Implementing USB Host mode on the Android Platform

Andrew Pyles
Gang Zhou

May 10, 2010
1 INTRODUCTION

Smart phones are becoming very popular. It is estimated that by 2012, there will be more smart phones than desktops. Over the past several years we have seen the launch of the Apple iPhone and shortly after that Android has become popular. According to [?] Android is gaining popularity and will soon have a large percentage of market share.

One piece of functionality that is missing from both of the iPhone and Android platform is the ability to plug in USB devices to the device. Since USB 2.0 requires a 500mA 5V power supply to each USB port, it is clear that in order to provide this functionality, that it will require an external power supply. Since smart phones have to rely on a battery pack for its power, the power management on the phones needs to be quite agressive.

The project is about implementing USB Host mode on a typical Android device. Specifically the HTC ADP1, otherwise known as the google developer phone. The project can be broken down into 4 categories. Hardware support is required in lieu of a special cable. Second, a Kernel driver is required for to provide USB Host mode. Third, a userland Finally, a graphical user level interface is provided.

2 Motivation

One of the challenges with Wireless sensor networking is to capture data and send it securely over the internet to a remote data sink. Traditionally this is done using expensive router equipment. In stead, why not use an ordinary cell phone for this task? Cellular networks already have an impressive data network in the US and it is growing internationally. Why not leverage this network to transmit important data?

Suppose an important health monitoring network needs to send sensitive patient data over the internet? Although this problem presents privacy concerns outside the scope of this document, it is very important problem. Using a cell phone over existing cellular networks can provide a key advantage.

In both of these examples, using a smart phone to transmit this data has possible only on a limited basis since the ability to plug in external USB devices is not possible. Bluetooth support is prevalent but a lot of medical devices are still using USB. USB is arguably more secure as well since the data does not need to be sent wirelessly where a bluetooth scanner could potentially intercept important data.

The goal of this paper is to enable any external USB device to plug into an Android phone. Since Android uses the Linux kernel, we can take advantage of the vast array of USB devices already available for Linux.
However, the focus of this paper is to enable Motes for wireless sensor networks. Most of these devices have a USB interface that utilizes a USB to serial port driver. From the operating systems perspective, this appears a bi-directional serial device. We then develop an application that can communicate to the mote and be able to read/write to this device.

3 Challenges

The ADP1 Android phone comes with a MSM 7201 USB HOST controller. Unfortunately due to restrictions documentation for it is not readily available. Thus the kernel driver requires some work to be able to be implemented correctly. Additionally, since the USB hardware does not generate any power an external power supply is required.

Since 2.6 the Linux kernel has supported UDEV, a userland device manager that listens for events from the kernel. We discuss this in more detail in section 4.4. Most Linux distributions include this functionality. However for the Android platform, UDEV is not included. A way to implement a subset of UDEV is required to be implemented in order to support the creation of new devices. Although a primitive device manager is included, some changes were required.

4 Implementation

4.1 Kernel support

In order to prepare the phone to support external USB devices, a kernel EHCI or host controller is required. We incorporated the suggestions from [?] to modify the Freescale MPC5121 driver to make it work with the Qualcomm MSM7201A chipset on the Android phone. With these modification the host controller is able to recognize USB devices with a caveat: enabling the HOST controller disables the USB client mode.

```c
static struct platform_driver ehci_msm7201_driver = {
    .probe = usb_hcd_msm7201_probe,
    .remove = usb_hcd_msm7201_remove,
    .shutdown = usb_hcd_platform_shutdown,
    .driver = {
        .name = "msm_hsusb",
    },
};
```
We briefly describe the functionality of the device driver. The EHCI host controller driver is placed in "drivers/usb/host" directory of the Linux kernel. The source file is in "ehci-msm7201.c". The skeleton of the driver is shown above. Specifically, the driver has to have the following functions identified:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>probe</td>
<td>Device specific initialization routine</td>
</tr>
<tr>
<td>remove</td>
<td>Device specific shutdown processing</td>
</tr>
<tr>
<td>shutdown</td>
<td>Generic hcd shutdown routine</td>
</tr>
<tr>
<td>driver</td>
<td>Name of the driver</td>
</tr>
</tbody>
</table>

