

# **Implementation of Linux device driver that communicates with motes**

**2004 summer**

**CS 558L Final Projects**

**USC**

Kelvin Chung  
[tatchung@usc.edu](mailto:tatchung@usc.edu)

Cassidy Yeung  
[yuiyeung@usc.edu](mailto:yuiyeung@usc.edu)

## 1.0 Introduction

This project implements a simple Linux device driver that communicates with Motes in various ways. Traditionally, user programs communicate with a mote by directly opening the serial port. This involves a detailed understanding of how Motes work. In this project, a Linux device driver for Motes is implemented to free programmers from understanding the intricacies of Motes. The project is separated into two parts: Linux device driver and Mote device driver. Besides, we developed a Mote Java class library that allows users to access the Linux driver functions easily via the Java Native Interface.

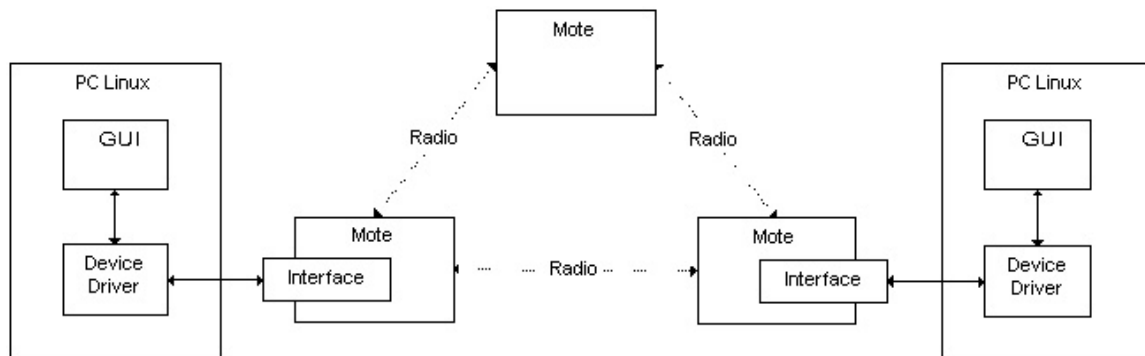
## 2.0 Design

The design of this project is to implement the following scenario. Two or more Linux PCs, each connected to its Mote via either Serial port or USB port, will communicate with each other using read/write communication file descriptors for the Mote driver. There are one or more Motes within the communication range, which are either attached to a sensor board or not. The software is architected into 4 layers, each one built on top of the other:

1. NesC Mote driver that handles communication between Mote/Mote and Mote/PC
2. Linux Mote kernel module that provides an API to communicate with the Mote driver
3. MoteDevice Java class library and JNI native interface that provide an easy-to-use interface to Java application programmers.
4. Java GUI application to demonstrate all the functions and display the results graphically.

The functionality provided by our Mote implementation are:

1. Loop the packets back from Motes to the PC to test if a Mote is working or not
2. Send messages to remote Motes, either broadcast or specific address
3. Get and set byte count for the specific Mote
4. Get and set the LED light on all the Motes (in broadcast mode) or a specific Mote
5. Get the sensor values from specific sensors (either attached to PC or via radio). Current supported sensor components are light intensity and temperature.
6. Get the address that is directly connected to the PC.



### **3.0 Mote NesC Driver Implementation**

There are basically two types of communication in motes: with the PC and with the motes. The mote and PC communicate through the serial port. The motes and motes communicate through the radio channel. A communication channel is built using module provided by the tinyOS.

#### **3.1 Communication between PC and mote**

The component UARTComm is renamed as PCComm. It is used to communicate with serial port. PCCommControl is the instance of StdControl. PCCommControl is wired to UARTComm. PCSndDataMsg is the instance of SendMsg. PCRcvDataMsg is the instance of ReceiveMsg. They are wired to UARTComm.sendMsg[DRV\_SNDMSG] and UARTComm.receiveMsg[DRV\_RCVMSG] respectively. DRV\_SNDMSG and DRV\_RCVMSG should be the same. However, The communication between PC and mote ignore the use of the port DRV\_SNDMSG and DRV\_RCVMSG.

Whenever the message is receive from the PC, the event PCRcvDataMsg.receive() will be triggered. It will then parse the message and perform the corresponding task depend on the message type. The content of the pointer will be stored temporary to different type of TOSMsg message and send out according to the requirement of the task. For sending data to PC, PCSndDataMsg.send() will be used. Atomic keyword is used to make sure the operation is not interrupted during the send and receive process.

#### **3.2 Communication between motes**

The component GenericComm is renamed as MoteComm. It is used to communicate through the radio channel. MoteCommControl is the instance of StdControl. MoteCommControl is wired to GenericComm. Similar to the communication between PC and mote, MoteSndDataMsg is the instance of SendMsg. MoteRcvDataMsg is the instance of ReceiveMsg. They are wired to GenericComm.sendMsg[MOT\_SNDMSG] and GenericComm.receiveMsg[MOT\_RCVMSG] respectively. The handler ID of the send message (MOT\_SNDMSG) and receive message (MOT\_RCVMSG) should be the same. It should be the same in the corresponding send and receive message. This is especially important in radio transmission. Otherwise, the messages will be lost.

Whenever the message is receive from other motes, the event MoteRcvDataMsg.receive() will be triggered. It will then parse the message and perform the corresponding task depend on the message type. The content of the pointer will be stored temporary to different type of TOSMsg message and send out according to the requirement of the task. For sending data to PC, MoteSndDataMsg.send() will be used. Atomic keyword is used to make sure the operation is not interrupted during the send and receive process.

