Project documentation

project title: *Keyboard interfacing with android phone*

Team Members: **Aashish Gupta, Aditya Aggarwal, Chetan Chauhan, Lohit Jain**

Team Mentor: **Nitish Srivastava, Kapil Dev Advani**

Basic Aim:

To interface any hardware usb keyboard with any android phone so as to write directly on mobile screen without using onscreen tap keyboard. For this we interfaced our usb keyboard with arduino mega adk board and used AUBTM-20 bluetooth module for pairing it with android for sending the data packets.

How did we get the idea

The baisc idea evolved from our fascination towards the working of Basic HID USB devices. We thought of developing a hardware which would replicate and process the USB signals and in turn act as Hardware Key Logger.
When we discussed this with Coordinators they appreciated our fascination and instead adviced us to try making something more useful for common man like a hardware for using keyboard for other

purpose. So it finally became developing hardware which interfaces any USB keyboard with Android mobile via Bluetooth. This will aid in creating a more convenient way of writing on Android Phone.

Theory

USB Protocol

USB data is sent in packets Least Significant Bit (LSB) first.
There are 4 main USB packet types :Token, Data, Handshake and Start of Frame.
Each packet is constructed from different field types, namely SYNC, PID, Address, Data, Endpoint, CRC and EOP.
The packets are then bundled into frames to create a USB message

The USB token packet is used to access the correct address and endpoint. It is constructed with the SYNC, PID, an 8 bit PID field, followed by a 7 bit address, followed by a 4 bit endpoint and a 5 bit CRC.
Both the address and endpoint field must be correctly decoded for correct operation.
The data packet may be of variable length, dependent upon the data. However, the data field will be an integral number of bytes.

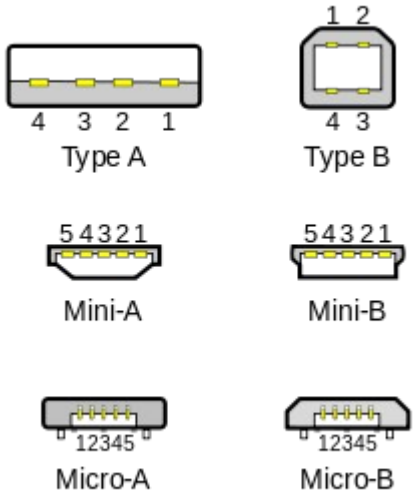
Human interface devices (HIDs)

Main article: USB human interface device class

Joysticks, keypads, tablets and other human-interface devices are also progressively migrating from MIDI, and PC game port connectors to USB.[*citation needed*] USB mice and keyboards can usually be used with older computers that have PS/2 connectors with the aid of a small USB-to-PS/2 adapter. Such adaptors contain no logic circuitry: the hardware in the USB keyboard or mouse is designed to detect whether it is connected to a USB or PS/2 port, and communicate using the appropriate protocol. Converters also exist to allow PS/2 keyboards and mice (usually one of each) to be connected to a USB port. These devices present two HID endpoints to the system and use a microcontroller to perform bidirectional translation of data between the two standards.[*citation needed*]

Physical appearance

Pinouts of Standard, Mini, and Micro USB plugs. The white areas in these drawings represent hollow spaces. As the plugs are shown here, the USB logo (with optional letter A or B) is on the top of the overmold in all cases.[23]



Micro-B USB 3.0 compatible (cable/male end)
USB 2.0 connector on the side of the specification standard micro USB 3.0 connector are aligned pin-minute increase in the standard.
No.1: power (VBUS)
No.2: USB 2.0 differential pair (D−)
No.3: USB 2.0 differential pair (D+)
No.4: USB OTG ID for identifying lines
No.5: GND
No.6: USB 3.0 signal transmission line (−)
No.7: USB 3.0 signal transmission line (+)
No.8: GND
No.9: USB 3.0 signal receiving line (−)
No.10: USB 3.0 signal receiving line (+)

USB 1.x/2.0 standard pinout

Pin	Name	Cable color	Description
1	VBUS	Red	+5 V
2	D−	White	Data −
3	D+	Green	Data +
4	GND	Black	Ground

USB 1.x/2.0 Mini/Micro pinout

Pin	Name	Cable color	Description
1	VBUS	Red	+5 V
2	D−	White	Data −
3	D+	Green	Data +
4	ID	None	Permits distinction of host connection from slave connection * host: connected to Signal ground * slave: not connected
5	GND	Black	Signal ground

Arduino mega adk for android

The Arduino ADK is a microcontroller board based on the ATmega2560 (datasheet). It has a USB host interface to connect with Android based phones, based on the MAX3421e IC. It has 54 digital input/output pins (of which 14 can be used as PWM outputs), 16 analog inputs, 4 UARTs (hardware serial ports), a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button.

