Listing 3.4: HID Configuration Descriptor

```
uint8_t * USBHID::configurationDesc() {
    static uint8_t configurationDescriptor[] = {
        CONFIGURATION_DESCRIPTOR_LENGTH,// bLength
        CONFIGURATION_DESCRIPTOR,       // bDescriptorType
        LSB(TOTAL_DESCRIPTOR_LENGTH),   // wTotalLength (LSB)
        MSB(TOTAL_DESCRIPTOR_LENGTH),   // wTotalLength (MSB)
        0x01,                           // bNumInterfaces
        DEFAULT_CONFIGURATION,          // bConfigurationValue
        0x00,                           // iConfiguration
        C_RESERVED | C_SELF_POWERED,    // bmAttributes
        C_POWER(0),                     // bMaxPower

        //Interface descriptor
        INTERFACE_DESCRIPTOR_LENGTH,    // bLength
        INTERFACE_DESCRIPTOR,           // bDescriptorType
        0x00,                           // bInterfaceNumber
        0x00,                           // bAlternateSetting
        0x02,                           // bNumEndpoints
        HID_CLASS,                      // bInterfaceClass
        HID_SUBCLASS_NONE,              // bInterfaceSubClass
        HID_PROTOCOL_NONE,              // bInterfaceProtocol
        0x00,                           // iInterface

        //hid descriptor
        HID_DESCRIPTOR_LENGTH,          // bLength
        HID_DESCRIPTOR,                 // bDescriptorType
        LSB(HID_VERSION_1_11),          // bcdHID (LSB)
        MSB(HID_VERSION_1_11),          // bcdHID (MSB)
        0x00,                           // bCountryCode
        0x01,                           // bNumDescriptors
        REPORT_DESCRIPTOR,              // bDescriptorType
        LSB(reportDescLength()),            // wDescriptorLength (LSB)
        MSB(reportDescLength()),            // wDescriptorLength (MSB)

        //endpoint descriptor (interrupt IN endpoint)
        ENDPOINT_DESCRIPTOR_LENGTH,     // bLength
        ENDPOINT_DESCRIPTOR,            // bDescriptorType
        PHY_TO_DESC(EPINT_IN),          // bEndpointAddress
        E_INTERRUPT,                    // bmAttributes
        LSB(MAX_PACKET_SIZE_EPINT),     // wMaxPacketSize (LSB)
        MSB(MAX_PACKET_SIZE_EPINT),     // wMaxPacketSize (MSB)
        1,                              // bInterval (milliseconds)

        //endpoint descriptor (interrupt OUT endpoint)
        ENDPOINT_DESCRIPTOR_LENGTH,     // bLength
        ENDPOINT_DESCRIPTOR,            // bDescriptorType
        PHY_TO_DESC(EPINT_OUT),         // bEndpointAddress
        E_INTERRUPT,                    // bmAttributes
        LSB(MAX_PACKET_SIZE_EPINT),     // wMaxPacketSize (LSB)
        MSB(MAX_PACKET_SIZE_EPINT),     // wMaxPacketSize (MSB)
        1,                              // bInterval (milliseconds)
    };
    return configurationDescriptor;
}
```

In the configuration descriptor, we find the interface descriptor which defines the class used by this device (HID_CLASS) and the number of endpoints that will be used (2). The HID descriptor gives information on the length of the report descriptor. Finally, there are endpoint descriptors. The communication uses two interrupt endpoints: one IN to send data to the host and one OUT to receive data from the host. Interrupts endpoint are polled by the host, that is why there is a bInterval field in endpoint descriptors.

HID: Class specific requests

The HID specification defines several class specific requests. All these requests are processed in the USBCallback_request() virtual method. The main specific requests are:

- GET_REPORT: allows the host to receive a report via the Control pipe. This request is useful at initialization time for absolute items and for determining the state of feature items
- SET_REPORT: allows the host to send a report to the device, possibly setting the state of input, output, or feature controls.
- GET_DESCRIPTOR: to request the HID descriptor and the report descriptor

Generic HID device

If a USBHID is instantiated, the device created is a generic HID device. This means that it is not a specific device such as a mouse or a keyboard but a device which can send and receive raw data. The report descriptor for this device is:

Listing 3.5: Generic HID Report Descriptor

```
uint8_t * USBHID::reportDesc() {
    static uint8_t reportDescriptor[] = {
        0x06, LSB(0xFFAB), MSB(0xFFAB),
        0x0A, LSB(0x0200), MSB(0x0200),
        0xA1, 0x01,           // Collection 0x01

        //data are sent in packets containing input_length bytes
        0x75, 0x08,           // report size = 8 bits
        0x15, 0x00,           // logical minimum = 0
        0x26, 0xFF, 0x00,     // logical maximum = 255
        0x95, input_length,   // report count
        0x09, 0x01,           // usage
        0x81, 0x02,           // Input (array)

        //data are received in packets containing output_length bytes
        0x95, output_length,// report count
        0x09, 0x02,           // usage
        0x91, 0x02,           // Output (array)

        0xC0                  // end collection

    };
    return reportDescriptor;
}
```

There are two main parts:

- Bytes are sent to the host by packets of length **input_length**
- Bytes are received by packets of length **output_length**

Additionnally, this class defines methods to send and receive reports:

Listing 3.6: Send and receive HID reports

```
bool USBHID::send(HID_REPORT *report)
{
    return USBDevice::writeEP(EPINT_IN, report->data, report->length, MAX_HID_REPORT_SIZE);
}


bool USBHID::read(HID_REPORT *report)
{
    uint16_t bytesRead = 0;
    bool result;
    result = USBDevice::readEP(EPINT_OUT, report->data, &bytesRead, MAX_HID_REPORT_SIZE);
    if(!readStart(EPINT_OUT, MAX_HID_REPORT_SIZE))
        return false;
    report->length = bytesRead;
    return result;
}
```

USBMOUSE

USBMOUSE class is a subclass of USBHID. The important descriptor of this class is the report descriptor which defines data structures which will be exchanged.

Listing 3.7: Mouse Report Descriptor

```
uint8_t * USBMouse::reportDesc() {
    static uint8_t reportDescriptor[] = {
        USAGE_PAGE(1),        0x01,        // Generic Desktop
        USAGE(1),             0x02,        // Mouse
        COLLECTION(1),        0x01,        // Application
        USAGE(1),             0x01,        // Pointer
        COLLECTION(1),        0x00,        // Physical

        // Buttons
        REPORT_COUNT(1),      0x03,        // 3 buttons
        REPORT_SIZE(1),       0x01,        // 1 bit for each button
        USAGE_PAGE(1),        0x09,        // Buttons
        USAGE_MINIMUM(1),     0x1,
        USAGE_MAXIMUM(1),     0x3,
        LOGICAL_MINIMUM(1),   0x00,        // Button not pressed
        LOGICAL_MAXIMUM(1),   0x01,        // Button pressed
        INPUT(1),             0x02,        // Input, absolute data
        REPORT_COUNT(1),      0x01,        // Padding to fill 1 byte (5 + 3)
        REPORT_SIZE(1),       0x05,
        INPUT(1),             0x01,

        // X, Y and scroll
        REPORT_COUNT(1),      0x03,        // 3 features: X, Y, scroll
        REPORT_SIZE(1),       0x08,        // 1 byte for X, 1 byte for Y and 1 byte for scroll
        USAGE_PAGE(1),        0x01,
        USAGE(1),             0x30,        // X
        USAGE(1),             0x31,        // Y
        USAGE(1),             0x38,        // scroll
        LOGICAL_MINIMUM(1),   0x81,        // Minimum: -127
        LOGICAL_MAXIMUM(1),   0x7f,        // Maximum: 127
        INPUT(1),             0x06,        // Input, Relative data

        END_COLLECTION(0),
        END_COLLECTION(0),
    };
    return reportDescriptor;
}
```

On this report descriptor, we can distinguish three different parts:

- At the beginning, there is the indication that a mouse report descriptor will be described
- Mouse buttons: There are 3 buttons represented by 1 bit. Values for these buttons are 0 or 1 (pressed or not). They are input values which means that the device will send information to the host
- X, Y and scroll: 3 different features, each represented by 1 byte. Values can vary between -127 and 127. They are also input values

The main method to send a mouse state to the host is:

Listing 3.8: Mouse Report Descriptor

```
bool USBMouse::mouseSend(int8_t x, int8_t y, uint8_t buttons, int8_t z) {
    HID_REPORT report;
    report.data[0] = buttons & 0x07;
    report.data[1] = x;
    report.data[2] = y;
    report.data[3] = -z; // >0 to scroll down, <0 to scroll up

    report.length = 4;

    return USBHID::send(&report);
}
```

Concerning USBKeyboard and USBMouseKeyboard classes, the important part is the definition of the report descriptor which defines the format of data sent to the host.