#### **3.3 Messages involved in the communication**

14 messages are used. They are loop message, communication message, get count request message, get count reply message, set count message, get led request message, get led reply message, set led message, photo request message, photo reply message, temperature request message, temperature reply message, get address request message and get address reply message. They are all defined in the header file <MoteDriver.h>.

##### **Loop the packet back**

The message is received from the PC and the mote will send the message back to PC. This is to make sure the mote is working and the connection between PC and the mote is of no problem. When the message is receive from the PC. It will be copy to TOSMsg LoopMsg and send back to PC. Function used to perform this is pCLoopBackTask.

#### *Loop message*

0	LOOP_MSG	1 byte
1-<28	Message involved	<28 bytes

#### **Send the data**

The data message is sent from the PC and tries to reach the mote address specific in byte 1-2. If the mote destination is the same as the directly connected mote, the message will not broadcast. Otherwise, it will broadcast to all other motes and if the message is the same as the broadcast mote, it will send the data to the PC.

#### *Communication message*

0	COMM_MSG	1 byte
1-2	Address of the receiving mote	2 bytes
3-<28	Message involved	<26 bytes

#### **Get and set byte count of motes**

Each mote is responsible in measuring the number of byte. The number will be stored and calculated locally at each mote as pcSendCount, pcRecvCount, moteSendCount and mote RecvCount. These values store the number of byte sent to and received from the PC through the serial port, and that sent to and received from the radio respectively. It is calculated inside the method. The sendCount are incremented at PCCommDataMsg.SendDone() and MoteCommDataMsg.sendDone(). The recvCount is incremented at PCCommDataMsg.receive() and MoteCommDataMsg.receive().

Whenever the get count request message is received, the motes retrieve the byte count information and sent it out using the get count reply message. The set count message resets the count byte information at the mote.

#### *Get count request message*

0	GET_COUNT_REQ_MSG	1 byte
1-2	Address of the destination mote	2 bytes

#### *Get count reply message*

0	GET_COUNT_REP_MSG	1 byte
1-2	Address of the destination mote	2 bytes
3-4	PC received count (from serial/USB port)	2 bytes
5-6	PC sent count (from serial/USB port)	2 bytes
7-8	Mote received count (from radio)	2 bytes
9-10	Mote received count (from radio)	2 bytes

#### *Set count message*

0	SET_COUNT_MSG	1 byte
1-2	Address of the destination mote	2 bytes
3-4	PC received set value (serial/USB port)	2 bytes
5-6	PC sent set value (serial/USB port)	2 bytes
7-8	Mote received set value (radio)	2 bytes
9-10	Mote received set value (radio)	2 bytes

#### **Get and set Led light**

The led information is one byte. It will get directly from the motes whenever it is called. The mechanism is basically the same as the set and get count message except that the information is set and get in runtime by calling Leds.set(value) and Leds.get().

#### *Get led request message*

0	GET_LED_REQ_MSG	1 byte
1-2	Address of the destination mote	2 bytes

#### *Get led reply message*

0	GET_LED_REP_MSG	1 byte
---	-----------------	--------

1-2	Address of the destination mote	2 bytes
3	Led value of the mote	1 bytes

*Set led message*

0	SET_LED_MSG	1 byte
1-2	Address of the destination mote	2 bytes
3-4	PC received set value (serial port)	2 bytes

**Get the environment value**

The environment values are retrieved using the ADC and ADCControl module. It will first call ADC.getData() and that method will call back the function ADC.dataReady(value) and get value we want. When we trigger the ADC.getData method, we need to wait for the dataReady method to be called in order to get the value. The mechanism for getting the photo information and the temperature information are the same except that the ADC and ADCControl wired to different component Photo and Temp.

Timers are used to trigger to get the value from the sensorboard. This is different from the method used in getting the led light value. Since there may have a considerable response time in the sensorboard and the value is not immediate available when it is called. This may lengthen the time for getting the information. To avoid that, the information is retrieved periodically using the timer and store in the mote. It will be stored as photoVal and tempVal for light intensity and temperature.

Whenever the get photo or get temperature request message is received, we will put the photoVal or tempVal in the correct field and send it out.

Message for requesting and replying the photo information

*Photo request message*

0	PHOTO_REQ_MSG	1 byte
1-2	Address of the destination mote	2 bytes

*Photo reply message sent to the PC*

0	PHOTO_REP_MSG	1 byte
1-2	Address of the destination mote	2 bytes
3-4	Photo value from the mote	2 bytes

Message for requesting and replying the temperature information

*Temperature request message get from PC*

0	TEMP_REQ_MSG	1 byte
1-2	Address of the destination mote	2 bytes

*Temperature reply message sent to the PC*

0	TEMP_REP_MSG	1 byte
1-2	Address of the destination mote	2 bytes
3-4	Temperature value from the mote	2 bytes

**Get directly connected mote address**

It will be more convenience if the address of the directly connected mote is known when the device is connected. This is for the sake of easy testing. The mechanism used is the same as the loop back function except that the address is put into the field instead of the original message.

*Get address request message*

0	GET_ADDR_REQ_MSG	1 byte
---	------------------	--------

*Get address reply message*

0	GET_ADDR_REP_MSG	1 byte
1-2	Address of the directly connected mote	2 bytes

### Request and reply pair

After the message is sent from PC to the first mote, the motes will insert the source address into the message. This is to mark the source address since the PC kernel will only accept the request if it is asked for a specific request. Therefore, in order to let the mote identify itself as the mote directly connected to the requested PC. It will insert its address in the packet as following.

