Semester Project:

Bluetooth Remote Control

Introduction

In our master cursus at EPFL, we have the possibility to do a semester project on which we have to work for 12 hours per week. The subject of this project can be either one of the suggested by the LAP laboratory or one chosen by a student. The subject of the semester project described in this document has been chosen by me, as it was something I wanted to do for a long time.

Table of contents

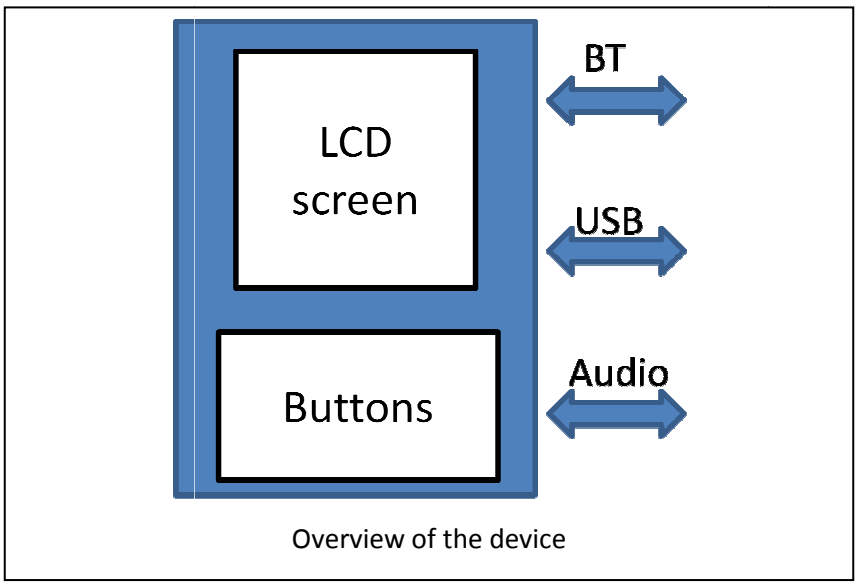
I) Aim of the project

The goal of this project is to create a complete and functional product, starting from scratch. Thus, you will see in this report many steps described, starting from the choice of components to the creation of DLLs for Windows.

We want to create a Bluetooth remote control having several functionalities:

- Display on a small LCD screen the tracks in the multimedia player's playlist; navigate through them using the keys on the remote.
- Display on the LCD the currently played track and its playing status.
- Headset functionality: it is possible to plug a standard microphone and headphones on the remote in order to use the remote control as a headset to make VoIP calls.
- External keyboard capability: when the remote control is connected to a computer using the appropriate USB cable, the remote will be detected as an external keyboard (using the HID class). Thus, if the user's computer doesn't have a Bluetooth chip, he will still be able to control his multimedia player using the special keys on the remote.
- Provide a platform with free tools: the compiler and the development environment are free, the compiled program can be sent to the microcontroller by using the USB bootloader. Thus, anyone can develop on the remote without spending money on the development tools.
- Good autonomy: the remote control will be powered using a lithium battery which will be charged when the remote is connected to a computer, using USB power.

Thus, the only part of the whole system on which this project relies is the Windows Bluetooth system drivers, since we don't take care of the low level parts of the Bluetooth communication. This project uses the serial port profile on top of the Bluetooth stack on both sides.



Overview of the device

II) Components choice

Here we will explain in details which components are chosen for this project and why.

a) Microcontroller

As told before, we need a microcontroller having an USB controller and a pre-embedded USB bootloader. I chose an 8-bit microcontroller from the AVR family: the AT90USB1287. It will be running at 8Mhz, has 128kB of flash memory which will be used to store the graphic library for the LCD screen, 8kB of ram memory, an embedded USB bootloader and it is possible to compile our program with avr-gcc before sending it to the flash memory.

b) Bluetooth chip

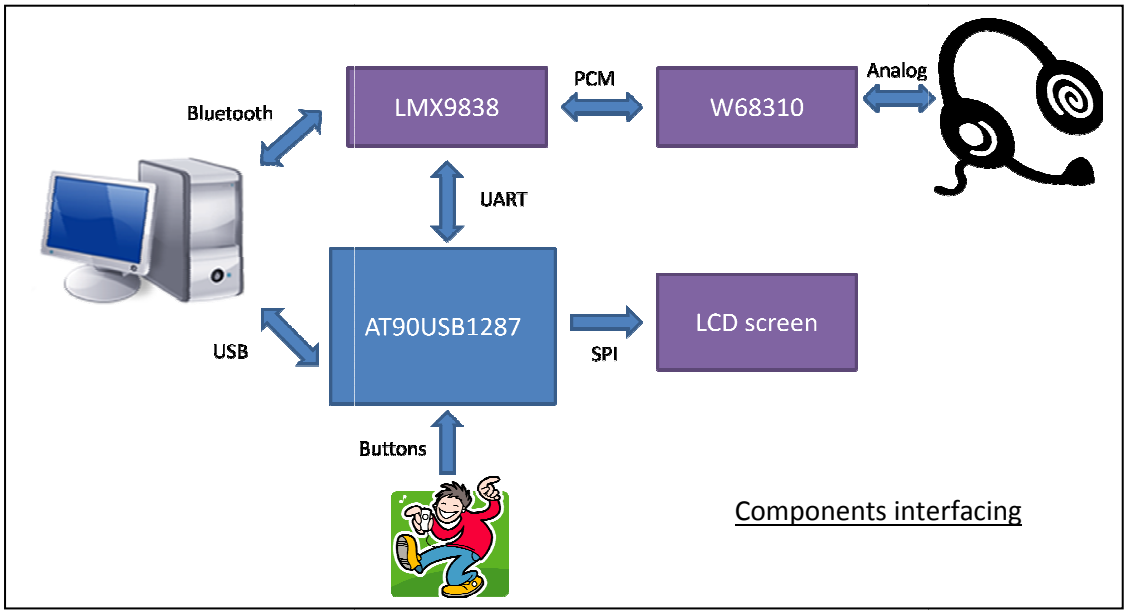
To avoid antenna dimensioning problems and parasites handling, I chose a component having an embedded antenna. This component is the LMX9838 from National Semiconductors. It is a Bluetooth to UART converter with an embedded Bluetooth stack. It handles several profiles: General Access Profiles (GAP), Service Discovery Application Profile (SDAP) and the Serial Port Profile (SPP). It is also possible to add a headset profile in it since the LMX9838 has a PCM controller which will be used to route audio data to an audio codec.