When the driver is initialized, the probe() function is called. This function initializes the host controller on the 7201 chip. While an indepth discussion of the Linux USB host controller subsystem is beyond the scope of this document, suffice it to say that once the hardware is initialized, the driver calls generic HCD functions common to all host controllers. For instance, as shown here:

```c
hcd = usb_create_hcd(driver, &pdev->dev, dev_name(&pdev->dev));
if (!hcd) {
    retval = -ENOMEM;
    goto err1;
}
```

Since the hardware does not appear to provide power to external USB devices, we did not attempt to enable support for it in the driver. In the next section we describe how we create a custom cable to power devices.

### 4.2 Hardware Support

After the kernel was up and running, we needed to hook up a device to the phone. This required a special USB cable. The cable that we used was a USB 2.0 A to Mini-B Extension cable as seen in figure one. The phone has a special HTC extUSB female adaptor built into the phone which includes mini-USB capabilities plus an additional pin for an audio jack. Additionally the phone can be charged through this adaptor with the USB cable provided with the phone. Since the
extUSB and the Mini-B adaptor are compatible, however, we were able to plug the extension cable directly into the phone.

The USB 2.0 specification requires 5V .5A peak load [?]. In order to generate the necessary power we modified the extension cable in the following way. First the positive and ground cables were cut on the phone side and a power supply was attached so that only the external USB device side could receive power. This way the battery is only used when the USB device is plugged in. This provides an added benefit that it does not use any of the phones battery.

The power supply that we used was a Bodhilabs [?] VPack5.0V-AA-2 device. This unit takes two AA batteries and uses a step up transformer to produce 5V with a peak load of .5A. Since our target device as described in section 4.3 will generate a much smaller load than that, this supply should be more than sufficient.

![Hardware enclosure](image)

**Figure 1: Hardware enclosure**

As can be seen in figure one, we show how the hardware enclosure works. For ease of use, we have attached a 7 port USB hub. The black cable coming out of the enclosure is attached to the phone. Devices are plugged into the HUB and powered from the external power supply.
Figure 2: Detail of how the power supply is connected to the extension cable. A tmote is connected to the USB hub.
4.3 TinyOS

In order to have TinyOS support there were two challenges: Systems implementation and TinyOS software modifications. We had to make some changes to the system to support external devices. Additionally, some software changes are required for TinyOS to be compatible with the Android platform.

Our department has a number of MOTE devices from Crossbow technologies [?], for example, the Tmote and Iris. These devices include an FTDI USB to serial chip which is already supported by Linux. The Linux driver creates an entry in the /dev/ directory and passes the information to the device manager. We ensure that the device was created with sufficient privileges by modifying the Android device manager. The details are described in the next section.

Finally, after the system implementation, we focus on the software changes required for TinyOS to work with the Android platform. TinyOS uses binary serialization to communicate bidirectionally from the mote to the host. This is done with the help of a C++ JNI interface. The C++ JNI library would require lots of changes to port to Android (Android does not include the C++ stdlib). Instead, we wrote a new Java TinyOS library that sends Ascii over the serial interface without any translation necessary. By default the application sends an unsigned 8bit integer at a time through the FTDI UART. TinyOS also uses an encoding similar to HDLC [?]. By removing the encoding, and printing ascii directly to UART, we were able to read the output on the phone, and to minimize the amount of work needed on the phone side to decode the message. This allowed us to implement a much simpler JNI interface to our application. In future work, we plan to re-implement the HDLC encoding as this will extend the functionality of the interface so that existing TinyOS code can be used without any modifications.

4.4 Device Management

Linux as of 2.6 and onwards uses udev, a user level process, for managing device files in the /dev directory. Android does something very similar with init. When a new device is added to the system, init creates a device with 700 level permissions owned by root. The Android security policy, however, ensures that every application has its own UID. Therefore we had to make a modification to ensure that correct permissions were created for the MOTE serial device.