Request message get from PC

0	GENERIC_REQ_MSG	1 byte
1-2	Address of the destination mote	2 bytes

Request message pass to other motes

0	GENERIC_REQ_MSG	1 byte
1-2	Address of the destination mote	2 bytes
3-4	Address of the source mote	2 bytes

Reply message sent among motes

0	GENERIC_REP_MSG	1 byte
1-2	Address of the source mote	2 bytes
3-4	Address of the destination mote	2 bytes
5-?	Get value from the mote	? bytes

Reply message sent to the PC

0	GENERIC_REP_MSG	1 byte
1-2	Address of the destination mote	2 bytes
3-?	Get value from the mote	? bytes

The mote will only reply to PC if it is found its address is in the field Address of the source mote in the reply message.

## 4.0 Mote Kernel Module Implementation

The kernel module is built using the latest Linux 2.6 kernel API on Fedora Core 2. In order to support USB interface which may have different kind of USB to serial converter, we use the *filp* kernel API to directly open device files */dev/usb/ttyUSB0* etc. and communicate with it. Similar we use */dev/ttyS0* etc. for Mote serial port connection. In this way there is no need to write different driver using *usb\_open()* kernel API for different USB converter (the one we are using is Keyspan). This also simplify module development by using the *read/write/open/close/poll* interface of *filp* kernel API directly. Currently we support Mote attached to USB 0...USB 3, Serial 0 and Serial 1 - most PC should have one slot available. Besides, the kernel module also support up to 6 Motes connecting to the same PC using the above interface. This can be useful for testing Radio communicate if only 1 PC is available.

### 4.1 Mote Device files

There are 7 types of device files, each type support 6 interface (4 USB and 2 Serial) with a total of 42 mote device drivers under */dev*. The naming convention is

*/dev/moteTYPE\_INTERFACE* where *TYPE = Addr, Loop, Comm, Led, Count, Photo, Temp*  
*INTERFACE = USB0, USB1, USB2, USB3, S0, S1*

#### 4.1.1 Device moteAddr

This read only device return 2 bytes that represent an unsigned short integer address of Mote e.g. *fd = open("/dev/moteAddr\_USB0", O\_RDONLY)*. The first and second byte return from read represents lower and higher order byte of integer respectively. This is true for all integer return from Mote discuss below. i.e.  $integer = (byte[0] \& 0xff) | ((byte[1] \ll 8) \& 0xff00)$ .

### 4.1.2 Device moteLoop

This device test if mote is functional. Anything written to the device should read back in next call unchanged. The length of of byte written should not greater than *TOSH\_DATA\_LENGTH* (26) defined in *mote.h* or else *write()* return -1.

### 4.1.3 Device moteComm

This device send a stream of bytes written (broken into different packet if greater than packet length 26) to it using file *write()*. This is send through radio to either a specific Mote address (using *ioctl()* discuss later) or broadcast to all Motes. The other side should issue *read()* on *moteComm*, otherwise packet will lost. This can be two way communication provided that user program synchronized read and write properly.

### 4.1.4 Device moteLed

This device get/set the 3 LED lights of target mote by sending 1 byte to it. If broadcast address is used all LED within the communication range will be set. The lower order 3 bits represent red, green, yellow respectively. The light is on if bit is one. The other 5 bits are ignored. If there is more than 1 byte written, each of them will write one byte at a time. Similarly *read()* will always return one byte at a time and represent the current LED light.

### 4.1.5 Device moteCount

This device get/set 4 integers of mote communication usage count of target mode address using 8 bytes. The usage count represent number of bytes send so far since last time counter set to other values. The order of usage counts are (i) PC to Mote (ii) Mote to PC (iii) Radio to Mote (iv) Mote to Radio. *Read()* will always return 8 bytes and *write()* will return -1 if length is not 8 bytes.

### 4.1.6 Device MotePhoto

This read only device get an integer (2 bytes) represents light intensity of sensor board that attached to target Mote specific by address. If the target mote doesn't have sensor board it will return zero light intensity.

### 4.1.7 Device MoteTemp

This read only device is similar to *MotePhoto* except that an integer (2 bytes) represents temperature value is return.

## 4.2 Mote Device Attributes

Two types of device attributes can be get/set via *ioctl()* system call to control device read/write. They are describes below.

### 4.2.1 Baud Rate

To support different kind of Mote (not necessary mica which we test), the API allow setting different device baud rate using:

```
ioctl(fd, MOTE_SET_BAUDRATE, rate);  
ioctl(fd, MOTE_GET_BAUDRATE, &rate);
```

where rate is an unsigned long. Currently support rates are 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 500000, 576000, 921600, 1000000, 1152000, 1500000, 2000000, 2500000, 3000000, 3500000, 4000000. If rate is not support by kernel (even though they list above) or not one of above *-EINVAL* is return. By default rate = 19200 which should work with mica.

## 4.2.2 Destination Address

The devices *moteComm*, *moteLed*, *moteCount*, *motePhoto*, *moteTemp* allow to perform specific functions on mote other than one attach to PC using:

```
ioctl(fd, MOTE_SET_DESTADDR, addr);
ioctl(fd, MOTE_GET_DESTADDR, &addr);
```

where address is unsigned long and the lower 2 bytes is used.

By default broadcast address is used *TOS\_BCAST\_ADDR* (*0xffff*) defined in *mote.h* that operation is performed on current Mote attach to PC (except *moteCOMM*). The call return *-EINVAL* if address set is the reserved value *TOS\_UART\_ADDR* (*0x007e*).