c) Audio codec

The LMX9838 supports 4 different audio codecs. I chose one of them: the Winbond W681310. It is a low cost, low current consumption PCM audio codec.

d) LCD screen

Since our microcontroller doesn't have an LCD controller, we need an LCD screen that we can communicate with a serial bus. I found the LCD screen used for all low cost Nokia cellular phones: it is a 130*130 pixels LCD controlled using an SPI bus. Even if the communication is made using a 9bits packet format, our microcontroller will be able to communicate with it using its SPI bus controller.



Components interfacing

III) Power handling

Since this remote control will be powered by a Lithium-Ion battery, we have to take care not to consume a lot of power. Here are described the different power save modes of every component.

a) Microcontroller

The AT90USB1287 has several power-save modes. The one used in this project is the "power-down" power-save mode. It is the lowest power consumption mode and will be activated when the remote has not been used for a long time and no USB and Bluetooth communications are active. The microcontroller will wake up when the user will press the menu key on the remote.

b) Bluetooth chip

In this project the LMX9838 will use an external crystal to consume less power. It has 4 different power modes that will use depending on how the remote is currently used

c) Audio codec

The W681310 has a power-enable pin. Thus, we will only activate the audio codec when an audio link has been established.

d) LCD screen

The LCD backlight is the part that consumes the more power. It will be enabled for a short time when the user is pressing any key on the remote control. The LCD screen controller has a power down mode where the LCD pixels driving circuit is turned off.

Using the power-saving strategy we obtain these power consumptions:

- Power save mode : 17.3 mW
- Backlight off, headset not plugged, no BT link : 94.4 mW
- Backlight off, headset plugged, no BT link : 95.2 mW
- Backlight on, headset not plugged, no BT link : 277 mW
- Backlight off, headset not plugged, data link : 118 mW
- Backlight on, headset not plugged, data link : 307 mW
- Backlight off, headset plugged, data & audio link : 177 mW
- Backlight on, headset plugged, data & audio link : 347 mW

We can see that by using all the power save modes of every component we can reduce the power consumption by a factor of **5.5** compared to the lowest power consumption use mode. If the remote is not used and charged for a long time, the battery won't be fully discharged since the 3.3V voltage regulator is not working when the voltage is lower than 3.0V. It is the same for the step-up for the LCD backlight.

IV) Work done

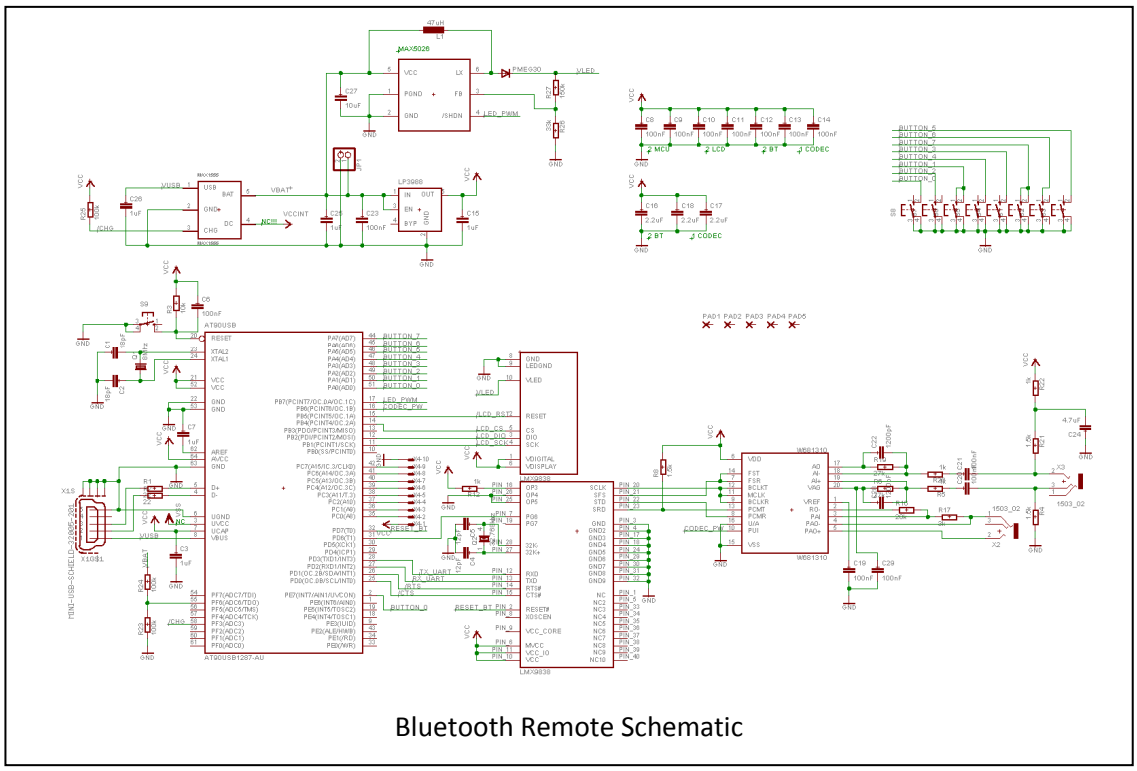
Here are described the different steps of the development of the Bluetooth Remote Control.

a) Components choice

As described in the chapters above, the first step of our project was to find the correct chips that will suit our project needs. A careful check must be done on the buses and packet format used between the different components.

