Industrial Engineering Internship Report

Embedded systems, Internet of Things and Communication

Author:
Samuel Mokrani

Supervisor:
Simon Ford

December 31, 2011

Contents

Chapter 1

ARM and Mbed

1.1 ARM

ARM Holdings plc is a British multinational semiconductor and software company. The headquarter is based in Cambridge. This company 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-a-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 developped by Acorn Computer involved in an Apple project. ARM got bigger little by little 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 the software development tools for microcontrollers. More recently, ARM joined with Texas Instrument, 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 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 of this success is their low electric power consumption; making them suitable for embedded systems. Even if ARM products are widely used, the company doesn't manufacture its own CPUs. The company licenses its technology as Intellectual Property (IP). Companies like Intel, Texas Intrument or Nvidia are making processors based on ARM's IP.

1.2 Mbed

In wish to pursue new areas of expertise, ARM founded Mbed in 2009. Mbeds core area is development of prototyping boards (called mbed) based on ARM processors (Cortex M0 and Cortex M3). The purpose is to make available a simple and easy to set-up prototyping solution using 32-bits processors. Mbeds boards have been designed for quick experimentation. Users can try something and see if it is doable in a very easily and quickly manner.

1.2.1 Mbed boards: LPC1768 and LPC11U24

For the moment, two boards have been developped. The first one is based on the LPC1768 microcontroller from NXP which uses a Cortex m3. More recently, a new board based on a LPC11U24 (Cortex m0) was born. This last mbed has been particularly designed to prototype USB devices or battery powered applications.

1.2.2 The online compiler

The two main keywords of Mbed are "simple" and "cloud computing". Users indeed use an online compiler to develop their program in C++. They compile online and transfer the binary received into the mbed, which is connected to a computer over a USB cable and detected as a mass storage device. They just have to press the

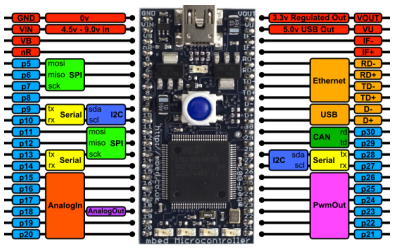


Figure 1.1: mbed board: lpc1768 (cortex m3)

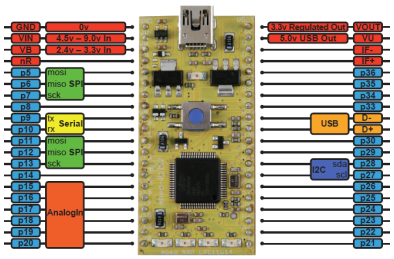


Figure 1.2: mbed board: lpc11U24 (cortex m0)

reset button to see their program running.

As explained before, users have a complete online IDE totally independant of the underlying operating system. This IDE has a lot of very 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 instanciate an object such as SPI, I2C,... to have access to a great API which abstract all the low-level layer.

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

To finish this tour of the mbed environment, users can find an active website. First, users have access to a forum. They have also access to a handbook where there is a lot of documentation and examples concerning mbed libraries, the hardware part of the mbed,... But the most important feature of this website is for me the central cookbook. This cookbook is a great source of example coming from other mbed users. You can almost find for instance a lot of libraries to use popular sensors like accelerometers, pressure sensors,... Each user is free to contribute to this part of the website by doing articles explaining their project.