## 4.3 Implementation

### 4.3.1 Preparation

The implementation is divided into 2 phase. First a simple user space program is written that open USB or Serial connection with Mote, send a packet according to the generic frame format, compute the checksum and receive it back. There is no such C program available in the tinyOS distribution. The closest on is *raw\_listen.c* under *tinys/tinys-1.x/tools/src* which dump the raw frame receive from mote. By examining the packet dump, studying the document and looking into Java implementation *tinys/tinys-1.x/tools/java/net/tinys/packet/Packetizer.java* we know how the packet is arrange and which bytes need to escape. The documentation, however, does not mention that there is an extra byte at the beginning of header describe packet type. We also need to carefully set terminal IO attributes. After that a small nesC Mote program is written and install in mote. We are able to verify that loop back run correctly using random packet from length 0 to 29. In the second phase, the actual kernel module is written.

### 4.3.2 Device Major/Minor Number

The device major number is assigned by *register\_chrdev(0, "mote", &mote\_fops)* which return an unused device number. The script *mote.init* (modify from sample init script for the a driver module by [rubini@linux.it](mailto:rubini@linux.it)) will (i) load the module *mote.ko* (ii) get the mote device major number from */proc/devices* (iii) create all 42 */dev/mote\** files using the major number and (iv) set correct file mode for both */dev/mote\**, serial and USB device when start. During shutdown the script *mote.init* will take care of removing all */dev/mote\** entry and unload the mote module.

The device minor number is assignment is shown in the header of same script. The higher order 4 bits of minor device number identifies which of 6 interface that Mote connect and lower order 4 bits identifies the 7 Mote functions.

### 4.3.3 Device File Operation

The *mote\_fops* table assigned by device support the following operations:

```
struct file_operations mote_fops = {
    open:    mote_open,
    release: mote_release,
    read:    mote_read,
    write:   mote_write,
    ioctl:   mote_ioctl,
    poll:    mote_poll,
    owner:   THIS_MODULE,
};
```

For all operations it check the correct function and port from device minor number. If invalid *-ENODEV* is returned . The correct read/write access mode is also verified for individual mode function. If invalid -

EACCESS is return. For open -EBUSY is returned if port interface already open and -ENOMEM if module cannot allocate the following private data structure. This is a pointer to an array of pointer to MoteDevStruct, one for each port interface. The last port interface release will free the private data structure.

```
struct MoteDevStruct {
    struct file *mote_fd; // internal port interface file descriptor
    int readEscaped; // Is the last byte read contain Escape sequence ?
    int readFrameOffset; // Position of last byte from current frame return to user read()
    int readFrameEnd; // Position of current frame end. If reach another frame need to fetch.
    int writeFrameOffset; // Position of last byte from current frame written successfully.
    int writeFrameEnd; // Position of current frame end. If reach write another frame from user.
    int writeFrameLen; // Length of current frame user pass in to be written (without header)
    int triggerRead; // How many byte to read before another trigger send
    unsigned char mote_wbuffer[MTU];
    unsigned char mote_rbuffer[MTU];
    struct semaphore sem; // protected read/write
    int baudRate; // current baud rate
    int dest; // address to send
};
```

For *moteAddr*, *moteLed*, *moteCount*, *motePhoto*, *moteTemp* read operation, the module will first write a trigger packet to Mote device before it returns the result. The format of packet is describe in section 3. In this case the correct implementation for block reading in user space is to have the first read() call before invoking select() to wait for read. For *moteComm* the device module will take care of broken a stream of byte into smaller packet and send to the other side. It will also remember which position of byte in frame have been read/written internally using the above private data structure.

Our interface for read/write follows exact semantic of port interface device driver. So if the driver does not implement block read(), Mote device read() will also do the same thing. It is however preferable for user application to do block reading instead of busy reading. For this reason we implement poll file operation so user can issues command select() to block waiting for file descriptor IO a specified amount of time.

The implementation consists of 2 files: *mote.c* and *mote.h*. A test program that use *moteLoop* device driver is written under *test/c/loopBackTest.c*. Note that *mote.h* is used by application program and also MoteDevice Java Native Interface below.

## 5.0 MoteDevice Java Class Library

To facilities user write program in Java. A MoteDevice class library is written. This library make use of Java Native Interface API to invoke native interface defined at *moteDeviceNative.c*. The native library is built as libMoteDev.so which must put in \$LD\_LIBRARY\_PATH when run. The MoteDevice class exploit the following API :

```
public MoteDevice() // Constructor for Mote Device
public void open(int func, int port) throws IOException // Open mote device
public void close() // Close mote device
public boolean isValid() // Is file descriptor valid ?
public int read(byte buffer[], int count) // Same as read(buffer, 0, count)
public int read(byte buffer[], int start, int count) // Try read count bytes to buffer
public int readFully(byte buffer[], int count) // Same as readFully(buffer, 0, count)
public int readFully(byte buffer[], int start, int count) // Block read until count bytes or error
public int write(byte buffer[], int count) // Same as write(buffer, 0, count)
public int write(byte buffer[], int start, int count) // Try write count bytes from buffer
public int writeFully(byte buffer[], int count) // Same as writeFully(buffer, 0, count)
public int writeFully(byte buffer[], int start, int count) // Block write until count bytes or error
public int setBaudRate(long baudRate) // Set baud rate
public int setDestAddr(long address) // Set destination Mote address
public long getBaudRate() // Get current baud rate
public long getDestAddr() // Get current Mote destination address
public int waitForRead() // wait read forever, same as waitForRead(-1)
public int waitForRead(long timeout) // return > 0 if read ready, = 0 timeout, < 0 error
public int waitForWrite() // wait write forever, same as waitForWrite(-1)
public int waitForWrite(long timeout) // return > 0 if read ready, = 0 timeout, < 0 error
public int waitForReadWrite(long timeout) // return > 0 read/write ready, = 0 timeout, < 0 error
```