3.2.5 CDC class to emulate a virtual serial port

Introduction

The Communication Device Class (CDC) is a general-purpose way to enable all types of communications on the Universal Serial Bus. This class makes it possible to connect telecommunication devices such as digital telephones or analog modems, as well as networking devices such as ADSL or Cable modems. While a CDC device enables the implementation of quite complex devices, it can also be used as a very simple solution for communication on the USB. For example, a CDC device can appear as a virtual COM port, which greatly simplifies application programming on the host side. For mbed a subset of the CDC class has been implemented in order to create a virtual serial port over USB.

Descriptors

Concerning the device descriptor, bDeviceClass has to be 02h to specify a CDC device.

Two interfaces are used in the configuration descriptor:

- **Communication Class Interface**: used to request and manage the device state. This interface is also used by the host to request capabilities and parameters of the communication. To provide information about the communication, four descriptors are used:
  - CDCHeaderDescriptor: marks the beginning of the concatenated set of functional descriptors for the interface
  - CDCCallManagementDescriptor: describes the processing of calls for the Communication Class interface
  - CDCAbstractControlManagementDescriptor: describes the commands supported by the Communication Class interface. This device supports for instance the request GET_LINE_CODING which allows the host to know parameters of the communication (number of stop bits, parity, number of data bits)
  - CDCUnionDescriptor: describes the relationship between a group of interfaces that can be considered to form a functional unit. Here, the master interface is the Communication Class Interface and the slave one is the data Class Interface
- **Data Class Interface**: this interface is used for data transfers. For the communication, either bulk or interrupt endpoints are required. For a serial port, reliability of the transmission is very important and the data transfers are not time-critical. That is why, two bulk endpoints (one IN and one OUT) are used

CDC: Class specific requests

The CDC specification defines several class specific requests. All these requests are processed in the USBCallback_request() virtual method. The main specific requests are:

- GET_LINE_CODING: allows the host to know parameters of the communication (number of stop bits, parity, number of data bits)
- SET_LINE_CODING: allows the host to specify typical line-character formatting properties

USBSerial class

Once the device is enumerated, data can be exchanged over the two bulk endpoints. To send a character, we just write the character on the bulk IN endpoint. Characters received are pushed on a circular buffer called buf. This step is done in the EPx_OUT_callback() virtual function called when a packet is received. We can then define two methods to send and receive data:

Listing 3.9: USBSerial: putc and getc

```
int USBSerial::_putc(int c) {
        writeEP(EPBULK_IN, uint8_t *)&c, 1, MAX_CDC_REPORT_SIZE)
    return 1;
}

int USBSerial::_getc() {
    uint8_t c;
    while (buf.isEmpty());
    buf.dequeue(&c);
    return c;
}
```

```
bool USBSerial::EP2_OUT_callback() {
    uint8_t c[65];
    uint16_t size = 0;

    //we read the packet received and put it on the circular buffer
    readEP(EPBULK_OUT, c, size, MAX_CDC_REPORT_SIZE)
    for (int i = 0; i < size; i++) {
        buf.queue(c[i]);
    }

    //call a potential handler
    rx.call();

    // We reactivate the endpoint to receive next characters
    readStart(EPBULK_OUT, MAX_PACKET_SIZE_EPBULK);
    return true;
}
```

3.2.6 MSD class

Introduction

The Mass Storage Device (MSD) class is an extension to the USB specification that defines how mass storage devices, such as a hard-disk, SD card or flash chip should operate on the USB. Many devices use the MSD class to simplify data transfer. More common devices which use this class are hard-drives, USB-stick, MP3 or video player.

MSD descriptors

The MSD configuration descriptor defines one interface with bInterfaceClass = 08h specifying the mass storage class. Two bulk endpoints are then defined to transfer data. Bulk endpoints are used because large amount of data can be transferred.