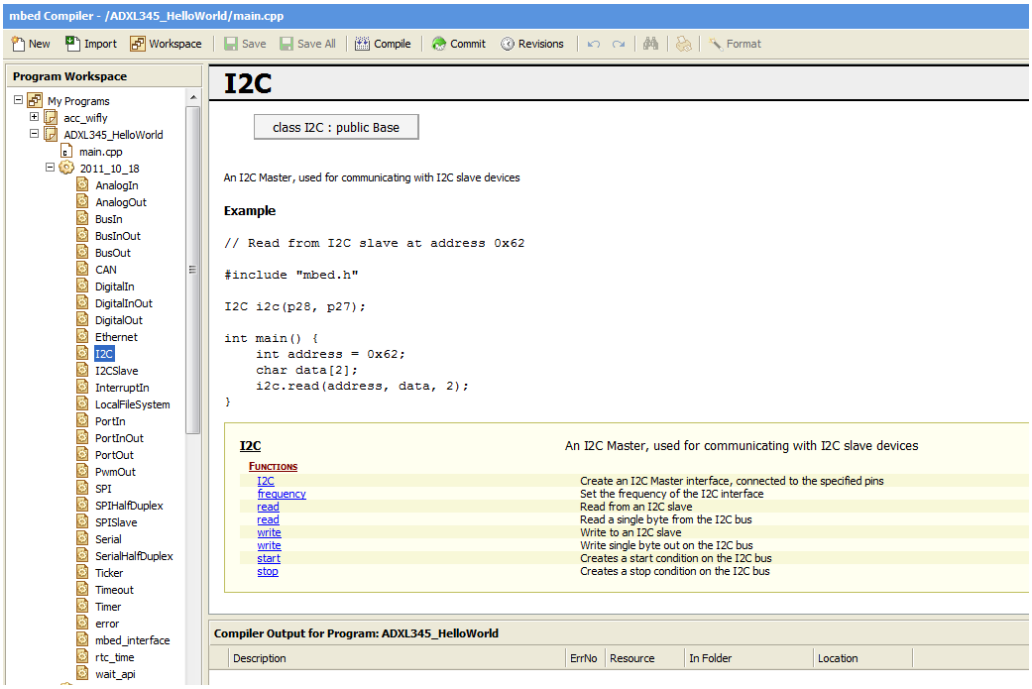


Figure 1.3: mbed libraries

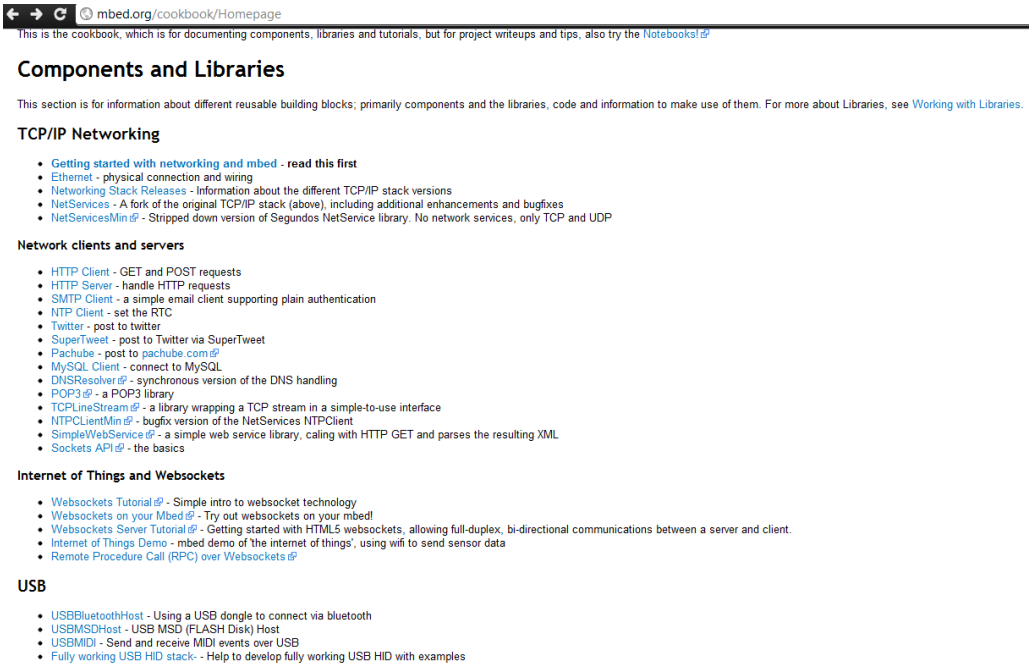


Figure 1.4: mbed cookbook

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, ect. Cisco predicts in an interview from 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. And Cisco is not the only one company to predict a such boom. VMware's CEO Paul Maritz said during a speech at the annual VMworld conference in Las Vegas in August 2011 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, working on the connection of different sensors to the Internet is becoming more and more crucial. I will describe in this chapter two projects concerning the Internet of Things: real-time data streaming from sensors and Remote Procedure Call mechanism. Both of these projects use a new feature of HTML5: a Webscocket communication.

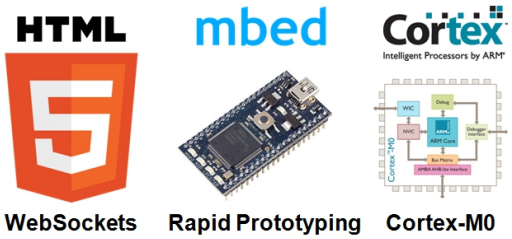


Figure 2.1: HTML5 on embedded systems

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.

This figure shows the reduction in latency. Once the connection is established, messages can flow from the server to the browser. As the connection remains open, there is no need to send another request to the server.

2.1.2 Websocket protocol

To establish a WebSocket connection, the client and server upgrade from the HTTP protocol to the WebSocket protocol during their initial handshake.

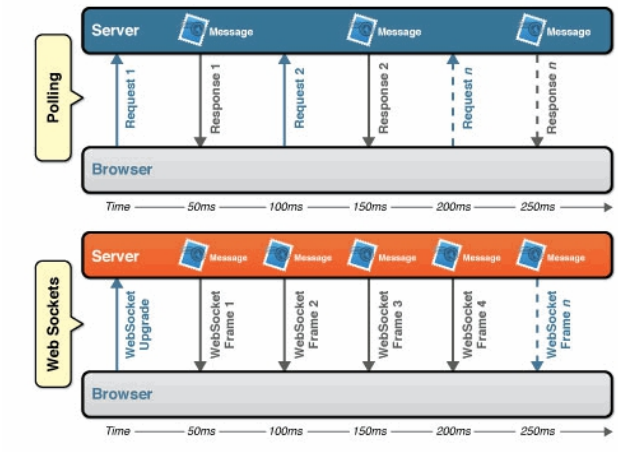


Figure 2.2: Reduction of polling and overhead by Websocket *(*Image coutesy of Kaazing)

The handshake from the client looks 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 looks 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.

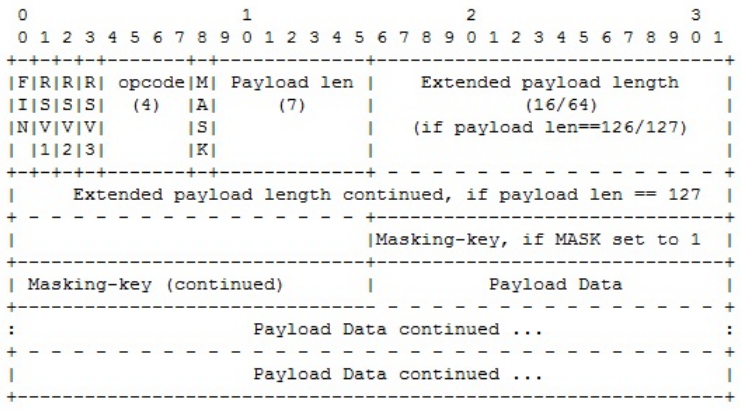


Figure 2.3: Websocket data framing

- 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

2.1.3 Architecture of a Websocket communication

A websocket communication involves several clients connected to the same websocket server. All messages from browsers are sent to the server. The server manages all messages. It can decide to send a message received from a client to another client, to broadcast all messages received to all clients connected,...

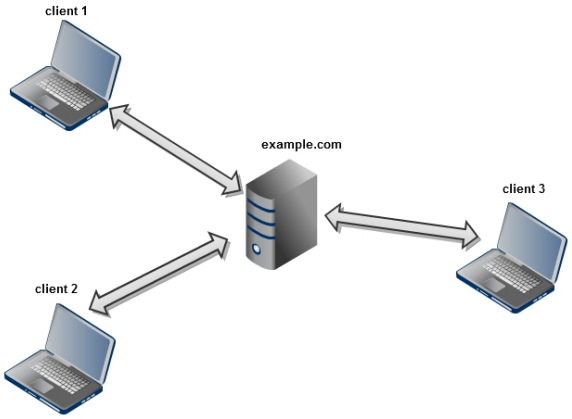


Figure 2.4: Example of websocket communication

For instance, 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 this few lines of javascript:

Listing 2.1: Device Descriptor

```
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.");
};
```

2.1.4 The mbed websocket library

A websocket client library has been developped in order to exchange data between an mbed and a server over a websocket communication. This library can either be used over an ethernet connection or a wifi connection using a roving networks wifly module. This 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 wifly 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 Real-Time Data Streaming from sensors

Mbed wanted to demonstrate that it was possible to access sensor data all over the world using an mbed board. For this, two prototypes 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 from everywhere as quick as possible. That's why we want to involve smartphones connected to the Internet over 3G for instance. To achieve this, we used QR codes to speed up the whole process. The steps to access data for a user are:

- Place the object (mbed and sensors) where desired
- Scan the QR code on the sensor with a smartphone to see its data displayed
- Click the 'add' button to add the sensor to your dashboard (central webpage showing all sensors connected)

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 (running on laptops or smartphones)

All mbed boards send messages in JSON format to the websocket server containing sensors data. The websocket server will broadcast to all client connected all messages received. So all browsers connected to this server are able to receive data from sensors.