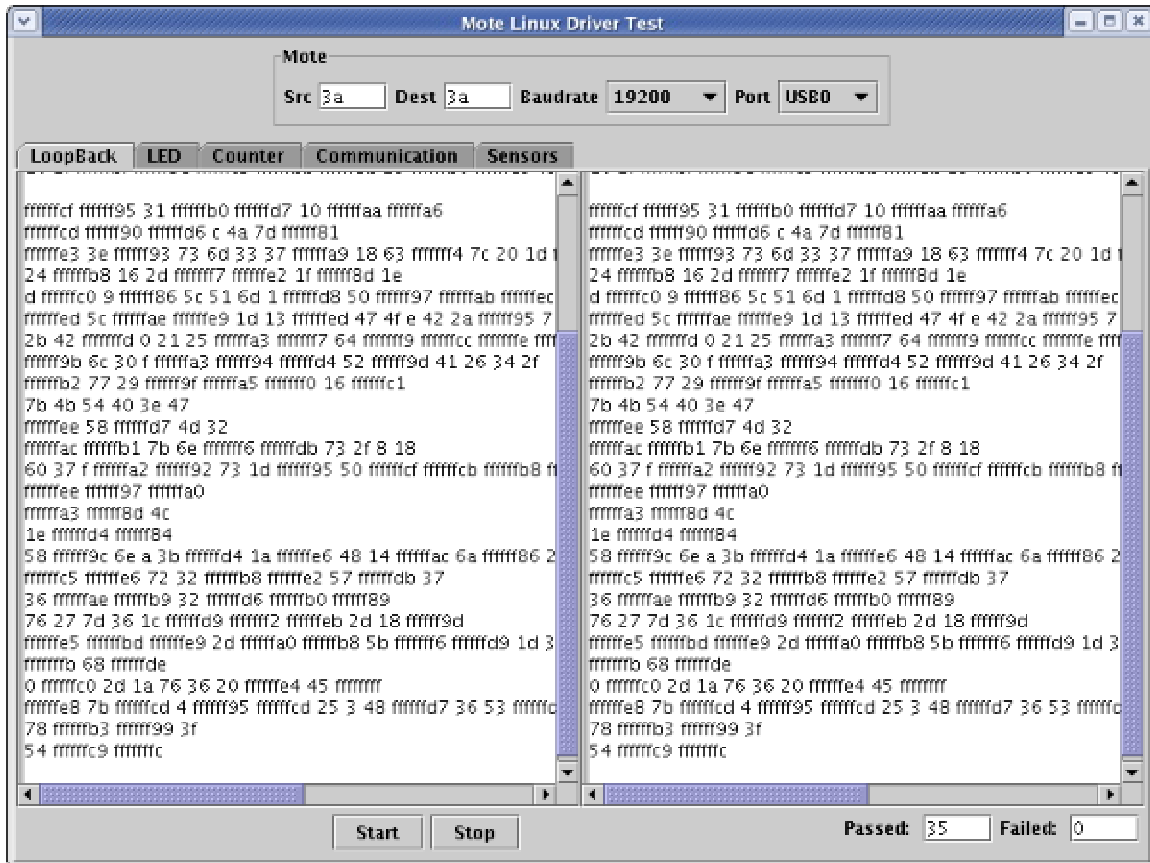
There are port and device function integer constant etc. defined in the same class to ease programmer. The class will map function and port integer to specific mote device name and file read write mote in native code. The library is located at test/java.

## 6.0 Mote Java GUI Application

Mote Linux driver test are created to perform different test of the application. All tests require sets of basic functions such as source, destination, baud rate, and port. The source address is the address of the mote that is directly connected to the Linux machine through serial port. It is retrieved automatically when the application is being started. The destination address can be set to different value for the user to contact different motes and perform required operation. By default, it will be the broadcast address (ffff) of the motes. All addresses in the fields are displayed in hexadecimal. The baud rate is the message rate of the motes. It will be varied for different version of motes. By default, it will be set to 19200. Port is the port used to connect to the motes. The kernel allows the motes to be connected through USB or serial. It also provides interface for different functions including loop back test, led test, get byte counter test, data communication test and sensor test.