MSD class specific requests

The MSD specification defines several class specific requests. All these requests are processed in the USBCallback_request() virtual method. The main specific requests are:

- RESET: used to reset the device by the host
- GET_MAX_LUN: allows the host to determine the maximum Logical Unit Number (LUN) supported by the device. This is not equivalent to the number of logical unit on the device, since units are numbered starting from 0. In the mbed library, only 1 logical unit is supported.

MSD transaction

The communication between the device and the host is divided into three steps:

- Command stage
- An optional data stage
- Status stage

The **command stage** is used by the host to transmit instructions to be performed by the device. These instructions are contained in a Command Block Wrapper (CBW) sent by the host. The CBW describes several parameters of the transaction:

When a CBW has been received and decoded by the device, an optional **data stage** may take place if the command requires it. During this step, data flow from or to the device (for instance, if the host wants to read the content of an SD card, there will be a data IN stage). Several transfers can occur successively.

Once the data stage performed, the device must send a Command Status Wrapper (CSW) in response to the CBW. A CSW is used to return the result of a command.

MSD State machine

To implement the three previous step of a transaction, a state machine is used. This state machine contains 5 states:

Offset	Field	Length (bytes)	Meaning
0	dCBWSignature	4	Signature to identify a valid CBW (0x43425355)
4	dCBWTag	4	id of the CBW. The device have to answer in the status stage by sending a Command Status Wrapper (CSW) containing the same id.
8	dCBWTransferLength	4	Length of transfer
12	bmCBWFlags	1	bit7: transfer direction (0: OUT, 1: IN)
13	bCBWLUN	1	**Bits 0-3**: logical unit to which the command is sent
14	bCBWCBLength	1	**Bits 0-5**: Length of command block in bytes
15	CBWCB	0-16	Command block: instructions to be executed by the device

Table 3.3: Command Block Wrapper (CBW)

Offset	Field	Length (bytes)	Meaning
0	dCSWSignature	4	Signature to identify a valid CSW (0x53425355)
4	dCSWTag	4	Same id as in the CBW
8	dCSWDataResidue	4	Difference between expected and real transfer length
12	bCSWStatus	1	Indicates the result (success or failure) of the command.

Table 3.4: Command Status Wrapper (CSW)

- WAIT_CBW: waiting for the reception of a valid CBW. If a non valid CBW is received, send a CSW reporting an error.
- PROCESS_CBW: decode and execute command from the CBW. If there is a processing error, go into the ERROR state
- SEND_CSW: a CSW is sent
- WAIT_CSW: waiting for CSW reception by the host
- ERROR: to report an error.

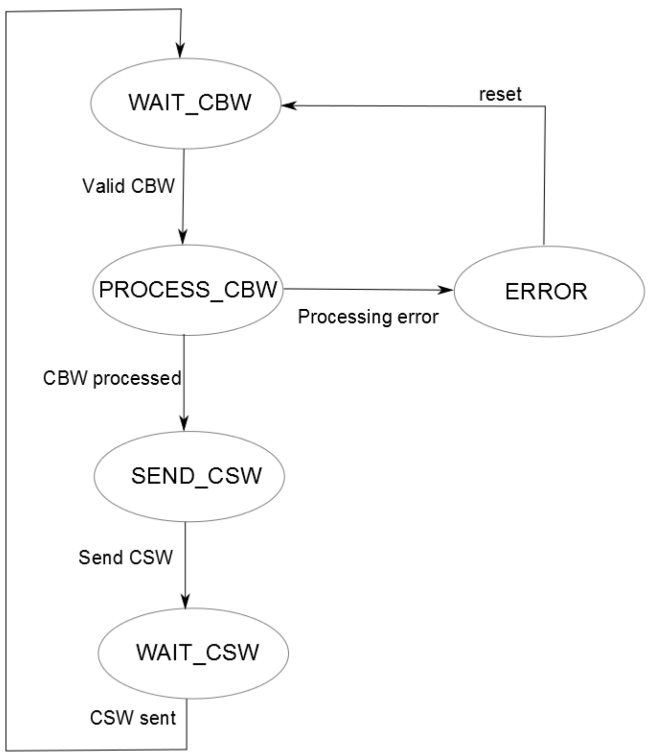


Figure 3.9: MSD state machine

SCSI commands

All commands sent by the host contained in the CBW are part of the SCSI command architecture. In the SCSI protocol, the initiator sends a SCSI command to the target which then responds. SCSI commands are sent in a block, which consists of a one byte operation code followed by five or more bytes containing command-specific parameters. Upon receiving and processing the CBW the target will return a status code byte.