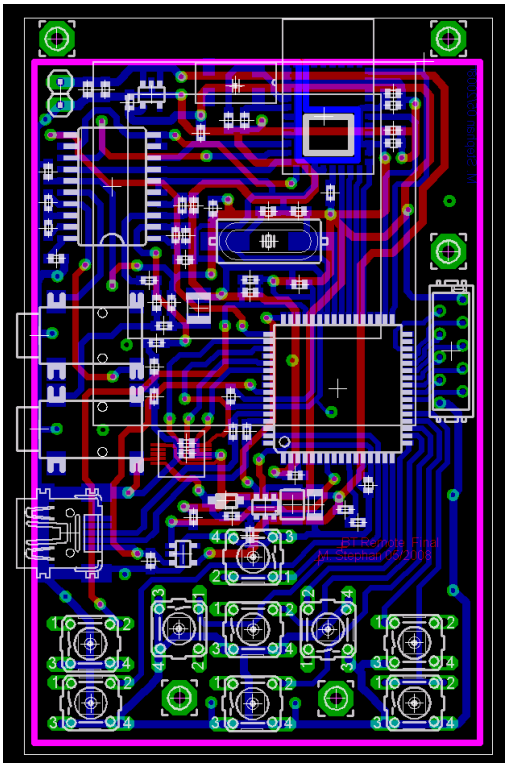
b) Components placement and routing

As I had a precise idea of the Bluetooth general aspect and design, I chose to take care of the printed circuit board creation.



Bluetooth Remote Schematic

Three printed circuit board versions have been done in order to obtain our final result.



Final printed circuit board

c) Firmware creation

The main part of this project development is the creation of the remote's firmware (6500 lines). Here are the several things done:

- LCD screen interfacing
- Keys interfacing (different modes)
- Bluetooth module interfacing
- Graphic library creation: lines, text, pictures, scrolling…
- Actions needed at first boot: BT profile addition, name & pin change, audio configuration...
- Protocol implementation for the dialog with the media player, using the BT serial port profile
- USB: implementation of a HID device
- Power handling: power save modes, battery voltage sensing

All the function and variables of the program are commented using doxygen. Thus, an HTML documentation is generated so the programmer can exactly know how the general program works.

d) Plugin creation

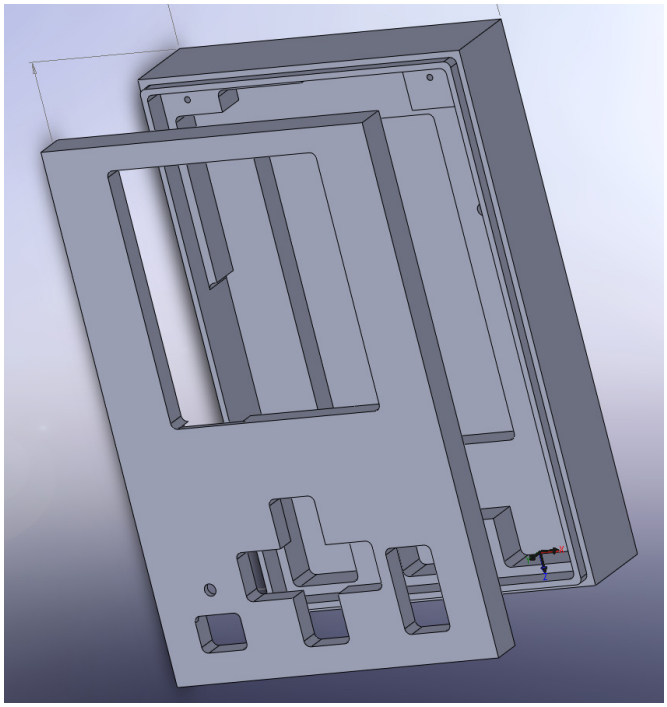
In parallel with the firmware creation, the media player plugin was programmed. The plugin is actually a Windows DLL created using Microsoft Visual Studio; the media player used is Winamp. When Winamp is launched, it automatically runs an init function in our DLL located in its plugin directory. Thus, when this init function is called a thread is created, which will be listening to our serial port over Bluetooth, decode the requests and send the appropriate answers to the remote.

```
winampGeneralPurposePlugin plugin =
{
        GPPHDR_VER,                             //
        "remote plugin",                        // Description
        init,                                   // Initialization function
        config,                                 // Configuration function
        quit,                                   // Deinitialization function
        0,
        0
};
```

Object returned to Winamp so it knows our program is a plugin

e) Box creation

As we wanted to create a complete product, a box was designed using Solidworks and produced using EPFL's CNC machines.