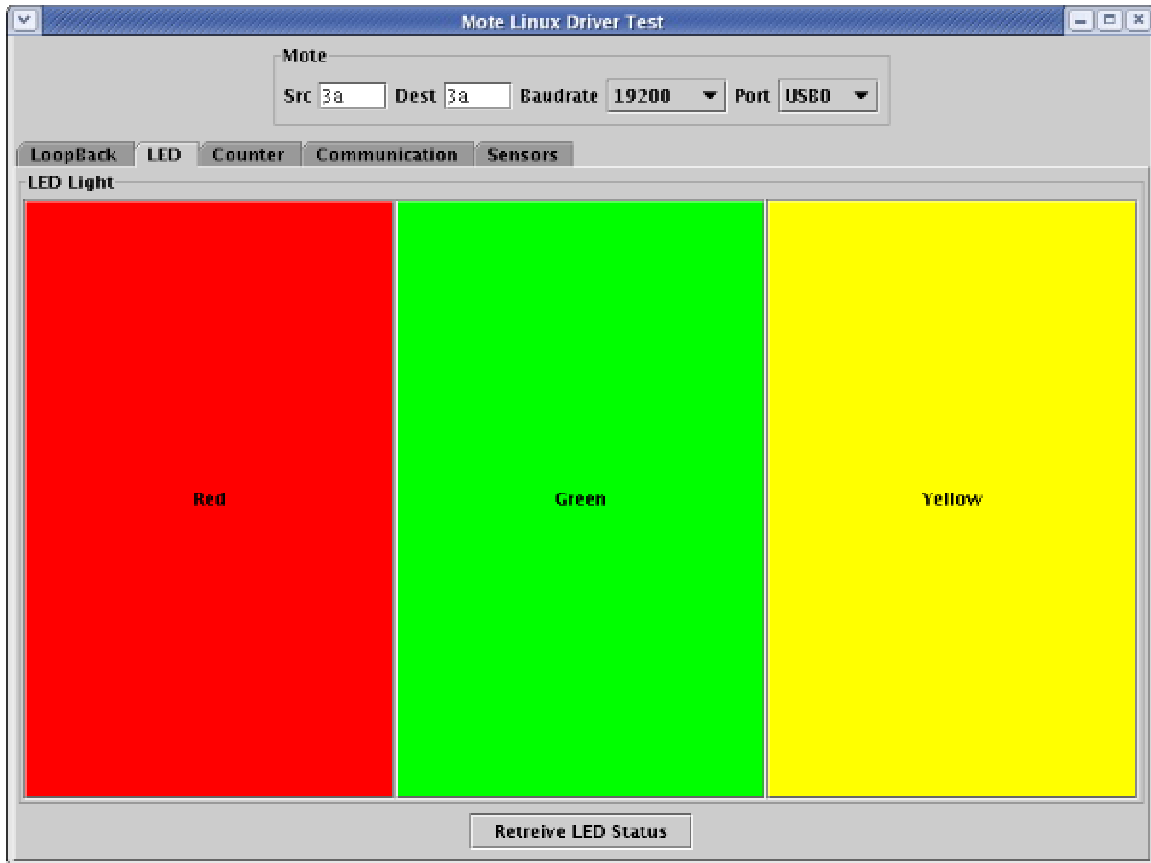
## 6.1 Loop back interface

It will serve as a test showing that the connection between PC and motes are running without problem. The loop back test will start when start button at the bottom are pressed. It will then generate some random packets and retrieve back on the right hand side. The message can be compared byte by byte here. If there is any inconsistency or error, it will be shown in the failed field at the bottom right hand corner.



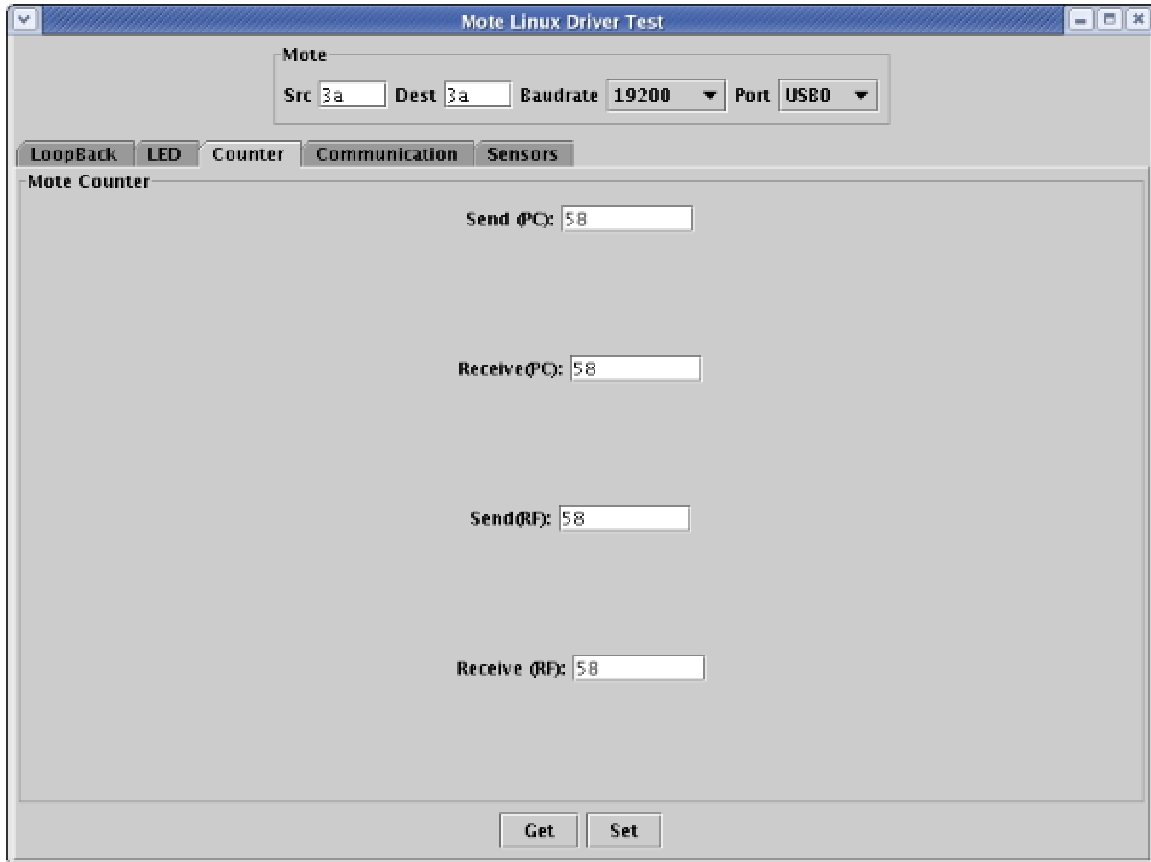
## 6.2 LED interface

LED light interface allow users to set and get the Led light setting on the motes. The light button can be pressed change the light. Destination can be set to obtain the required result. The remote LED light value can also be retrieved using the Retrieve LED status button at the bottom. If destination address is set to broadcast (ffff), all the light within the detectable range will be set to the required value.