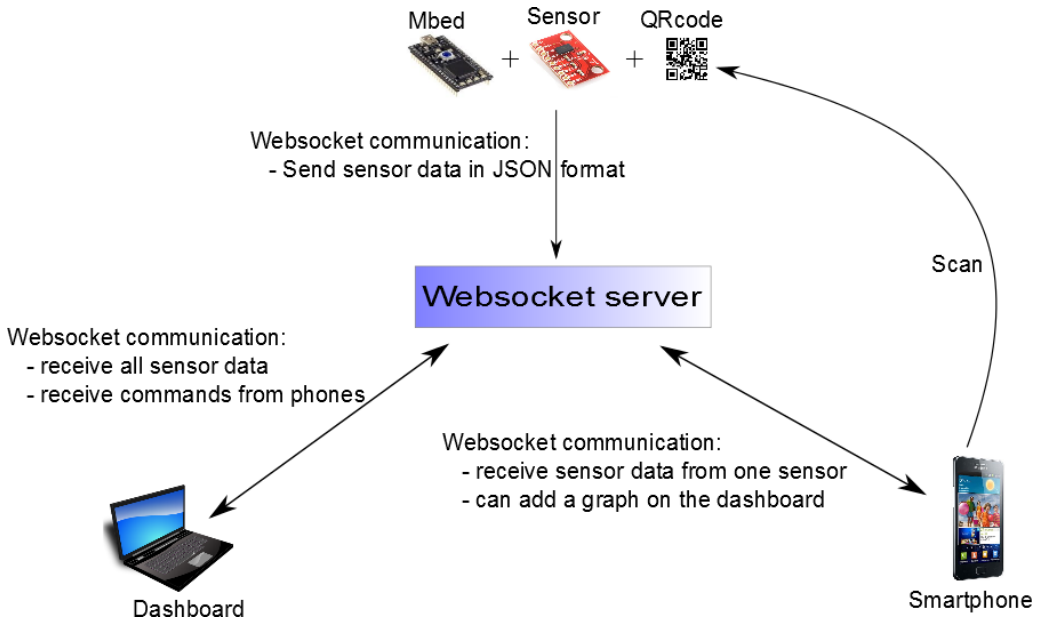


Figure 2.5: Architecture real-time data streaming

2.2.2 Websocket server

The websocket server which has been developed uses Tornado framework. This framework is written in Python. It's a non blocking web server. It uses epoll which permit to handle 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: Open a websocket connection

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

if __name__ == "__main__":
    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(443)
    tornado.ioloop.IOLoop.instance().start()
```

These few lines launch a http server associated with a handler (WSHandler) when an event occurs. All requests having a path matching the regular expression **/ws/*/*** will be redirected to 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 a 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. You can override three different events in WShandler:

- 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 is especially designed to prototype. The main parts are:

- one mbed
- a wifi module
- sensors

To connect the mbed to the internet and access the websocket server, a wifi module has been chosen. 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 wifly 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 has been developed by the mbed community to use these two previous sensors. The light sensor is connected over a GPIO as the microphone.

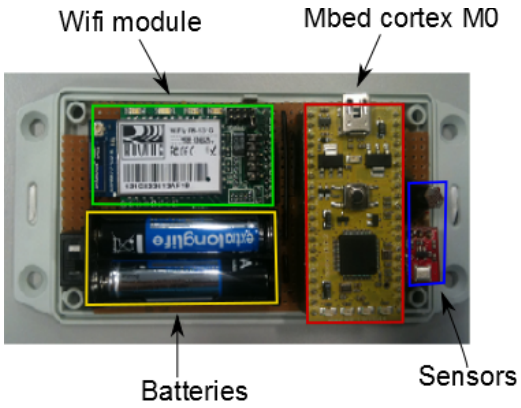


Figure 2.6: Environmental board

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

To develop webpages running on a laptop in the case of the dashboard or running on a smartphone, I wanted to use another new HTML5 feature which is 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:

- smartphones webpage: displays only the graph of the corresponding QR code scanned. Two buttons can be used to add or remove a specific graph of the dashboard webpage.
- dashboard: displays all graphs added by users.

Realtime graphs

I use Smoothie Charts which is a javascript library to stream data on 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 live chart refreshed everytime that we receive a new data.

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 will receive a such message, it will refresh a realtime graph showing the sensor data evolution.
Messages sent by smartphones control graphs on the dashboard webpage. For instance, if a user scans the QR code of the accelerometer board and press the add button, the message

{"id"="acc_add"}

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

The architecture of all webpages is quite simple:

- Connection to the websocket server
- When a message is received from sensors:
  - check the validity of the message
  - update the corresponding graph (there is one graph per sensor board)
- When a message is received from smartphones:
  - check the validity of the message
  - update the Document Object Model (DOM) using the jQuery javascript library

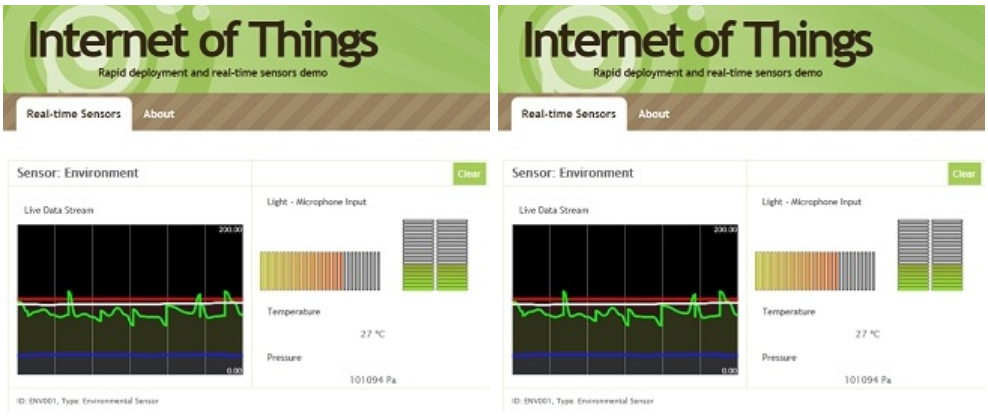
Listing 2.6: Streaming data code example

```
    // 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.

2.2.5 Conclusion

This first project shows the powerful of the new HTML5 feature: websockets. I am able to stream sensor data and display them in realtime graphs. Therefore, these data are accessible all over the world.