Main commands that have to be processed by the device are:

- SCSI_INQUIRY: The host usually issues an Inquiry command right after the enumeration phase to get more information about the device
- SCSI_READ_CAPACITY: The Read Capacity command enables the host to retrieve the number of blocks present on a media, as well as their size
- SCSI_READ (6)/(10): This command is used by the host to read data from the device.
- SCSI_REQUEST_SENSE: If there is an error during the execution of a command, the host will issue a Request Sense to get more information about the problem
- SCSI_TEST_UNIT_READY: This command provides a way to check if a logical unit is ready. If the logical unit is not ready, the device reports an error in the CSW. Then the host sends a SCSI_REQUEST_SENSE to have more information about the error

The main method to decode a CBW and a SCSI command is:

Listing 3.10: MSD CBW decoding

```
void USBMSD::CBWDecode(uint8_t * buf, uint16_t size) {
    if (size == sizeof(cbw)) {
        memcpy((uint8_t *)&cbw, buf, size);
        if (cbw.Signature == CBW_Signature) {
            csw.Tag = cbw.Tag;
            csw.DataResidue = cbw.DataLength;
            if ((cbw.CBLength <  1) || (cbw.CBLength > 16) ) {
                fail();
            } else {
                switch (cbw.CB[0]) {
                    case TEST_UNIT_READY:
                        testUnitReady();
                        break;
                    case REQUEST_SENSE:
                        requestSense();
                        break;
                    case INQUIRY:
                        inquiryRequest();
                        break;
                    case MODE_SENSE6:
                        modeSense6();
                        break;
                    case READ_FORMAT_CAPACITIES:
                        readFormatCapacity();
                        break;
                    case READ_CAPACITY:
                        readCapacity();
                        break;
                    case READ10:
                    case READ12:
                        if (infoTransfer()) {
                            if ((cbw.Flags & 0x80)) {
                                stage = PROCESS_CBW;
                                memoryRead();
                            } else {
                                stallEndpoint(EPBULK_OUT);
                                csw.Status = CSW_ERROR;
                                sendCSW();
                            }
                        }
                        break;
                    case WRITE10:
                    case WRITE12:
                        if (infoTransfer()) {
                            if (!(cbw.Flags & 0x80)) {
                                stage = PROCESS_CBW;
                            } else {
                                stallEndpoint(EPBULK_IN);
                                csw.Status = CSW_ERROR;
```

```
                                sendCSW();
                            }
                        }
                        break;
                    case VERIFY10:
                        if (!(cbw.CB[1] & 0x02)) {
                            csw.Status = CSW_PASSED;
                            sendCSW();
                            break;
                        }
                        if (infoTransfer()) {
                            if (!(cbw.Flags & 0x80)) {
                                stage = PROCESS_CBW;
                                memOK = true;
                            } else {
                                stallEndpoint(EPBULK_IN);
                                csw.Status = CSW_ERROR;
                                sendCSW();
                            }
                        }
                        break;
                    default:
                        fail();
                        break;
                }
            }
        }
    }
}
```

Use of USBMSD with a specific storage chip

The USBMSD class implements the MSD protocol as defined previously. But this class doesn't work standalone, a subclass is needed to define pure virtual functions called in USBMSD in order to access a specific storage chip.

Six pure virtual functions have to be defined:

- virtual **int disk_read(char * data, int block)**: function to read a block
- virtual **int disk_write(const char * data, int block)**: function to write a block
- virtual **int disk_initialize()**: function to initialize the memory
- virtual **int disk_sectors()**: return the number of blocks
- virtual **int disk_size()**: return the memory size
- virtual **int disk_status()**: return the status of the storage chip (0: OK, 1: not initialized, 2: no medium in the drive, 4: write protection)

3.2.7 Audio class

Introduction

The audio class enables a USB device to send and receive, for example, encoded speech or music. In the mbed library, a USB audio device has been developed in order to send and receive audio packets in the same device. An application of the Audio class could be to play music on a speaker or an I2S/I2C chip connected to the mbed, or to send audio packets from a microphone. Devices in the audio class use isochronous transfers for audio streams. Audio devices can either be synchronous or asynchronous:

- Asynchronous isochronous audio endpoints produce or consume data at a rate that is locked either to a clock external to the USB or to a free-running internal clock. These endpoints cannot be synchronized to a Start Of Frame (SOF)
- Synchronous isochronous endpoints are handled in the Start Of Frame event generated each millisecond

In the mbed library, only the synchronous mode has been developed:

- on each Start Of Frame event:
  - if a user has requested the reading of an audio packet: try to read a packet on the OUT isochronous endpoint
  - if a user wants to send an audio packet: write the audio packet on the IN isochronous endpoint

Descriptors

Three interfaces are defined in the configuration descriptor:

- Audio Control Interface: describes the functional behavior of the device including input terminal, output terminal or features unit:
  - An Input Terminal is used to interface between outside the device and other Units inside the device
  - An Output Terminal is used to interface between Units inside the audio function and outside the device
  - A feature Unit provides audio control concerning volume or mute for instance
- Audio Streaming Interface: used to interchange digital audio data streams between the Host and the audio device. In our case, two audio streaming interfaces are defined:
  - to receive an audio stream over an isochronous OUT endpoint
  - to send a stream over an isochronous IN endpoint

  This interface specifies the type of data exchanged, the number of channels used, the number of bits used to encode a sample or the frequency. In the mbed library, a PCM representation on 16 bits is used to send and receive audio packets. Other parameters can be chosen by the user.

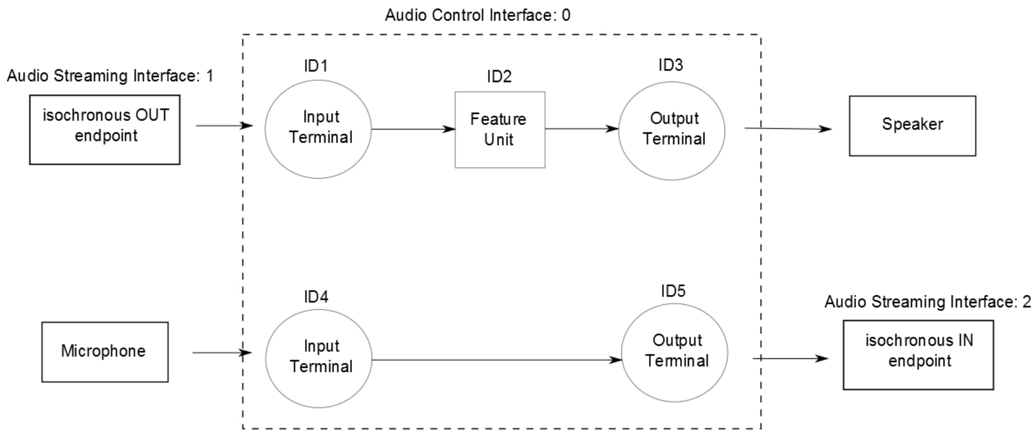


Figure 3.10: Audio device interfaces

Audio class specific requests

The Audio specification defines several class specific requests. All these requests are processed in the USBCallback_request() virtual method. Specific requests have been implemented:

- GET_CUR: allows the host to know the current volume or the mute state
- SET_CUR: allows the host to set the current volume or the mute state
- GET_MIN: allows the host to know the minimum volume level
- GET_MAX: allows the host to know the maximum volume level
- GET_RES: allows the host to know the resolution attribute between two volume levels

Audio packet management

As explained before, one audio packet is sent or received on a Start Of Frame event (each millisecond). So let's say that a frequency of 48 kHz has been chosen with 2 channels (stereo). Knowing that each sample of a packet is 16 bits long, 48 * 2 bytes will be received or sent each millisecond for one channel. In total, for 2 channels, 48 * 2 * 2 bytes will be received. The audio length packet received or sent each millisecond can be calculated as:

AUDIO_LENGTH_PACKET = (FREQ / 500) * nb_channel

Audio packets have to be interpreted differently, according to the number of channels:

The main part for sending and receiving audio packets is in the virtual function called in USBHAL on each Start Of Frame event:

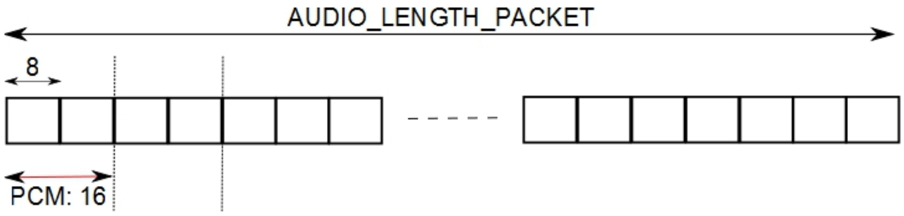


Figure 3.11: PCM packet: mono

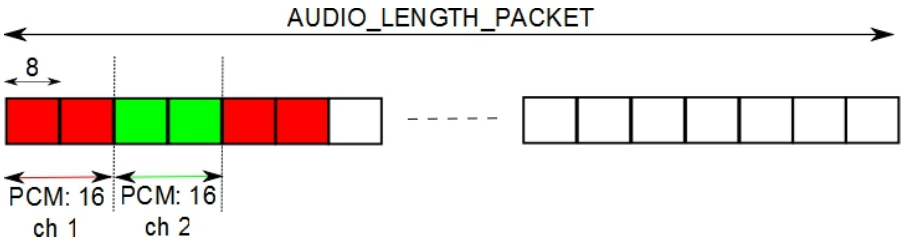


Figure 3.12: PCM packet: stereo

Listing 3.11: Audio packets management in SOF events

```
// Called in ISR context on each Start Of Frame
void USBAudio::SOF(int frameNumber) {
    uint16_t size = 0;

    //read the isochronous endpoint if the user has provided a buffer
    if (buf_stream_in != NULL) {
        if (USBDevice::readEP_NB(EP3OUT, (uint8_t *)buf_stream_in, &size, PACKET_SIZE_ISO_IN)↩
            ) {
            // if an audio packet is available, notify
            if (size) {
                available = true;
                readStart(EP3OUT, PACKET_SIZE_ISO_IN);
            }
        }
    }

    //write if needed
    if (buf_stream_out != NULL) {
        USBDevice::writeNB(EP3IN, (uint8_t *)buf_stream_out, PACKET_SIZE_ISO_OUT, ↩
            PACKET_SIZE_ISO_OUT);
    }

    SOF_handler = true;
}
```

3.3 Conclusion

Mbed users can now use their mbed to emulate a USB device. I really wanted explore all the capabilities of USB. This is why I developed several classes:

- Human Interface Device (HID), in my opinion the best way to develop a custom USB device as it permits the exchange of raw data without the need for a specific driver on the host side.
- Communication Device Class (CDC) to provide a virtual serial port
- Mass Storage Device (MSD) to access a block storage chip from a computer
- Audio to receive and send audio packets
- MIDI to send and receive MIDI messages

Conclusion

By way of conclusion, I again want to thank my manager Simon Ford. He permitted me to discover new domains involved in my projects:

- the Internet of Things project where I had to connect sensors to the Internet over a websocket communication.
- the Remote Procedure Call Mechanism over websocket which permitted to call methods on a distant mbed
- the USB device stack to emulate USB devices with an mbed

The Internet of Things and RPC projects were very interesting because they needed several skills such as:

- software skills in embedded systems to program the mbed
- software skills in web programming
- hardware skills

Thanks to these projects, I learned a great deal about web programming, in particular Javascript. I am now more familiar with web technologies such as HTML5 and websockets. It was very good experience, I found, to develop a project which required several skills. I also really appreciated working on the USB. It was a new domain for me, and it was very interesting to discover all about the USB protocol. It was satisfying and motivating to observe technical results day after day on the USB device stack, knowing that it was a totally new subject for me. In the end, I was happy to be able to provide to mbed users a USB device stack able to emulate many USB classes.

This internship has been a great experience. It was a real pleasure to work with the mbed team in a very good atmosphere. Furthermore, carrying out this internship in a foreign country allowed me to be immersed in a different culture and face the problems asscoiated with living and working entirely with a non-native language (English). I met several interesting people both in and out of the lab, such as engineers, students, and also people not connected with computer science.

References

http://www.bbc.co.uk/news/technology-13613536
TODO