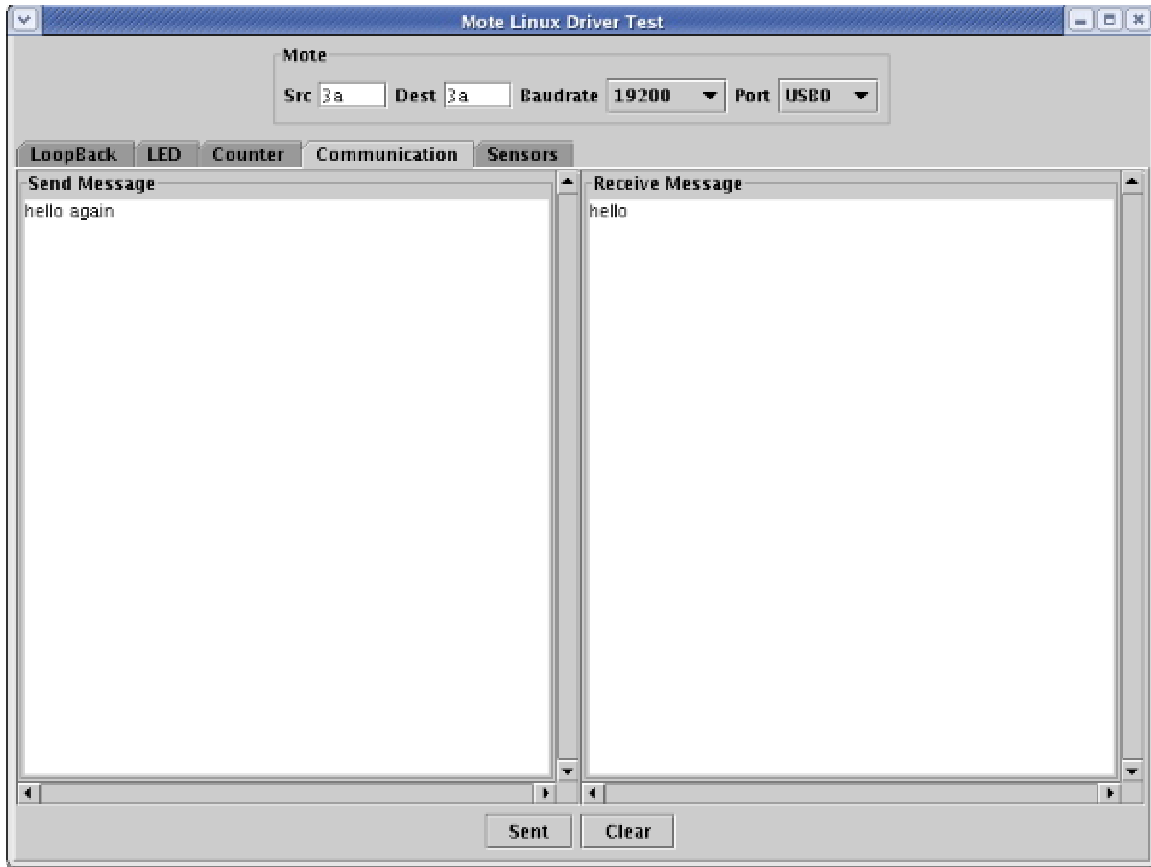
### 6.3 Counter interface

The number of byte being sent and received over the serial port (to PC) and radio (to other motes) are measured by pressing the get button. The value on the motes can also be set to a specific value using the set button.



