The User's Manual for HP TinyOS Software Development

Andrew Christian

andrew.christian@hp.com

The User’s Manual for HP TinyOS Software Development

by Andrew Christian

Revision History

Revision $Revision: 1.1 $ $Date: 2005/06/15 14:13:00 $ Revised by: $Author: andyc $

Table of Contents

Chapter 1. About this Document

This document describes the hardware and software architecture used by Hewlett-Packard's Cambridge Research Laboratory for our experiments in small computer platforms.

This document is very much work in progress. Please send Andrew your suggestions, comments, and updates.

The members of the CRL Metro Project Team are: Brian Avery, Steven Ayer, Bor-rong Chen, Andrew Christian, Don Denning, Ben Kuris, Jamey Hicks, Bodhi Priyantha

Chapter 2. Getting started quickly

The goal of this chapter is to get you up and running as quickly as possible. For our purposes, we will assume that you have already checked out a copy of the TinyOS source tree, have a Linux development machine running a 2.6 kernel, and have three Telos revision B motes available.

2.1. Setting up your environment

You'll need copies of the MSP430 tool chain, the bootstrap loader, and the TinyOS source tree.

Set up your environment variables in your `.bash_profile` or other appropriate location. My version looks like:

```
# Tiny OS settings
PATH=$PATH:/usr/local/msp430/bin:/usr/local/tinyos/bin
TINYOSDIR=/home/andyc/repository/tinyos-1.x
TOSDIR=$TINYOSDIR/tos
MAKERULES=$TINYOSDIR/tools/make/Makerules
```

2.1.1. Running the Sniffer

First, let's run an 802.15.4 scanner software. This can be found in `contrib/handhelds/apps/ZSniff`. Assuming that you have a Telos B mote attached to `/dev/ttyUSB0`, type:

```
make telosb install bsl,/dev/ttyUSB0
python zdump.py sniff -d /dev/ttyUSB0
```

By default, the **zdump** program uses 802.15.4 channel 11 and displays output on a single line. Here's some sample output:

```
9.53     BEACON beacon_order=15 superframe_order=0 final_cap_slot=0 PANCOORD fcf=8000 [BEAC
9.53     DATA_REQUEST fcf=8023 [COMMAND ACKREQUEST] dsn= 42 src=be:ef/00:01 rssi=18 lqi=106
9.53     ACK fcf=0012 [ACK FRAMEPEND] dsn= 42 rssi=-10 lqi=107 crc=True
9.55     BEACON beacon_order=15 superframe_order=0 final_cap_slot=0 PANCOORD fcf=8000 [BEAC
9.60     BEACON beacon_order=15 superframe_order=0 final_cap_slot=0 PANCOORD fcf=8000 [BEAC
9.61     BEACON_REQUEST fcf=0803 [COMMAND] dsn=247 dest=ff:ff/ff:ff rssi=-44 lqi=105 crc=Tr
9.61     BEACON beacon_order=15 superframe_order=0 final_cap_slot=0 PANCOORD ASSOC_OK fcf=8
```

There are options to the **zdump** program to decode TCP/IP traffic, display multiple lines per packet, set and hop channels, and filter packets by type. (* Add link to zdump reference here *)

2.1.2. Setting up an Access Point

The HP TinyOS tree provides clients that can communicate using TCP/UDP over IP over 802.15.4. An access point is needed to connect the wireless 802.15.4 client back to the wired Ethernet. We have a hardware project to build a small embedded Linux computer with wired Ethernet and 802.15.4 radios (* add link here *). For people without the hardware, the `contrib/handhelds/apps/AccessPoint` software configures a Telos Mote to act as basic access point. Wired Ethernet traffic from the Mote is sent over the USB cable to a host PC running a custom Linux kernel module.

We'll assume that you have a Linux development machine running kernel version 2.6 and a Telos Mote connected to `/dev/ttyUSB1`. The first step is to build and install the kernel module that routes IP traffic:

```
cd contrib/handhelds/apps/AccessPoint/kernel
make
insmod ./telos_ap.ko
```

Now we install the access point software on the Mote.

```
cd contrib/handhelds/apps/AccessPoint
make telos IP=10.0.1.1 LONG_ADDRESS=1 install bsl,/dev/ttyUSB1
```

A few things to note here: First, you had to specify an IP address for the access point. The Telos Mote will be acting as a network interface card for your Linux computer, and as such, it needs to have a network address. Some day we will try to remove this restriction so you can assign the address using **ifconfig** or equivalent. Second, you had to specify a LONG_ADDRESS. This two byte value is used to create a unique network MAC address for the CC2420 radio. The LONG_ADDRESS is not needed for devices that have a DS2411 identification chip (such as the Telos revision B or TMote Sky motes).