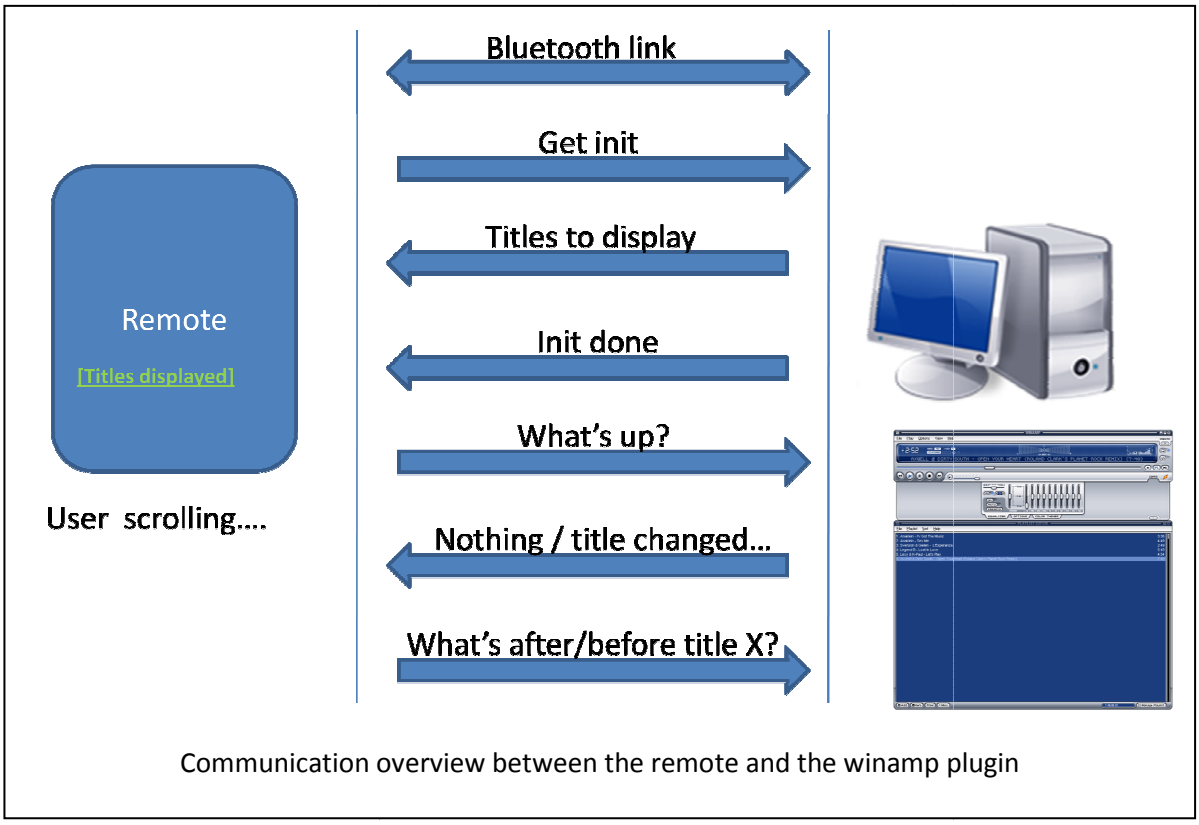
3D box model

V) Communication overview

Once the media player is launched, a Bluetooth communication is established between the remote and the user computer. When the remote detects that the communication has been established, it automatically sends a “get_init” message to our plugin in order to have the list of titles to display on its LCD screen. It will then periodically send “get_update” packets to know if any changes occurred (title / playlist change, currently played status update…).

The remote only stores 16 titles in his memory. Since we can only display 8 titles on the screen, 8 others are cached so they can be directly displayed when the user browse through the titles. When the remote sees that its buffer begins to be empty, it then asks the plugin for the missing titles.

A home-made protocol has been implemented using the serial port. It takes care of packet fragmenting that can occur, has a command identifier field and a variable packet size.



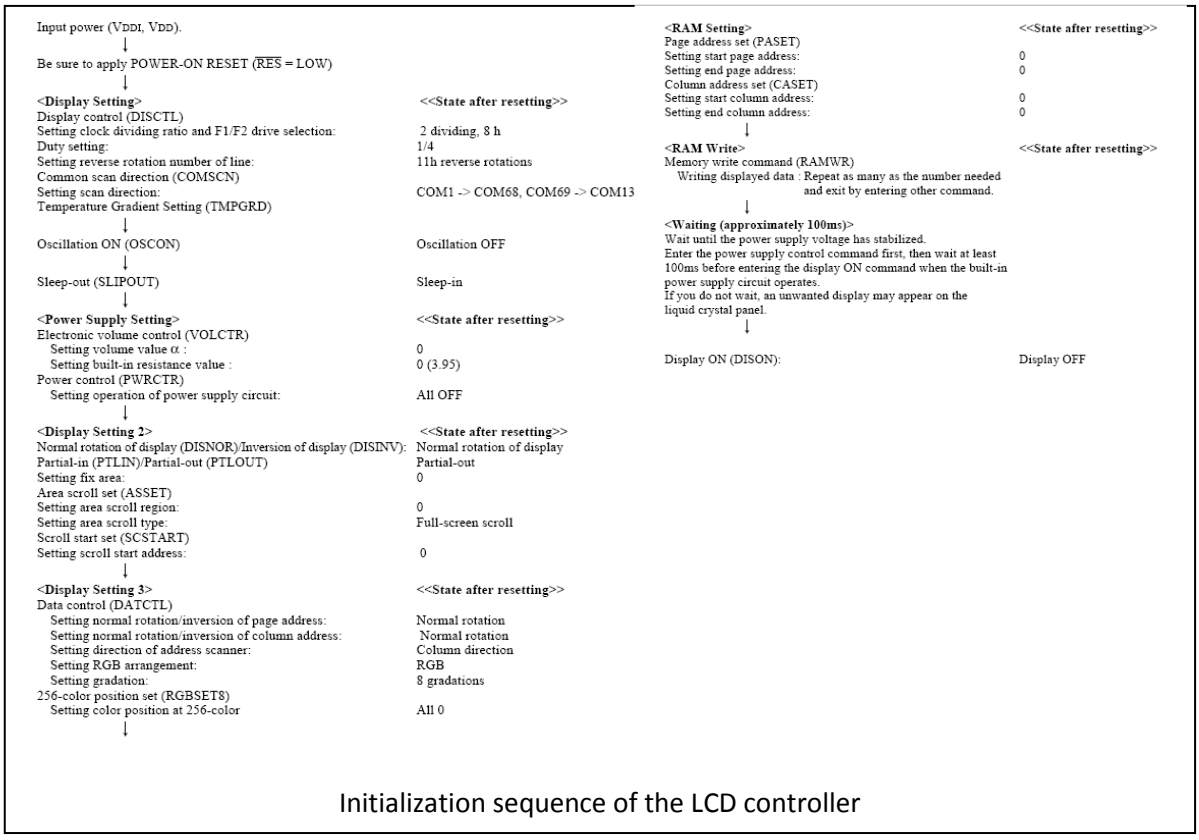
Communication overview between the remote and the winamp plugin