Before we discuss the device management portion of the implementation, we pause here to discuss briefly the layout of the Android system.

In order implement this functionality we had to modify the Android source code. The Android source code Is comprised of several factors. As previously discussed, Android is based upon the Linux 2.6.2x kernel. Since Android is built on
top of Linux, it is similar to UNIX system. In all UNIX systems, there is a process with a process id (pid) of one. This process is called “init” and is responsible for spawning all other user processes on the system. As previously mentioned, Android has a primitive device manager implementation that is included in init.

The Android device manager listens on a netlink socket for uevent file descriptors [?] . When a new device is registered to the kernel, the kernel sends information about that new device on the netlink socket. The Android device manager then handles that uevent and creates appropriate devices. Since Android was not designed to handle multiple devices, we added simple code that will work for USB serial devices. When a new uevent file descriptor is received that contains the name "/dev/ttyUSB" we ensure that the permissions are set correctly. As we can see, we made a change as seen below that will ensure that all users of the system can read/write to the system. This needed to be added because by default all devices were created with rw access only by the root user.

```c
if(! strcmp (path, "/dev/ttyUSB" )) {
    chmod (path, S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH);
}
```

### 4.5 Application

After initial connectivity was completed, we were able to retrieve packets over the serial interface. TinyOS relies upon a C++ JNI interface to Java to translate the TinyOS binary protocol to a human readable format. Porting the C++ JNI interface to Android proved to be a bit of a challenge. The C++ code includes calls to the C++ std library which is not included in Android. Although this may have been possible to port these libraries to the phone, we went in another direction.

We based our application on an opensource Android application. This application entitled Android-serialport-api was designed to take input from a serial device. It consists of a simple JNI interface that allowed us to read directly from the serial port. The application polls the serial device and outputs the results to the screen. The application allows the baudrate to be configured and it saves the settings in a configuration file.

One drawback to the application is that when the application is running, it will power on the display and use up additional battery power. Future work is required to see if we can receive input in a minimal battery state. When the device is fully awake, the battery is depleted quickly. We save this for future work.
For this iteration, minimal changes were made to this application. In future work we plan to expand this application to store received data from the MOTE and store in a SQLite database and periodically send a summary to a network server.

4.6 Miscellaneous

In this section we discuss other changes that do not fit into any particular category. Android provides a debug console for developers called Android developer bridge. This allows developers to have a login shell where files can be transferred to/from, as well as install applications. We had to disable this support for this project because it enabled USB client mode on the device. It is not possible to run both USB host mode and client mode at the same time. To get around this issue, we had to create two kernels. One with USB host mode, and the other a default or stock kernel. Each time we want to enter USB debug mode we have to reflash the boot partition with the default kernel. Similarly to get USB host mode support or custom kernel must be flashed. Although this is a drawback, it is manageable. The only time USB debug support is needed is for development purposes. In which case this is one extra step can be tolerated.

An alternative to this support is a loadable kernel module. This presents a challenge though, in that the Host controller support cannot be built as a kernel module. However, it can be built either as a gadget or function or function module. As of this writing there is some confusion in that gadget drivers are what are being used in the latest linux kernel [?], whereas the version that is used by Android favors function. We have omitted this option for this reason, also the benefit does not seem to meet the cost.

5 Conclusion

In enabling a USB host controller on the HTC Dream platform we believe that this can be beneficial to both researchers and consumers. To the best of our knowledge we are the first ones to enable a smart phone for use as a data aggregator with TinyOS. This will allow for further research to be done when a computer is not readily available. Most phones include an internet plan as part of the monthly fees. Thus using this device as long as coverage is available, data can then be uploaded to a remote server for further analysis.

Additionally this phone can now be useful for consumers as well. Virtually any USB device can be plugged into the phone as long as there is a linux driver available. One possibility would be to obtain a USB video adaptor. With some work, a video projector could be hooked up to the phone. Also a USB keyboard
could be plugged in as well.

5.1 Special thanks

Special thanks to Matt Keally for his help on the TinyOS work with the TinyOS library.