After the Internet of Things project, mbed wanted to develop a Remote Procedure Call mechanism between several mbeds over a websocket communication.

2.3 Remote Procedure Call over Websockets

After the Internet of Things project which shows the power of HTML5 in embedded systems, mbed wanted to use websockets to design a RPC (Remote Procedure Call) mechanism. The idea is to enable users to call a specific method on a specific mbed. Both of these mbed are connected to the Internet.

2.3.1 Architecture: mbed boards, websocket server

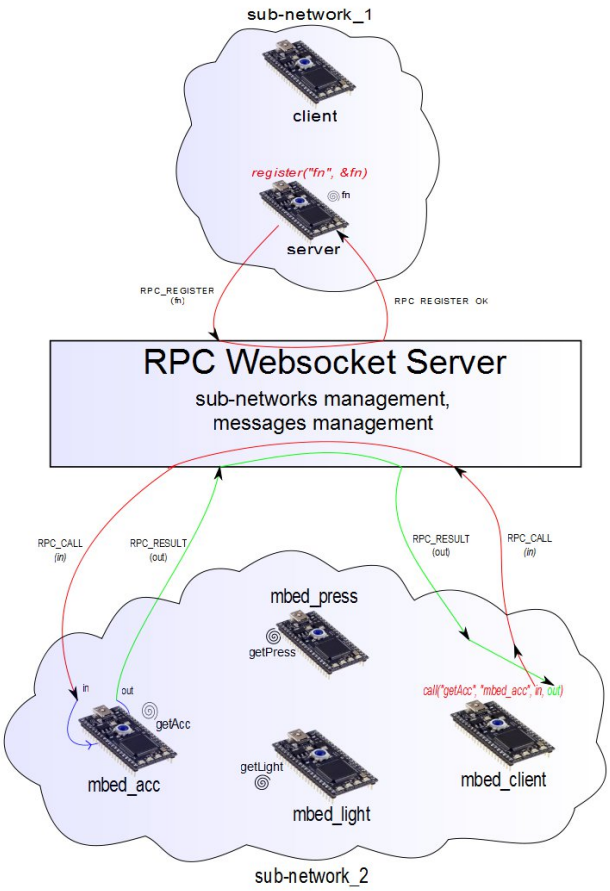


Figure 2.8: RPC architecture

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

- Two different sub-networks. On each sub-networks, all Mbeds can share methods or execute a non-local one.
- A RPC Websocket server which is responsible to manage all sub-networks and all messages exchanged. This server is written in Python and uses Tornado.

As for the Internet of Things 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:

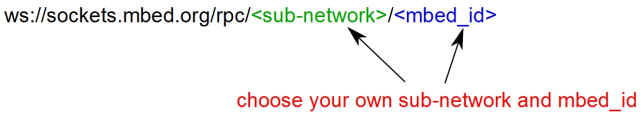


Figure 2.9: RPC: mbed identification on a subnetwork

Different steps are required to execute a distant method:

- connect several mbeds to the same subnetwork over a websocket communication
- then 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

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 argument 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 to 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 developed 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: MbedJSONValue manipulation

```
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: gateway
  - 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: MbedJSONValue manipulation

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

This method, in order to call a distant method, realizes 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
- Fill the "out" reference with the result

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: MbedJSONValue manipulation

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

This method does the following steps:

- Try 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 an ethernet jack is required.

TODO: photo

Software: mbed which registers a method

I will present the main program running on an mbed which has an accelerometer connected. The idea is to remotely access these accelerometers values from another mbed.

Listing 2.11: Register 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) {
        printf("getAcc is registered\r\n");
            //wait for incoming CALL requests
            rpc.work();
    }
    else {
        printType(t);
    }

}
```