The third step is to running the access point user-space daemon. The daemon sets up the kernel module and updates routing tables.

```
cd contrib/handhelds/apps/AccessPoint/daemon
make
zattach -n -v /dev/ttyUSB1
```

If the daemon is running correctly, you should see something like:

```
Device name telos0

Event: Reset
IP:    10.0.1.1
Addr:  a0:a0:00:00:00:00:00:01
PanID: 0xbeef
Freq:  2405 (channel 11)
SSID:  CRL-Medical
setting host ip address
```

The **zattach** program handles updating the routing tables as clients join and leave the access point. However, you won't be able to connect to those clients outside of your local Linux machine unless you turn on IP forwarding:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
```

Remember, if you are going to forward packets off of your local computer, you need to supply routable IP addresses.

Notice that the output of the **zattach** program contains the device name `telos0`. This is an actual Linux network device. Try **ifconfig** to see its settings or running **ethereal** and watching the network traffic on the device.

2.1.3. Running a Simple Telnet Client

Now we'll install and test a simple Telnet client that runs over 802.15.4. We'll assume that you have a Telos revision B mote plugged into `/dev/ttyUSB2`.

```
cd contrib/handhelds/swtest/TelnetServer
make telosb IP=10.0.1.2 install bsl,/dev/ttyUSB2
```

As with the access point, if you are running a client device that doesn't have a DS2411 identification chip, you must specify a unique LONG_ADDRESS value.

The client will scan all of the local 802.15.4 channels, locate available access points (by sending `BEACON REQUEST` packets), select the best access point, and associate with it. If everything goes well, you should see the output of the **zattach** program displaying:

```
Event: Associate
IP:    0.0.0.0
Addr:  a0:a0:00:00:0c:59:f8:53
SAddr: 0
Flags: 0x81

Event: ARP
IP:    10.0.1.2
Addr:  a0:a0:00:00:0c:59:f8:53
SAddr: 0
Flags: 0x81
adding client route
```

Now you should be able to run **ping 10.0.1.2** and **telnet**. You can watch packets fly back and forth both from the **zdump** software that shows you what is going over the air and from **tcpdump** or **ethereal** running on the `telos0` interface.

2.2. Running with Wires

The problem with debugging network code is that the wireless mode of operation takes a long time and requires at least two motes (one as an access point, one as a client). This can be avoided by running the mote in a "wired" mode.

From the perspective of the Linux host computer, the traffic across the USB cable contains two types of traffic:

- IP packets
- Status packets showing clients associating and de-associating

To simply our debugging, the client mote can be compiled to route IP traffic directly over the USB cable, rather than over the 802.15.4 radio. This "wired" mode also generates a set of fake status packets for the Linux host computer so that it appears to be just an ordinary access point. To compile the Telnet server sample code for wired mode, type:

```
cd contrib/handhelds/swtest/TelnetServer
make telosb IP=10.0.1.4 HOST_IP=10.0.1.5 install bsl,/dev/ttyUSB2
cd ../../apps/AccessPoint/daemon
zattach -n -v /dev/ttyUSB2
```

The HOST_IP variable assigns an IP address to the faux access point. It also serves as a compile-time flag that pulls in the `WiredIPClientC.nc` code instead of the `WirelessIPClientC.nc` code.

Chapter 3. Utilities

A number of Python utilites are available in `tools/scripts`. This section contains a description of each utility.

3.1. Memory usage

The `memuse.py` script parses compiled MSP430 programs and extracts information about memory usage including detailed statistics for the data and text section use for NC module. What it actually does is to use the `msp430-nm` program to extract the symbol table and parses each entry for space usage. The NESCC tools create symbol table entries of the form `Module$Element`. The `memuse.py` script chops the symbol name and uses this to compile statistics.

Here is a sample run of `memuse.py`:

```
cd contrib/handhelds/swtest/WebServer
memuse.py build/telosb/main.exe
```

which generates the following output:

```
Module                   Data    Text
other                      73   18038
UIP_M                     312    4548
ClientM                    42    3288
HTTPServerM               446    2952
MSP430ADC12M               25    2002
CC2420HighLevelM           35    1788
CC2420RxM                  60    1456
TimerM                     77    1114
IEEEUtilityM                0     796
CC2420LowLevelM             0     662
HPLUSART0M                  4     546
WebServerM                430     480
RefVoltM                    3     476
ADCM                        7     452
DS2411LiteM                 0     372
LinkLayerM                  0     352
TimerJiffyAsyncM            5     222
MessagePoolM              803     210
MSP430ClockM                0     164
MSP430DCOCalibM             2     100
RandomLFSR                  6      72
LedsC                       1      52
HPLADC12M                   0      46
```

```
InternalVoltageM            1      18
InternalTempM               1      18

Summary:
          Data  1100-111c (28 bytes)
           BSS  111c-1a01 (2277 bytes)
         Stack  1a01-3900 (7935 bytes)
   Above stack  3900-4000 (1792 bytes)
   Text (code)  4000-e172 (41330 bytes)
     Data init  e172-e18e (28 bytes)
     Free Text  e18e-ffe0 (7762 bytes)
```