Each of the 50 digital pins on the ADK can be used as an input or output, using pinMode(), digitalWrite(), and digitalRead() functions. They operate at 5 volts. Each pin can provide or receive a maximum of 40 mA and has an internal pull-up resistor (disconnected by default) of 20-50 kOhms. In addition, some pins have specialized functions.

The USB host interface given by MAX3421E IC allows the ADK Arduino to connect and interact to any type of device that have a USB port. For example, allows you to interact with many types of phones, controlling Canon cameras, interfacing with keyboard, mouse and games controllers as Wiimote and PS3.

AUBTM-20 Bluetooth Module

AUBTM-20 works on UART communication.

AUBTM-20 UART provides the main interface to exchange data with other host system using the RS232 protocol. An external commands set is provided for the host system to control and configure AUBTM-20.
Four signals are provide for UART function. TX and RX transmit data between AUBTM-20 and the host. UART is initially configured to work at 9600 bps baudrate, 8-bit, no parity and 1 stop bit. The host could reconfigure the UART by issuing command.
In arduino adk board ,we have to declare two pins for UART communication.
TX of Bluetooth module is to be connected to RX of arduino adk and RX of Bluetooth module to TX of arduino adk as th data transmitted by the Bluetooth module is received by the receiving end of arduino adk.

Android application development

download four tier software development system for android app
refer the following link for guidance -
www.niktechs.wordpress.com.
For app development

2.
The Android project we are going to write is going to have to do a few things:
1. Open a bluetooth connection
2. Send data
3. Listen for incoming data
4. Close the connection
But before we can do any of that we need to take care of a few pesky little details. First, we need to pair the Arduino and Android devices. You can do this from the Android device in the standard way by opening your application drawer and going to **Settings**. From there open **Wireless and network.** Then **Bluetooth settings.** From here just scan for devices and pair it like normal. If it asks for a pin it's 8888.
You shouldn't need to do anything special from the Arduino side other than to have it turned on. Next we need to tell Android that we will be working with bluetooth by adding this element to the **<manifest>** tag inside the **AndroidManifest.xml** file:
<uses-permission android:name="android.permission.BLUETOOTH" />
Alright, now that we have that stuff out of the way we can get on with opening the bluetooth connection. To get started we need a **BluetoothAdapter** reference from Android. We can get that by
calling
```
BluetoothAdapter.getDefaultAdapter();
```
The return value of this will be null if the device does not have bluetooth capabilities. With the adapter you can check to see if bluetooth is enabled on the device and request that it be turned on if its not with this code:
```
if(!mBluetoothAdapter.isEnabled())
{
Intent enableBluetooth = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
startActivityForResult(enableBluetooth, 0);
}
```

3.
Now that we have the bluetooth adapter and know that its turned on we can get a reference to our Arduino's bluetooth device with this code:
```
Set pairedDevices = mBluetoothAdapter.getBondedDevices();
if(pairedDevices.size() > 0)
{
for(BluetoothDevice device : pairedDevices)
{
if(device.getName().equals("AUBTM-20")) //Note, you will
need to change this to match the name of your device
{
mmDevice = device;
break;
}
}
}
```

4.
Armed with the bluetooth device reference we can now connect to it using this code:

```
UUID uuid = UUID.fromString("00001101-0000-1000-8000-
00805f9b34fb"); //Standard SerialPortService ID
mmSocket = mmDevice.createRfcommSocketToServiceRecord(uuid);
mmSocket.connect();
mmOutputStream = mmSocket.getOutputStream();
mmInputStream = mmSocket.getInputStream();
```

This opens the connection and gets input and output streams for us to work with. You may have noticed that huge ugly UUID. Apparently there are standard UUID's for various devices and Serial Port's use that one.

If everything has gone as expected, at this point you should be able to connect to your Arduino from your Android device. The connection takes a few seconds to establish so be patient. Once the connection is established the red blinking light on the bluetooth chip should stop and remain off.