Software: mbed which calls a distant method

Listing 2.12: Register a 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>());
    } else
        printType(t);
}
```

TODO: screenshot of result

2.3.6 Websocket server

The server uses the same framework used for the realtime streaming data project: Tornado. The server is responsible to manage 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

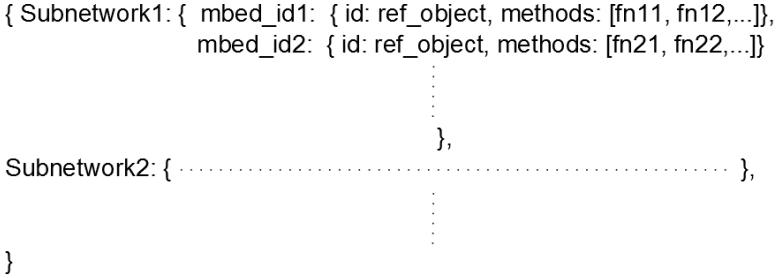


Figure 2.10: Data structure for the RPC websocket server

The websocket server follows the following logic:

- 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

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 if 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 are supporting 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 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 of 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 need 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 does not exceed 7. The following diagram represents a possible connection architecture on a USB bus.

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

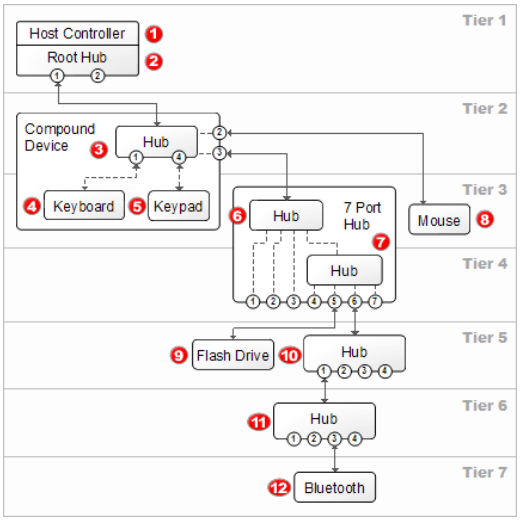


Figure 3.1: USB physical topology [1]

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 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 endpoints has a specific direction: either OUT to receive data from the host or IN to send data to the host.

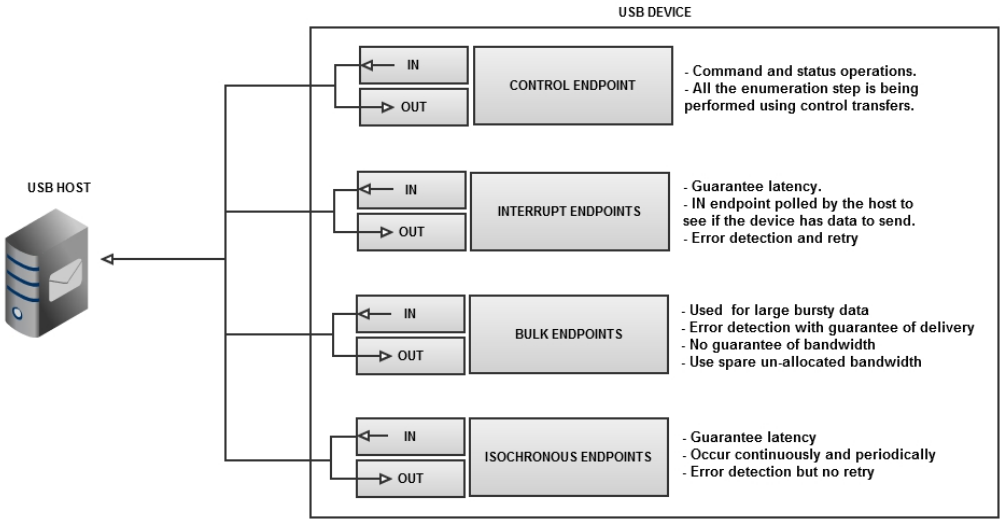


Figure 3.2: Different types of endpoints

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

- **Control endpoint**: All USB devices once connected performed 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 use 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,

streaming audio and video devices use isochronous transfers. Such devices need a guaranteed delivery rate for data but if an error occurs, the data is not re-transmitted.

3.1.4 Packets exchanged

Common fields in a USB packet

Field	Length	Meaning
SYNC	low and full speed: 8bits high speed: 32bits	A packet starts with a SYNC pattern to allow the receiver bit clock to synchronise with the data.
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 anytime, 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 some 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.

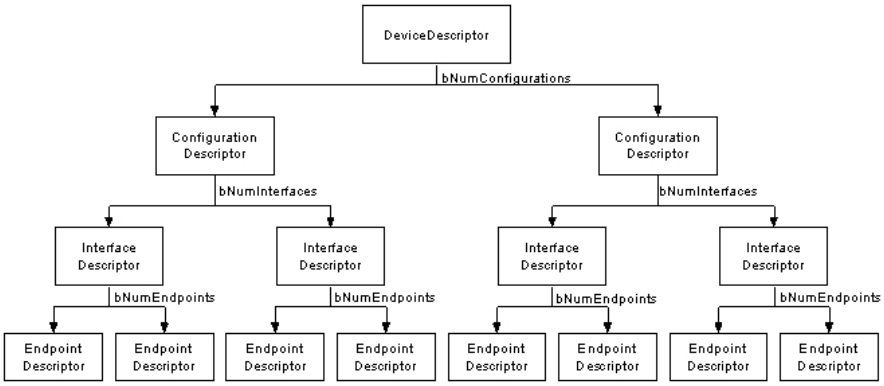


Figure 3.3: Descriptors

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.

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

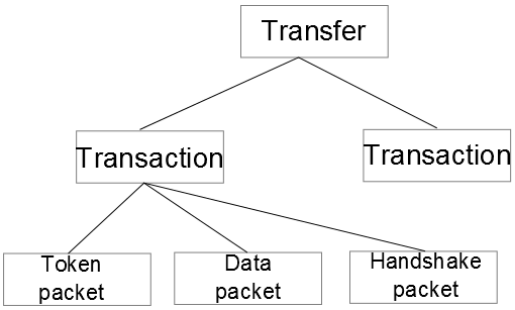


Figure 3.4: USB 2.0 transaction

- 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 developped 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 devices functionality. This class code is parsed by the USB host stack to load an appropriate device driver. Several classes has 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

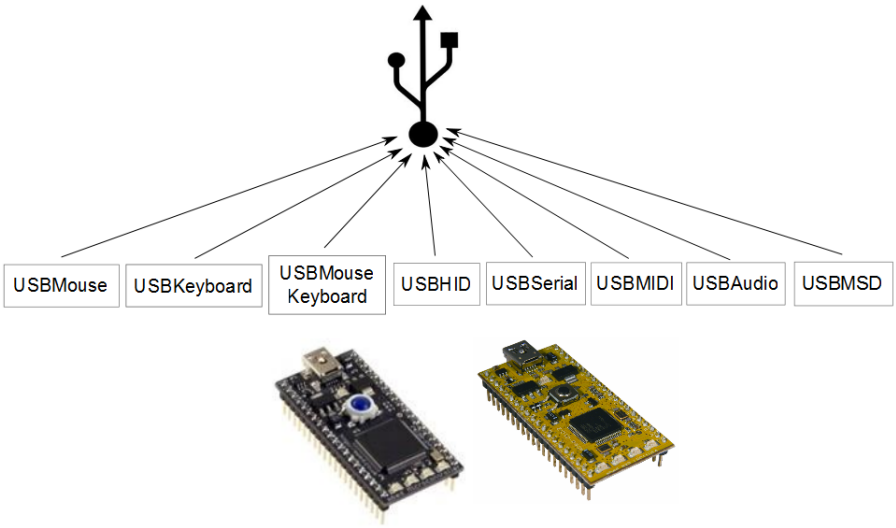


Figure 3.5: USB device stack capabilities

3.2.1 USB Device stack architecture

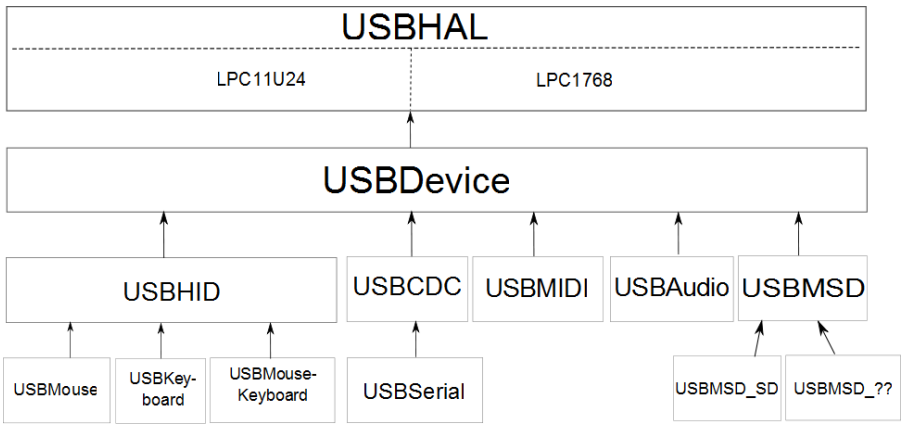


Figure 3.6: USB device stack architecture

- 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 to abstract the hardware. At this level, no differences are made between the LPC11U24 and the LPC1768 concerning the target. USBDevice is in charge to perform 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 instanciated, the mbed is enumerated as a generic HID device so that raw data can be sent and receive 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 message 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

Source code of main methods are in Annexe A: USBHAL class

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 2kbytes 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:

- 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::endpointWriteResult.

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 1ms)
- 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

3.2.3 USBDevice: target abstraction and setup packet treatment

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 treatment

Two data structure has been defined in order to perform the control endpoint treatment. 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 3.1: Data structures for control endpoint treatment