VI) Work done – Firmware details

Here we will describe in details the work mentioned above: packets, signals…

a) LCD screen interfacing

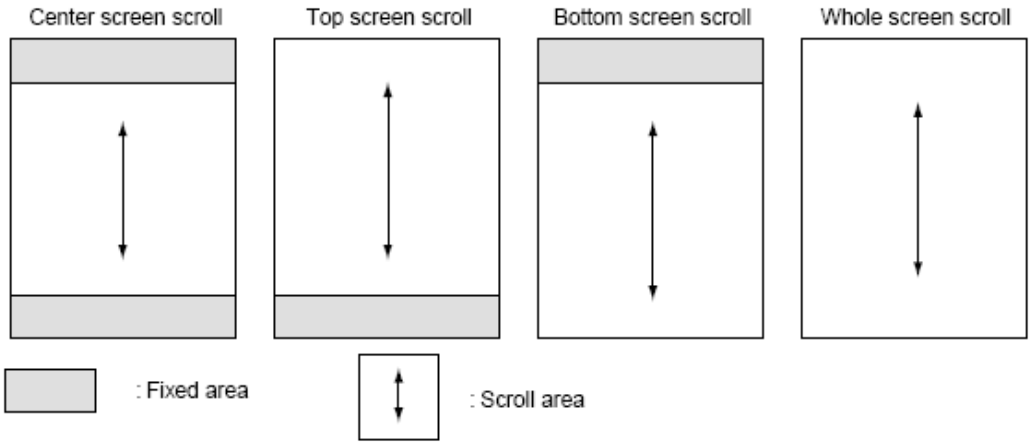
As previously told, the LCD screen has an integrated controller, the EPSON S1D15G10. We can communicate with it using the SPI bus. The packet is 9 bits long, so since our microcontroller SPI bus controller only handles 8 bit packets, we will generate the ninth bit by hand. No data can be received by the LCD controller; the first bit of a packet is used to specify whether it is a data or command packet. A precise procedure is given to start the lcd screen.



Initialization sequence of the LCD controller

The LCD controller has an integrated memory where the picture to be displayed on the screen is stored. Pixels color is coded on 12 bits but a palette can be used to give a pixel color coded on 8 bits. This project uses this mode.

The LCD controller also handles screen scrolling. Thus, it will be used when the user navigates through the titles displayed on the screen.



Scrolling modes on the EPSON LCD controller

b) Bluetooth module interfacing

To communicate with the LMX9838, we use the UART controller on our microcontroller.

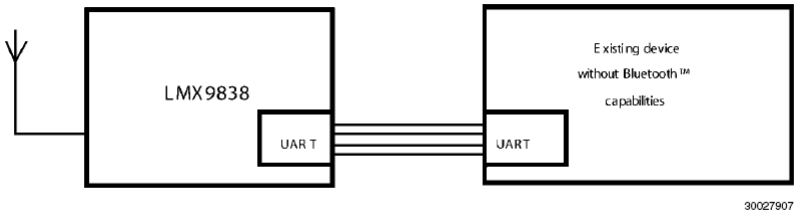


FIGURE 8. Bluetooth Functionality

TABLE 13. Package Framing

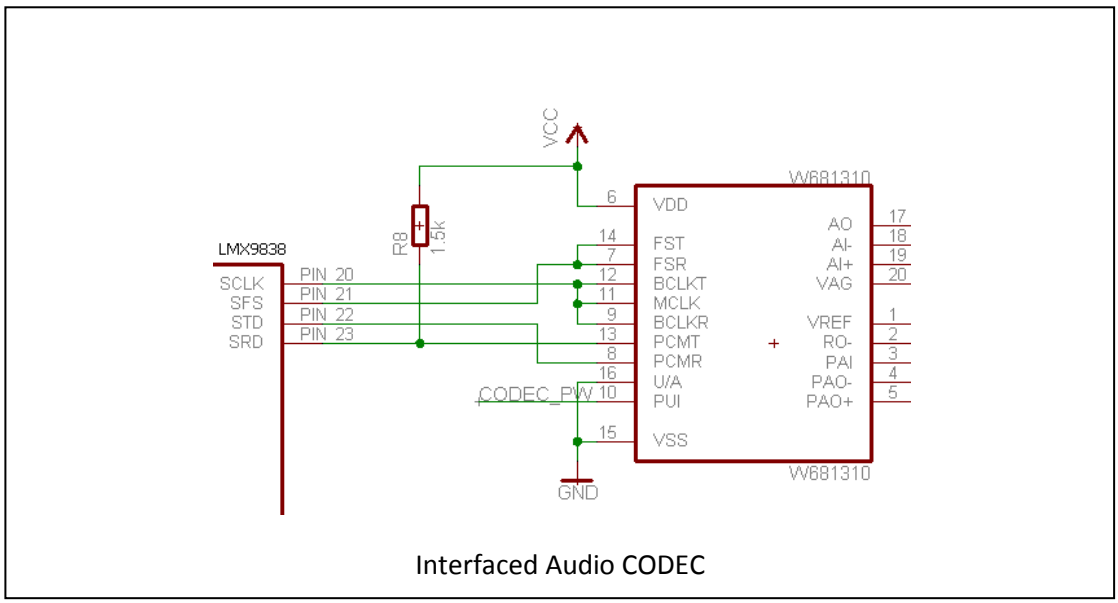
Start Delimiter	Packet Type ID	Opcode	Data Length	Check sum	Packet Data	End Delimiter
1 Byte	1 Byte	1 Byte	2 Bytes	1 Byte	<Data Length> Bytes	1 Byte
	---------- Checksum ----------					

Packet format for the LMX9838

The LMX9838 has a special packet format we have to respect in order to communicate with it. Each command sent to the Bluetooth module is confirmed by the appropriate message. As our controller can't handle the /RTS and /CTS signals, we use the UART in 2 wire mode, so we have to be sure that our device is fast enough to not miss any data.

c) Audio CODEC interfacing

The communication between the Bluetooth module and the Audio codec is done using a PCM bus. Once we told the LMX9838 which of the Audio CODECs it supports we use, it will automatically use the right packet format. Thus, we only have to connect the right signals between the two components.