Closing the connection is simply a matter of calling close() on the input and output streams as well as the socket

Now we are going to have a little fun and actually send the Arduino some data from the Android device. Make yourself a text box (also known as an EditText in weird Android speak) as well as a send button. Wire up the send button's click event and add this code:

```
String msg = myTextbox.getText().toString();
msg += "\n";
mmOutputStream.write(msg.getBytes());
```

5.

With this we have one way communication established! Make sure the Arduino is plugged in to the computer via the USB cable and open the Serial Monitor from within the Arduino IDE. Open the application on your Android device and open the connection. Now whatever your type in the textbox will be sent to the Arduino over air and magically show up in the serial monitor window.

Sending data is trivial. Sadly, listening for data is not. Since data could be written to the input stream at any time we need a separate thread to watch it so that we don't block the main ui thread and cause Android to think that we have crashed. Also the data written on the input stream is relatively arbitrary so there isn't a simple way to just be notified of when a line of text shows up. Instead lots of short little packets of data will show up individually one at a time until the entire message is received. Because of this we are going to need to buffer the data in an array until enough of the data shows up that we can tell what to do. The code for doing this and the threading is a little be long winded but it is nothing too complicated.

First we will tackle the problem of getting a worker thread going. Here is the basic code for that:

```
final Handler handler = new Handler();
workerThread = new Thread(new Runnable()
{
public void run()
{
while(!Thread.currentThread().isInterrupted() && !
stopWorker)
{
//Do work
}
}
});
workerThread.start();
```

The first line there declares a Handler which we can use to update the UI, but more on that later. This code will start a new thread. As long as the run() method is running the thread will stay alive. Once the run() method completes and returns the thread will die. In our case, it is stuck in a while loop that will

keep it alive until our boolean **stopWorker** flag is set to true, or the thread is interrupted. Next lets talk
about actually reading data.
The input stream provides an **.available()** method which returns us how many (if any) bytes of data are
waiting to be read. Given that number we can make a temporary byte array and read the bytes into it like so:

```
int bytesAvailable = mmInputStream.available();
if(bytesAvailable > 0)
{
byte[] packetBytes = new
byte[bytesAvailable];
mmInputStream.read(packetBytes);
}
```

This gives us some bytes to work with, but unfortunately this is rather unlikely to be all of the bytes we need (or who know its might be all of them plus some more from the next command). So now we have do that pesky buffering thing I was telling you about earlier. The read buffer is just byte array that we can store incoming bytes in until we have them all. Since the size of message is going to vary we need to allocate enough space in the buffer to account for the longest message we might receive. For our purposes we are allocating 1024 spaces, but the number will need to be tailored to your specific needs. Alright, back to the code. Once we have packet bytes we need to add them to the read buffer one at a time until we run in to the end of line delimiter character, in our case we are using a newline character (Ascii code 10)

```
for(int i=0;i<bytesAvailable;i++)
{
byte b = packetBytes[i];
if(b == delimiter)
{
byte[] encodedBytes = new byte[readBufferPosition];
System.arraycopy(readBuffer, 0, encodedBytes, 0, encodedBytes.length);
final String data = new String(encodedBytes, "US-ASCII");
readBufferPosition = 0;
//The variable data now contains our full command
}
else
{
readBuffer[readBufferPosition++] = b;
}
}
```

At this point we now have the full command stored in the string variable **data** and we can act on it however we want. For this simple example we just want to take the string display it in on a on screen label. Sticking the string into the label would be pretty simple except that this code is operating under a worker thread and only the main UI thread can access the UI elements. This is where that Handler variable is going to come in. The handler will allow us to schedule a bit of code to be executed by main UI thread. Think of the handler as delivery boy who will take the code you wanted executed and deliver it to main UI thread so that it can execute the code for you. Here is how you can do that:

```
handler.post(new Runnable()
{
public void run()
{
myLabel.setText(data);
}
});
```

And that is it! We now have two way communication between the Arduino and an Android device! Plugin the Arduino and open the serial monitor. Run your Android application and open the bluetooth connection. Now what type in the textbox on the Android device will show up in the serial monitor window for the Arduino, and what you type in the serial monitor window will show up on the Android device.

Programming arduino

The Arduino ADK can be programmed with the Arduino software .The open-source Arduino environment makes it easy to write code and upload it to the i/o board. It runs on Windows, Mac OS X, and Linux. The environment is written in Java and based on Processing, avr-gcc, and other open source software.