```
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;
```

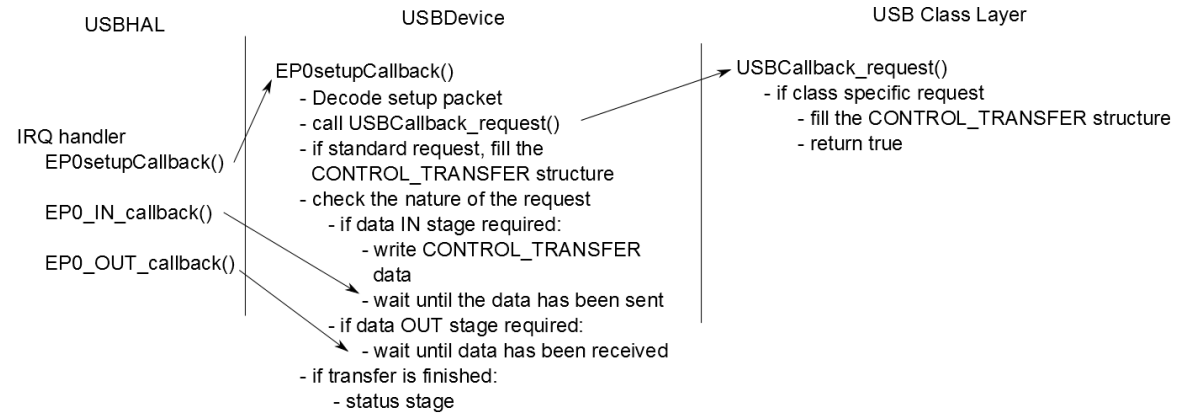


Figure 3.8: Setup packets treatment

For instance, a GET_DEVICE_DESCRIPTOR request is processed following this scheme:

- 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 on 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

3.2.4 HID 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 class devices include keyboards, mice, trackballs, and joysticks. An HID device 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 as 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 host and device resides in 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. Examples of report descriptor are provided in next sections of this document concerning HID devices such as mice or keyboards.

Concerning standard descriptors, they are quite similar for all HID devices:

Listing 3.2: 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;
}
```

Listing 3.3: 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(this->reportDescLength()),  // wDescriptorLength (LSB)
        MSB(this->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)
        10,                             // 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)
        10,                             // bInterval (milliseconds)
    };
    return configurationDescriptor;
}
```

In the device descriptor, the host will find information such as the maximum packets length that can be exchanged over the control endpoint, the device identification (vendor_id, product_id and product_release) and the USB version used.

In the configuration descriptor, we find particularly 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. There are finally 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.

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

Listing 3.4: Send and receive HID reports

```
bool USBHID::send(HID_REPORT *report)
```

```
{
    return USBDevice::write(EPINT_IN, report->data, report->length, MAX_HID_REPORT_SIZE);
}


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


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

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 kind of 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*

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 3.6: 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.
- Buttons mouse: 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. There are also input values.

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

Listing 3.7: 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);
}
```

3.2.5 CDC class

Introduction

The Communication Device Class (CDC) is a general-purpose way to enable all types of communications on the Universal Serial Bus (USB). This class makes it possible to connect telecommunication devices such as digital telephones or analog modems, as well as networking devices like ADSL or Cable modems. While a CDC device enables the implementation of quite complex devices, it can also be used as a very simple method 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

On this class, two interfaces are used:

- **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. This interface requires the control endpoint for device management. Optionnally, an interrupt IN endpoint can be used yo send event notifications to the host3.
- **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.

Listing 3.8: USBSerial device descriptor

```
uint8_t * USBCDC::deviceDesc() {
    static uint8_t deviceDescriptor[] = {
        18,                     // bLength
        1,                      // bDescriptorType
        0x10, 0x01,             // bcdUSB
        2,                      // bDeviceClass (CDC)
        0,                      // bDeviceSubClass
        0,                      // bDeviceProtocol
        MAX_PACKET_SIZE_EP0,    // bMaxPacketSize0
        LSB(VENDOR_ID), MSB(VENDOR_ID),  // idVendor
        LSB(PRODUCT_ID), MSB(PRODUCT_ID),// idProduct
        0x00, 0x01,             // bcdDevice
        1,                      // iManufacturer
        2,                      // iProduct
        3,                      // iSerialNumber
        1                       // bNumConfigurations
    };
    return deviceDescriptor;
}
```

This device descriptor is very common. The only change is the bDeviceClass field which contains the CDC class code.

Listing 3.9: USBSerial configuration descriptor

```
uint8_t * USBCDC::configurationDesc() {
    static uint8_t configDescriptor[] = {
        9,                      // bLength;
        2,                      // bDescriptorType;
        LSB(CONFIG1_DESC_SIZE), // wTotalLength
        MSB(CONFIG1_DESC_SIZE),
        2,                      // bNumInterfaces
        1,                      // bConfigurationValue
        0,                      // iConfiguration
        0x80,                   // bmAttributes
        50,                     // bMaxPower


        // interface descriptor: Communication Class Interface
        9,                      // bLength
        4,                      // bDescriptorType
        0,                      // bInterfaceNumber
        0,                      // bAlternateSetting
        1,                      // bNumEndpoints
        0x02,                   // bInterfaceClass
        0x02,                   // bInterfaceSubClass
        0x01,                   // bInterfaceProtocol
        0,                      // iInterface

        // CDC Header Functional Descriptor, CDC Spec 5.2.3.1, Table 26
        5,                      // bFunctionLength
        0x24,                   // bDescriptorType
        0x00,                   // bDescriptorSubtype
        0x10, 0x01,             // bcdCDC

        // Call Management Functional Descriptor, CDC Spec 5.2.3.2, Table 27
        5,                      // bFunctionLength
        0x24,                   // bDescriptorType
        0x01,                   // bDescriptorSubtype
        0x03,                   // bmCapabilities
        1,                      // bDataInterface

        // Abstract Control Management Functional Descriptor, CDC Spec 5.2.3.3, Table 28
        4,                      // bFunctionLength
        0x24,                   // bDescriptorType
        0x02,                   // bDescriptorSubtype
        0x06,                   // bmCapabilities