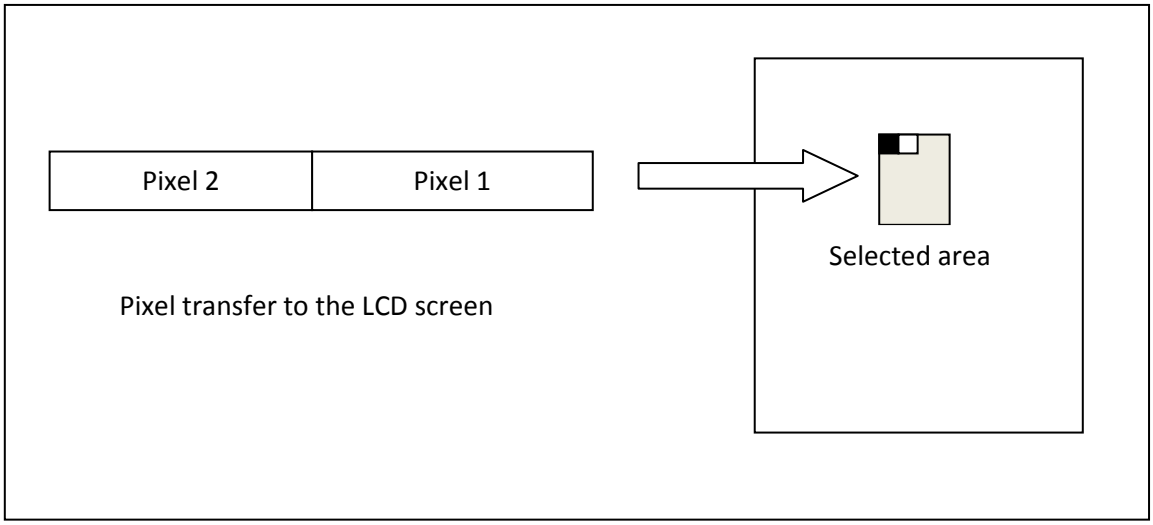


Interfaced Audio CODEC

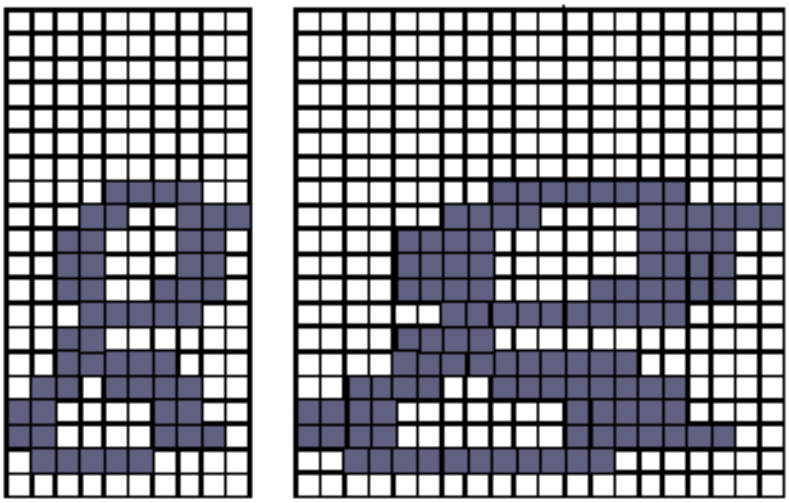
The Frame Sync Receive and Frame Sync Transmit inputs are connected together, as well as the receive / master / transmit clock input as data are sent / received simultaneously.

d) Graphic library

Several functions have been implemented in order to draw graphics needed for a correct graphical interface. To write pixels on the LCD screen, we previously have to define to the LCD controller a square on the screen where we will draw. Pixel data are sent one after the other until the square has been filled. Thus, we don't need to redraw the whole screen when something changes, only the interesting part.



One font has been implemented in this project, in three different sizes. This font is a bitmap font, stored in memory in a binary format (‘1’ for black, ‘0’ for white) in order to not waste space.

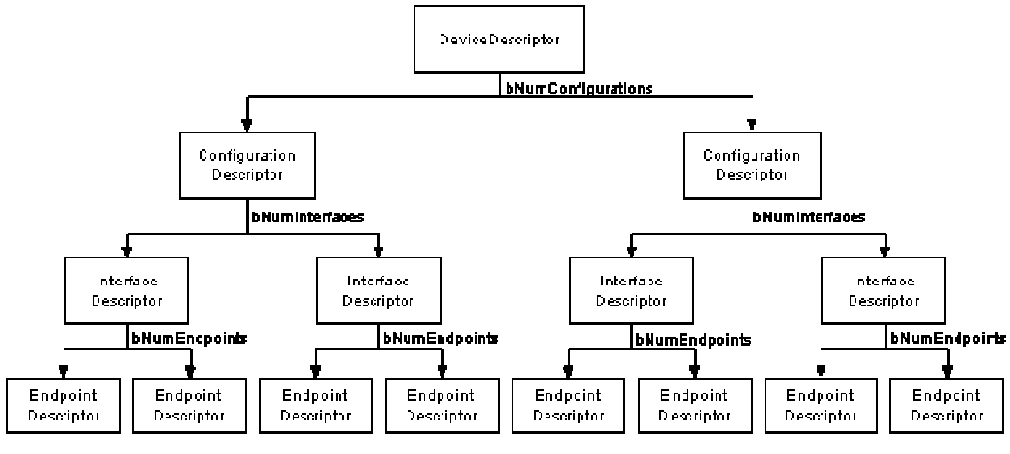


Binary table containing our font in different sizes

Thus, it is possible to write complete string on the LCD screen by displaying one character after the other.

e) HID device

As we want our remote control to be recognized as an external USB keyboard, we have to implement the whole hierarchy of descriptors. When the USB device will be plugged, the operating system will ask for them from the top to the bottom.