As the Arduino uses USB host shield we must include the corresponding library in our code.
Our code:

```
#include <avr/pgmspace.h>
#include <SoftwareSerial.h>
#include <avrpins.h>
#include <max3421e.h>
#include <usbhost.h>
#include <usb_ch9.h>
#include <Usb.h>
#include <usbhub.h>
#include <avr/pgmspace.h>
#include <address.h>
#include <hidboot.h>

#include <printhex.h>
#include <message.h>
#include <hexdump.h>
#include <parsetools.h>
SoftwareSerial mySerial(10,11); // RX, TX
int shiftkeypressed=0;
int overwrite=0;
int isitdwm=0;

class KbdRptParser : public KeyboardReportParser
{
      void PrintKey(uint8_t mod, uint8_t key);

protected:
        virtual void OnKeyDown      (uint8_t mod, uint8_t key);
        virtual void OnKeyUp        (uint8_t mod, uint8_t key);
        virtual void OnKeyPressed(uint8_t key);
};
```

```
void KbdRptParser::PrintKey(uint8_t m, uint8_t key)
{
   MODIFIERKEYS mod;
   *((uint8_t*)&mod) = m;
   Serial.print((mod.bmLeftCtrl   == 1) ? "C" : " ");
   Serial.print((mod.bmLeftShift  == 1) ? "S" : " ");
   Serial.print((mod.bmLeftAlt    == 1) ? "A" : " ");
   Serial.print((mod.bmLeftGUI    == 1) ? "G" : " ");

   Serial.print(" >");
   Serial.print(key);
   if(isitdwn==1)
   {
   if(key==42)
   overwrite=1;
   if(key==40)
   overwrite=-17;
   isitdwn=0;
   }
   Serial.print("< ");

   Serial.print((mod.bmRightCtrl   == 1) ? "C" : " ");
   Serial.print((mod.bmRightShift  == 1) ? "S" : " ");
   Serial.print((mod.bmRightAlt    == 1) ? "A" : " ");
   Serial.println((mod.bmRightGUI    == 1) ? "G" : " ");

   if((mod.bmRightShift ==1)||(mod.bmLeftShift==1))
   {
      shiftkeypressed=1;
   }
};

void KbdRptParser::OnKeyDown(uint8_t mod, uint8_t key)
{
   Serial.print("DN ");
   isitdwn=1
   PrintKey(mod, key);
   uint8_t c = OemToAscii(mod, key);
   if((c>96)&&(c<123))
   {
    if(shiftkeypressed==1)
    c=c-32;
    shiftkeypressed=0;
   }

   c=c+overwrite;
   overwrite=0;
   Serial.println(" Overwrite value is");
   Serial.print(overwrite);
```

```
    if (c)
        OnKeyPressed(c);
}

void KbdRptParser::OnKeyUp(uint8_t mod, uint8_t key)
{
    Serial.print("UP ");
    isitdwn=0;
    PrintKey(mod, key);
}

void KbdRptParser::OnKeyPressed(uint8_t key)
{
    Serial.print("ASCII: ");
    mySerial.println((char)key);
};

USB     Usb;
//USBHub     Hub(&Usb);
HIDBoot<HID_PROTOCOL_KEYBOARD>    Keyboard(&Usb);

uint32_t next_time;

KbdRptParser Prs;

void setup()
{
    Serial.begin( 115200 );
    Serial.println("Start");
    mySerial.begin(9600);
    if (Usb.Init() == -1)
        Serial.println("OSC did not start.");

    delay( 200 );

    next_time = millis() + 5000;

    Keyboard.SetReportParser(0, (HIDReportParser*)&Prs);
}

void loop()
{
    Usb.Task();
}
```

Utility

Android smart phone are becoming more common by the day. We chat, send messages, send emails and much more. But most people find the soft keyboard inconvenient.

Our project is a solution.

Future Scope

We can generalize this for every application on android phone.
We can also make mouse for android as mouse can also work on boot protocol.

Useful links

http://bellcode.wordpress.com/2012/01/02/android-and-arduino-bluetooth-communication/
www.developer.amdroid.com.

A word of thanks

We would like to thanks the coordinators Nikhil, Rudra and Anurag also our mentors Nitish and Kapil for their valuable input in our project. We would specially like to thank Nehchal for his help in our project.

Also thank our peers for helping us along the way. Also like to thank the institute and Electronics club for giving us the chance to do this project. We would also like to thank our parents for their valuable support.