        // Union Functional Descriptor, CDC Spec 5.2.3.8, Table 33
        5,                      // bFunctionLength
        0x24,                   // bDescriptorType
        0x06,                   // bDescriptorSubtype
        0,                      // bMasterInterface
        1,                      // bSlaveInterface0

        // endpoint descriptor
        ENDPOINT_DESCRIPTOR_LENGTH,     // bLength
        ENDPOINT_DESCRIPTOR,            // bDescriptorType
        PHY_TO_DESC(EPINT_IN),          // bEndpointAddress
        E_INTERRUPT,                    // bmAttributes (0x03=intr)
        LSB(MAX_PACKET_SIZE_EPINT),     // wMaxPacketSize (LSB)
        MSB(MAX_PACKET_SIZE_EPINT),     // wMaxPacketSize (MSB)
        16,                             // bInterval



        // interface descriptor: Data Class Interface
```

```
        9,          // bLength
        4,          // bDescriptorType
        1,          // bInterfaceNumber
        0,          // bAlternateSetting
        2,          // bNumEndpoints
        0x0A,       // bInterfaceClass
        0x00,       // bInterfaceSubClass
        0x00,       // bInterfaceProtocol
        0,          // iInterface

        // endpoint descriptor
        7,                      // bLength
        5,                      // bDescriptorType
        PHY_TO_DESC(EPBULK_IN), // bEndpointAddress
        0x02,                   // bmAttributes (0x02=bulk)
        LSB(MAX_PACKET_SIZE_EPBULK),    // wMaxPacketSize (LSB)
        MSB(MAX_PACKET_SIZE_EPBULK),    // wMaxPacketSize (MSB)
        0,                      // bInterval

        // endpoint descriptor
        7,                      // bLength
        5,                      // bDescriptorType
        PHY_TO_DESC(EPBULK_OUT),// bEndpointAddress
        0x02,                   // bmAttributes (0x02=bulk)
        LSB(MAX_PACKET_SIZE_EPBULK),    // wMaxPacketSize (LSB)
        MSB(MAX_PACKET_SIZE_EPBULK),     // wMaxPacketSize (MSB)
        0                       // bInterval
    };
    return configDescriptor;
}
```

Four CDC specific descriptors are used in this configuration descriptor:

- 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.

Otherwise, this descriptor contains the 2 interfaces, as expected. The first one uses one IN endpoint and the second one two bulk endpoints.

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, flash chip should operate on the USB. Many devices use the MSD class to symplify data transfer. More common devices which use this class are hard-drives, USB-stick, MP3 or video player.

How the USB MSD class works ?

Endpoints used

This class uses three endpoints:

- Control endpoint: used for the enumeration step and to unhalt others endpoints when an error occured. The MSD protocol indeed specifies that the device must halt endpoints which are used to transfer data when an error occurs. So the endpoint 0 is also used to return in a functionnal state.
- The other two endpoints are bulk endpoints as big amount of data is exchanged between host and device. They are used for transferring commands and data over the bus.

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) isent by the host. The CBW describes several parameters of the transaction:

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)

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.

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)

State machine

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

- WAIT_CBW: waiting for CBW reception
- PROCESS_CBW: decode and execute command from the CBW
- SEND_CSW: a CSW will be send
- WAIT_CSW: waiting for CSW reception by the host
- ERROR: to report an error

TODO: put diagram of state machine

Implementation of the USBMSD class

Source code of main methods are in Annexe B: USBMSD class

Evolution of the state machine

Thanks to virtual functions called from USBHAL when an interrupt occurs, we are able to evolve our state machine.

Listing 3.10: State machine

```
// Called in ISR context when a data are received
bool USBMSD::EP2_OUT_callback() {
    uint16_t size = 0;
    uint8_t buf[MAX_PACKET_SIZE_EPBULK];
    read(EPBULK_OUT, buf, &size, MAX_PACKET_SIZE_EPBULK);
    switch (stage) {
```

```
        // the device has to decode the CBW received
        case READ_CBW:
            CBWDecode(buf, size);
            break;

        // the device has to receive data from the host
        case PROCESS_CBW:
            switch (cbw.CB[0]) {
                case WRITE10:
                case WRITE12:
                    memoryWrite(buf, size);
                    break;
                case VERIFY10:
                    memoryVerify(buf, size);
                    break;
            }
            break;

        // an error has occured: stall endpoint and send CSW
        default:
            stallEndpoint(EPBULK_OUT);
            csw.Status = CSW_ERROR;
            sendCSW();
            break;
    }

    //reactivate readings on the OUT bulk endpoint
    readStart(EPBULK_OUT, MAX_PACKET_SIZE_EPBULK);
    return true;
}

// Called in ISR context when data has been transferred
bool USBMSD::EP2_IN_callback() {
    switch (stage) {

        // the device has to send data to the host
        case PROCESS_CBW:
            switch (cbw.CB[0]) {
                case READ10:
                case READ12:
                    memoryRead();
                    break;
            }
            break;

        //the device has to send a CSW
        case SEND_CSW:
            sendCSW();
            break;

        // an error has occured
        case ERROR:
            stallEndpoint(EPBULK_IN);
            sendCSW();
            break;

        // the host has received the CSW -> we wait a CBW
        case WAIT_CSW:
            stage = READ_CBW;
            break;
    }
    return true;
}
```

According to the state, the device will decode a CBW, send a CSW, write or read to and from the momory.

Differents commands in the CBW

All commands sent by the host is aprt of the SCSI command architecture. In 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 CDB the target will return a status code byte. The method in annexe B which requires more information is decodeCBW(). Main command that can 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 block 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 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

3.2.7 Audio class

TODO

Chapter 4

References

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

USBHAL class

Listing 1: USBHAL::USBHAL()

```
USBHAL::USBHAL(void) {
    NVIC_DisableIRQ(USB_IRQn);

    // nUSB_CONNECT output
    LPC_IOCON->PIO0_6 = 0x00000001;

    // Enable clocks (USB registers, USB RAM)
    LPC_SYSCON->SYSAHBCLKCTRL |= CLK_USB | CLK_USBRAM;

    // Ensure device disconnected (DCON not set)
    LPC_USB->DEVCMDSTAT = 0;

    // to ensure that the USB host sees the device as
    // disconnected if the target CPU is reset.
    wait(0.3);

    // Reserve space in USB RAM for endpoint command/status list
    // Must be 256 byte aligned
    usbRamPtr = ROUND_UP_TO_MULTIPLE(usbRamPtr, 256);
    ep = (EP_COMMAND_STATUS *)usbRamPtr;
    usbRamPtr += (sizeof(EP_COMMAND_STATUS) * NUMBER_OF_LOGICAL_ENDPOINTS);
    LPC_USB->EPLISTSTART = (uint32_t)(ep) & 0xffffff00;