USB Descriptors tree

	Reset (837.4 us)						0,000 000…
	Power ON						0,001 785…
	Suspended (1.5 ms)						0,002 745…
	Reset (10.8 ms)						0,004 235…
	High speed Detection Handshake			TIMEOUT			0,014 246…
	Suspended (1.6 ms)						0,015 839…
	Reset (12.1 ms)						0,017 423…
	High speed Detection Handshake			TIMEOUT			0,027 434…
⊞	GetDescriptor (Device)	0 (1)	0	OK	FS	18 bytes (12 01 10 01 00 00 00 20…)	0,074 253…
	Reset (11.1 ms)						0,079 354…
	High speed Detection Handshake			TIMEOUT			0,089 364…
⊞	SetAddress (1)	0 (1)	0	OK	FS	No data	0,116 966…
⊞	GetDescriptor (Device)	1	0	OK	FS	18 bytes (12 01 10 01 00 00 00 20…)	0,163 969…
⊞	GetDescriptor (Configuration)	1	0	OK	FS	9 bytes (09 02 3B 00 02 01 04 A0…)	0,168 969…
⊞	GetDescriptor (String lang IDs)	1	0	OK	FS	4 bytes (04 03 09 04)	0,173 970…
⊞	GetDescriptor (String iSerialNumber)	1	0	OK	FS	8 bytes (08 03 31 00 2E 00 30 00)	0,179 970…
⊞	GetDescriptor (Configuration)	1	0	OK	FS	59 bytes (09 02 3B 00 02 01 04 A0…)	0,185 970…
⊞	GetDescriptor (String lang IDs)	1	0	OK	FS	4 bytes (04 03 09 04)	0,192 971…
⊞	GetDescriptor (String iProduct)	1	0	OK	FS	34 bytes (22 03 42 00 6C 00 75 00…)	0,198 972…
⊞	GetDescriptor (String lang IDs)	1	0	OK	FS	4 bytes (04 03 09 04)	0,205 972…
⊞	GetDescriptor (String iProduct)	1	0	OK	FS	34 bytes (22 03 42 00 6C 00 75 00…)	0,211 972…
⊞	GetDescriptor (Device)	1	0	OK	FS	18 bytes (12 01 10 01 00 00 00 20…)	0,648 006…
⊞	GetDescriptor (Configuration)	1	0	OK	FS	9 bytes (09 02 3B 00 02 01 04 A0…)	0,653 006…
⊞	GetDescriptor (Configuration)	1	0	OK	FS	59 bytes (09 02 3B 00 02 01 04 A0…)	0,658 006…
⊞	SetConfiguration (1)	1	0	OK	FS	No data	0,664 006…
⊞	GetDescriptor (String iInterface)	1	0	OK	FS	3 bytes (03 03 00)	0,702 010…
⊞	GetDescriptor (String iInterface)	1	0	OK	FS	3 bytes (03 03 00)	0,722 011…
⊞	SetIdle (All, Indefinite)	1	0	OK	FS	No data	0,731 012…
⊞	GetDescriptor (Report)	1	0	OK	FS	23 bytes (05 01 09 06 A1 01 05 07…)	0,734 011…
⊞	SetIdle (All, Indefinite)	1	0	OK	FS	No data	0,754 013…
⊞	GetDescriptor (Report)	1	0	OK	FS	50 bytes (05 0C 09 01 A1 01 85 01…)	0,757 014…

USB device identification when plugged

When an USB device is plugged, the host sends to the device a “set address” packet containing the address that it wants to assign to it. Once this packet has been acknowledged, the device descriptor is asked.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of the Descriptor in Bytes (18 bytes)
1	bDescriptorType	1	Constant	Device Descriptor (0x01)
2	bcdUSB	2	BCD	USB Specification Number which device complies too.
4	bDeviceClass	1	Class	Class Code (Assigned by USB Org) If equal to Zero, each interface specifies it's own class code If equal to 0xFF, the class code is vendor specified. Otherwise field is valid Class Code.
5	bDeviceSubClass	1	SubClass	Subclass Code (Assigned by USB Org)
6	bDeviceProtocol	1	Protocol	Protocol Code (Assigned by USB Org)
7	bMaxPacketSize	1	Number	Maximum Packet Size for Zero Endpoint. Valid Sizes are 8, 16, 32, 64
8	idVendor	2	ID	Vendor ID (Assigned by USB Org)
10	idProduct	2	ID	Product ID (Assigned by Manufacturer)
12	bcdDevice	2	BCD	Device Release Number
14	iManufacturer	1	Index	Index of Manufacturer String Descriptor
15	iProduct	1	Index	Index of Product String Descriptor
16	iSerialNumber	1	Index	Index of Serial Number String Descriptor
17	bNumConfigurations	1	Integer	Number of Possible Configurations

Device descriptor	
bcdUSB	1.1
bDeviceClass	Class information at interface level
bMaxPacketSize0	32
idVendor	CHERRY GmbH
idProduct	0x0004
bcdDevice	1.0
iManufacturer	1
iProduct	2 "Bluetooth Remote"
iSerialNumber	3 "1.0"

Device descriptor composition (left), our device descriptor (right)

For reasons that will see later, we took the Vendor and Product ID from a Cherry multimedia keyboard. Once the device descriptor has been sent, the other descriptors are asked.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	Configuration Descriptor (0x02)
2	wTotalLength	2	Number	Total length in bytes of data returned
4	bNumInterfaces	1	Number	Number of Interfaces
5	bConfigurationValue	1	Number	Value to use as an argument to select this configuration
6	iConfiguration	1	Index	Index of String Descriptor describing this configuration
7	bmAttributes	1	Bitmap	D7 Reserved, set to 1. (USB 1.0 Bus Powered) D6 Self Powered D5 Remote Wakeup D4..0 Reserved, set to 0.
8	bMaxPower	1	mA	Maximum Power Consumption in 2mA units