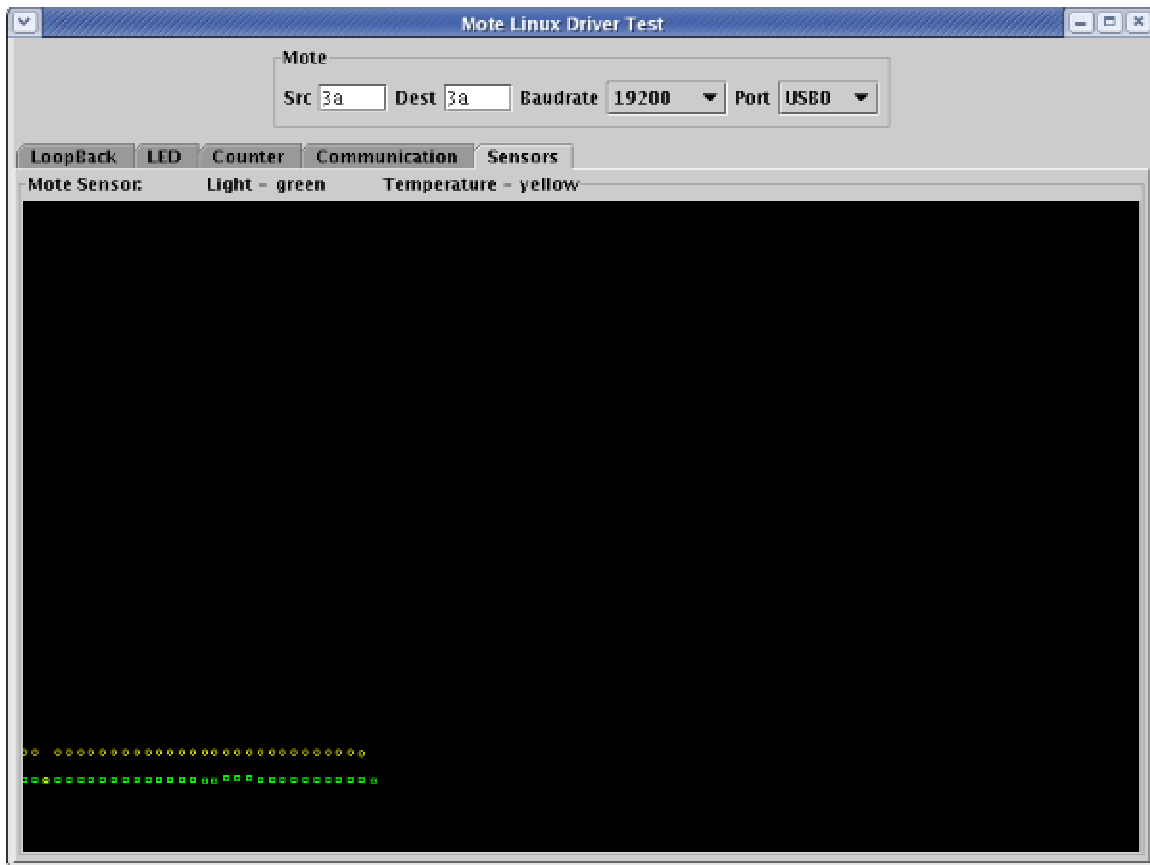
## 6.4 Communication interface

The data message can be sent to the remote destination address. The destination address should be either broadcast or address that connect to a computer. The message is typed in and sent in the left send message text area. Upon receiving, it displays in the right receive message text area. Clear button is provided to clear up the receive message text area.



## 6.5 Sensor interface

The value of the sensor board are retrieved from the destination motes and displayed in a graphical method. Light intensity is in green and temperature is in yellow. A linked list is used to store the retrieved value from the mote. When the value is transverse, the whole screen will shift to the new value.



## 7.0 Installation and Testing

To install, unzip the moteDriver.tar, then type

- (i) copy MoteDriver/ to tinyos/tinyos-1.x/apps
- (ii) cd tinyos/tinyos-1.x/apps/MoteDriver
- (iii) make mica
- (iv) Attach Mote to parallel interface
- (v) make reinstall mica
- (vi) Repeat (iv) and (v) for all Motes
- (vii) ./build.sh (to built mote.ko)
- (viii) ./mote.init start (to load mote kernel module and create /dev/mote\*)
- (ix) cd test/c
- (x) ./build.sh (to built c loopback test program)
- (xi) Attach Mote to USB/Serial port interface
- (xii) ./loopBackTest /dev/moteLoop\_xxx where xxx is the interface that mote attached
- (xiii) cd ../java
- (xiv) ./build.sh (This will built MoteDevice Java library and GUI application)
- (xv) java MoteTest

## 8.0 Discussion

We have changed our implementation from original specification a little e.g. the Linux Interface API. The motivation is that `ioctl()` should only use to control device driver attributes, not send and receive data from device. Also we add a significantly number of device to support more operations and extend our Mote to support remote sensing. This greatly enhanced the usefulness of our driver. We also change the packet format that mote communicate with PC to incorporate more functions. We are able to meet our aggressive goal.

The difficulty in this project involves studying a lot of material which we are not familiar with us such as Linux Device Driver book, TinyOS, Mote documentation. Besides, we need to study the TinyOS source distribution to figure out exactly how it works. During the development of Linux module we also need to study a lot of Linux source code, in particular the file system and device IO part to figure out how to write the module and what kernel API is provided. The process to build module and methodology is also different in the the latest Linux 2.6. Device file `devfs` is also obsolete. We encounter problem to load a module due to header file in source tree has a custom Linux version. We need to frequently search on Internet news group and web site to gather additional information.

NesC Mote driver implementation is also challenging to debug. Since there is no print out to see and we cannot use the Mote simulator for this kernel module. The only way to debug is turn on off LED light and this process is slow. Besides for the radio communication part we also spend quite a lot of time to make it works. Sometime we spend lots of time only to find out that it is Mote radio communication malfunction or only receive but not able to send message. Furthermore, the mote antenna is broken easily. We have broken 4 motes and need to ask friend to fix it on the night before deadline. And of the 5 motes we got, 2 of them are not working, 1 can only receive but not send radio signal.

The demonstration plan is to setup PC and Mote as in Section 1. Then use the Java GUI application to demonstration functionality of our mote driver interactively. We have developed about 4000 lines of code for this project. The responsibility is divided as below:

(i)	NesC mote driver	~800 lines	(Cassidy)
(ii)	Linux Mote Kernel module	~1100 lines	(Kelvin)
(iii)	MoteDevice Java Class	~460 lines	(Kelvin)
(iv)	Java GUI Mote application	~940 lines	(Kelvin)
(v)	C loopbackTest	~170 lines	(Kelvin)

## 9.0 Conclusion

We have developed a Linux Driver for Mote to support varies functionality such as remote sensing of data. On top of it a MoteDevice Java class library is build for application programming to us. We also developed an interactive GUI application to test all functionality provided by our Mote Driver using our MoteDevice class library. We believe that this is a great contribution to the evolving Mote community for sensor network.

## 10.0 References

- [http://www.xbow.com/Products/Product\\_pdf\\_files/Wireless\\_pdf/6020-0042-05\\_A\\_MICA2.pdf](http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0042-05_A_MICA2.pdf)
- Tiny OS Tutorial. <http://www.tinyos.net/tinyos-1.x/doc/>
- Linux Device Drivers, 2<sup>nd</sup> Edition. <http://www.xml.com/ldd/chapter/book/>
- Using Files in Kernel code [http://uqconnect.net/~zzoklan/lkd/using\\_files.html](http://uqconnect.net/~zzoklan/lkd/using_files.html)