    // Reserve space in USB RAM for Endpoint 0
    // Must be 64 byte aligned
    usbRamPtr = ROUND_UP_TO_MULTIPLE(usbRamPtr, 64);
    ct = (CONTROL_TRANSFER *)usbRamPtr;
    usbRamPtr += sizeof(CONTROL_TRANSFER);
    LPC_USB->DATABUFSTART =(uint32_t)(ct) & 0xffc00000;

    // Setup command/status list for EP0
    ep[0].out[0] = 0;
    ep[0].in[0] =  0;
    ep[0].out[1] = CMDSTS_ADDRESS_OFFSET((uint32_t)ct->setup);

    // Route all interrupts to IRQ, some can be routed to
    // USB_FIQ if you wish.
    LPC_USB->INTROUTING = 0;

    // Set device address 0, enable USB device, no remote wakeup
    devCmdStat = DEV_ADDR(0) | DEV_EN | DSUS;
    LPC_USB->DEVCMDSTAT = devCmdStat;

    // Enable interrupts for device events and EP0
    LPC_USB->INTEN = DEV_INT | EP(EP0IN) | EP(EP0OUT);
    instance = this;

    //attach IRQ handler and enable interrupts
    NVIC_SetVector(USB_IRQn, (uint32_t)&_usbisr);
    NVIC_EnableIRQ(USB_IRQn);
}
```

Listing 2: USBHAL::realiseEndpoint()

```
bool USBHAL::realiseEndpoint(uint8_t endpoint, uint32_t maxPacket, uint32_t options) {
    uint32_t tmpEpRamPtr;

    if (endpoint > LAST_PHYSICAL_ENDPOINT) {
        return false;
    }

    // Not applicable to the control endpoints
    if ((endpoint==EP0IN) || (endpoint==EP0OUT)) {
        return false;
    }

    // Allocate buffers in USB RAM
    tmpEpRamPtr = epRamPtr;

    // Must be 64 byte aligned
```

```
    tmpEpRamPtr = ROUND_UP_TO_MULTIPLE(tmpEpRamPtr, 64);

    if ((tmpEpRamPtr + maxPacket) > (USB_RAM_START + USB_RAM_SIZE)) {
        // Out of memory
        return false;
    }

    // Allocate first buffer
    endpointState[endpoint].buffer[0] = tmpEpRamPtr;
    tmpEpRamPtr += maxPacket;

    if (!(options & SINGLE_BUFFERED)) {
        // Must be 64 byte aligned
        tmpEpRamPtr = ROUND_UP_TO_MULTIPLE(tmpEpRamPtr, 64);

        if ((tmpEpRamPtr + maxPacket) > (USB_RAM_START + USB_RAM_SIZE)) {
            // Out of memory
            return false;
        }

        // Allocate second buffer
        endpointState[endpoint].buffer[1] = tmpEpRamPtr;
        tmpEpRamPtr += maxPacket;
    }

    // Commit to this USB RAM allocation
    epRamPtr = tmpEpRamPtr;

    // Remaining endpoint state values
    endpointState[endpoint].maxPacket = maxPacket;
    endpointState[endpoint].options = options;

    // Enable double buffering if required
    if (options & SINGLE_BUFFERED) {
        LPC_USB->EPBUFCFG &= ~EP(endpoint);
    } else {
        // Double buffered
        LPC_USB->EPBUFCFG |= EP(endpoint);
    }

    // Enable interrupt
    LPC_USB->INTEN |= EP(endpoint);

    // Enable endpoint
    unstallEndpoint(endpoint);
    return true;
}
```

Listing 3: USBHAL::endpointReadResult()

```
EP_STATUS USBHAL::endpointReadResult(uint8_t endpoint, uint8_t *data, uint32_t *bytesRead) {

    uint8_t bf = 0;

    if (!(epComplete & EP(endpoint)))
        return EP_PENDING;
    else {
        epComplete &= ~EP(endpoint);

        //check which buffer has been filled
        if (LPC_USB->EPBUFCFG & EP(endpoint)) {
            // Double buffered (here we read the previous buffer which was used)
            if (LPC_USB->EPINUSE & EP(endpoint)) {
                bf = 0;
            } else {
                bf = 1;
            }
        }

        // Find how many bytes were read
        *bytesRead = (uint32_t) (endpointState[endpoint].maxPacket - BYTES_REMAINING(ep[PHY_TO_LOG↩
            (endpoint)].out[bf]));

        // Copy data
        USBMemCopy(data, ct->out, *bytesRead);
        return EP_COMPLETED;
    }
}
```

Listing 4: USBHAL::endpointWrite()

```
EP_STATUS USBHAL::endpointWrite(uint8_t endpoint, uint8_t *data, uint32_t size) {
    uint32_t flags = 0;
    uint32_t bf;

    // Validate parameters
    if (data == NULL) {
        return EP_INVALID;
```

```
    }

    if (endpoint > LAST_PHYSICAL_ENDPOINT) {
        return EP_INVALID;
    }

    if ((endpoint==EP0IN) || (endpoint==EP0OUT)) {
        return EP_INVALID;
    }

    if (size > endpointState[endpoint].maxPacket) {
        return EP_INVALID;
    }

    if (LPC_USB->EPBUFCFG & EP(endpoint)) {
        // Double buffered
        if (LPC_USB->EPINUSE & EP(endpoint)) {
            bf = 1;
        } else {
            bf = 0;
        }
    } else {
        // Single buffered
        bf = 0;
    }

    // Check if already active
    if (ep[PHY_TO_LOG(endpoint)].in[bf] & CMDSTS_A) {
        return EP_INVALID;
    }

    // Check if stalled
    if (ep[PHY_TO_LOG(endpoint)].in[bf] & CMDSTS_S) {
        return EP_STALLED;
    }

    // Copy data to USB RAM
    USBMemCopy((uint8_t *)endpointState[endpoint].buffer[bf], data, size);

    // Add options
    if (endpointState[endpoint].options & RATE_FEEDBACK_MODE) {
        flags |= CMDSTS_RF;
    }

    if (endpointState[endpoint].options & ISOCHRONOUS) {
        flags |= CMDSTS_T;
    }

    // Add transfer
    ep[PHY_TO_LOG(endpoint)].in[bf] = CMDSTS_ADDRESS_OFFSET( \
                                      endpointState[endpoint].buffer[bf]) \
                                      | CMDSTS_NBYTES(size) | CMDSTS_A | flags;

    return EP_PENDING;
}
```