Configuration descriptor	
bNumInterface	2
bConfigurationValue	1
iConfiguration	4
bmAttributes. RemoteWakeup	Supported
bmAttributes. SelfPowered	No, Bus Powered
bMaxPower	100 mA

Configuration descriptor composition (left), our configuration descriptor (right)

In our configuration descriptor, we define the power needed for our device, several attributes and the number of interfaces. Our remote control will be detected as a device which has 2 interfaces. One interface will be used for the normal keys (a to z, left/right/up/down arrow) and another for the multimedia keys (play/pause, next / prev track, increase/decrease volume).

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes (9 Bytes)
1	bDescriptorType	1	Constant	Interface Descriptor (0x04)
2	bInterfaceNumber	1	Number	Number of Interface
3	bAlternateSetting	1	Number	Value used to select alternative setting
4	bNumEndpoints	1	Number	Number of Endpoints used for this interface
5	bInterfaceClass	1	Class	Class Code (Assigned by USB Org)
6	bInterfaceSubClass	1	SubClass	Subclass Code (Assigned by USB Org)
7	bInterfaceProtocol	1	Protocol	Protocol Code (Assigned by USB Org)
8	iInterface	1	Index	Index of String Descriptor Describing this interface

Interface descriptor	
bInterfaceNumber	0
bAlternateSetting	0
bNumEndpoints	1
bInterfaceClass	Human Interface Device
bInterfaceSubClass	Boot Interface
bInterfaceProtocol	Keyboard
iInterface	5

Interface descriptor	
bInterfaceNumber	1
bAlternateSetting	0
bNumEndpoints	1
bInterfaceClass	Human Interface Device
iInterface	6

Interface descriptor composition (left), our 2 interface descriptors (right)

For an HID device, to every interface descriptor is associated an endpoint and an HID descriptor, so the host knows how to communicate with the device.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes (7 bytes)
1	bDescriptorType	1	Constant	Endpoint Descriptor (0x05)
2	bEndpointAddress	1	Endpoint	Endpoint Address Bits 0..3b Endpoint Number. Bits 4..6b Reserved. Set to Zero Bits 7 Direction 0 = Out, 1 = In (Ignored for Control Endpoints)
3	bmAttributes	1	Bitmap	Bits 0..1 Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt Bits 2..7 are reserved. If Isochronous endpoint, Bits 3..2 = Synchronisation Type (Iso Mode) 00 = No Synchonisation 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4 = Usage Type (Iso Mode) 00 = Data Endpoint 01 = Feedback Endpoint 10 = Explicit Feedback Data Endpoint 11 = Reserved
4	wMaxPacketSize	2	Number	Maximum Packet Size this endpoint is capable of sending or receiving
6	bInterval	1	Number	Interval for polling endpoint data transfers. Value in frame counts. Ignored for Bulk & Control Endpoints. Isochronous must equal 1 and field may range from 1 to 255 for interrupt endpoints.

Endpoint descriptor	
bEndpointAddress	1 IN
bmAttributes. TransferType	Interrupt
wMaxPacketSize	3 bytes
bInterval	10 frames (10 ms)

Endpoint descriptor	
bEndpointAddress	2 IN
bmAttributes. TransferType	Interrupt
wMaxPacketSize	3 bytes
bInterval	10 frames (10 ms)

Endpoint descriptor composition (left), our 2 endpoints descriptors (right)

Several transfer types exist for each endpoint. The transfer type used here is Interrupt since we will periodically check if the user pressed any key. Once the host knows how to communicate with the device, it will ask each interface for its HID report descriptor describing which keys are on the keyboard.

```
USAGE_PAGE (Generic Desktop)                                  05 01
USAGE (Keyboard)                                              09 06
COLLECTION (Application)                                      A1 01
  REPORT_COUNT (6)                                            95 06
  REPORT_SIZE (8)                                             75 08
  LOGICAL_MINIMUM (0)                                         15 00
  LOGICAL_MAXIMUM (101)                                       25 65
  USAGE_PAGE (Keyboard)                                       05 07
  USAGE_MINIMUM (Reserved (no event indicated))               19 00
  USAGE_MAXIMUM (Keyboard Application)                        29 65
  INPUT (Data,Ary,Abs)                                        81 00
END_COLLECTION                                                C0
```

HID report descriptor for the keyboard interface

Above is the report descriptor used for the keyboard part of our USB device. We can see that 6*8 = 48 bits will be returned indicating which keys are pressed. A key pressed is stored in one of the 6 8bits slots sent every 10ms to the host.

f) Protocol used

As previously said, a protocol has been implemented in this project, using the serial port. The packet format is the following:

Packet length	*Packet type*	*Packet data*

Packets used for the communication between the remote and the plugin

It can occur that the packets we receive are fragmented. Thus, we use the "packet length" field to reconstruct them.

VII) Improvements

Here is the list of the different things that could be improved:

- The printed circuit board size could be reduced. Indeed, because the first prototypes were made by myself, I chose to use wide wires to easily solder the different components.
- The graphic library can be improved. Only one font is available, simple graphic effects and shapes are available.
- The packaging can be better. The current packaging is a simple box.

VIII) Evolution

Now we have created a complete product, we can wonder about its evolution. Because all the tools needed to develop on this platform are free, everyone can create their own firmware. This project has Bluetooth & USB interfaces, an LCD screen and an expansion port so many different things can be done using this platform.

IX) Conclusion

In a short amount of time, I managed to create a complete and functional product using all the means given at my disposal at EPFL. This project was very interesting for me as it needed many different skills in many different domains.

Final version of the remote

Annexes