# Wiimote - Linux integration

This document describes how to connect Wiimote to a personal computer running Linux operating system. Linux distribution used is Ubuntu 8.04 Hardy, however instructions should apply to other major distributions with minor changes.

Connecting the Wii Remote to a personal computer is done via a Bluetooth connection, therefore a Linux compatible Bluetooth adapter is required. You can test if Linux recognized your Bluetooth device by looking at the dmesg output. After plugging the adapter to a USB port, type dmesg on the command line. Final lines should be similar to the following output:

```
medialab@medialab:~$ dmesg
...
...
[ 2293.704845] Bluetooth: Core ver 2.11
[ 2293.705876] NET: Registered protocol family 31
[ 2293.705881] Bluetooth: HCI device and connection manager initialized
[ 2293.705885] Bluetooth: HCI socket layer initialized
[ 2293.707929] Bluetooth: HCI USB driver ver 2.9
[ 2293.709317] usbcore: registered new interface driver hci_usb
```

If you don't see anything resembling it, any indication of Bluetooth device being recognized or some error message about unidentified device, most probably your Bluetooth adapter is not supported by Linux and you should try another device. Major brands like ASUS, Belkin are supported. To see the list of supported devices visit the following address:

http://www.wiili.org/index.php/Compatible_Bluetooth_Devices

However, beware that the list is not complete and it's quite likely that devices not listed there are already recognized, so you should try to be sure.

Once the device is recognized, generic bluetooth software like development library, bluetooth daemon, etc. could be installed by issuing the following command:

```
medialab@medialab:~$ sudo apt-get install bluez-utils
libbluetooth-dev libbluetooth2
```

Once these are installed, Ubuntu automatically starts bluetooth daemon. In other distributions you may need to do it by hand. At this point you can check whether your Bluetooth device recognizes your Wiimote. Make sure that batteries are installed in Wiimote and press buttons 1 and 2 simultaneously. Right after that, typing the following command on Linux terminal should display a similar output:

```
medialab@medialab:~$ hcitool scan
Scanning ...
    00:1F:32:AC:EC:D2    Nintendo RVL-CNT-01
```

As can be seen above, Bluetooth adapter has found the Wiimote device and displayed its MAC number. Note that depending on your environment and the antenna of your Bluetooth device, you can see a lot of other devices like cellphones or computers listed as well. Make sure that a device with name "Nintendo RVL-CNT-01" is present.

The final piece of software required to make the integration complete is CWiid. It's a package that contains some demo applications and a library for interfacing with Wiimote. Issue the following command to install it's components:

```
medialab@medialab:~$ sudo apt-get install libcwiid1
libcwiid1-dev wmgui
```

Once installed you can confirm that Wiimote integration is complete by running wmgui application and observing the results. Run wmgui as follows:

```
medialab@medialab:~$ wmgui
```

A GUI application will appear. From File menu choose connect. Follow the instructions on the screen and press buttons 1 and 2 of your Wiimote simultaneously and press OK on the dialog box. All four leds on Wiimote should start blinking and after a short pause, you should see "Connected" message written in the status bar of the application. Now you can press various buttons and see it echo on the screen. To see accelerometer and IR data (requires Sensor Bar) select respective options from the Settings menu.

At this point integration of Wiimote with Linux is complete. You can program write programs that process location, accelerometer and button press data using libcwiid API, documentation on which can be found at the following address:

http://abstrakraft.org/cwiid/

## Wiimote - Programming

In order to program Wiimote, you need to have libcwiid1-dev and libbluetooth-dev packages installed under Ubuntu. Under different Linux distributions these packages are named differentli, e.g. on Fedora they are called libcwii-dev and bluez-libs-devel. However they are named, you need two packages that provide cwiid.h and bluetooth.h header files. Once these package are installed, you can write C or C++ programs by including cwiid.h header file and using the functions exported in this file. Also when compiling your program you need to add -lcwiid1 and -lbluetooth switches to the compiler to enable linking with cwiid and bluetooth libraries respectively, otherwise you will get errors about undefined functions during the linking process.

The following section will describe cwiid.h header in detail and give a tutorial how to get started on programming with cwiid library. This document describes version 0.6 of the library.

After including cwiid.h header file in your application, probably the first thing you need to do is to define the following two variables:

```
bdaddr_t g_bdaddr = *BDADDR_ANY;
cwiid_wiimote_t *g_wiimote = NULL;
```

Here we definea  bluetooth address and Wiimote handle structures respectively and initialize them to default values. These will be initialized to values representing actual device later.  Although it is not necessary to know for the library user, these two structures are defined in bluetooth.h and cwiid.h header files respectively as follows:

```
/* BD Address */
typedef struct {
    uint8_t b[6];
} __attribute__((packed)) bdaddr_t;

/* Wiimote struct */
struct wiimote {
    int flags;
    int ctl_socket;
    int int_socket;
    pthread_t router_thread;
    pthread_t status_thread;
    pthread_t mesg_callback_thread;
    int mesg_pipe[2];
    int status_pipe[2];
    int rw_pipe[2];
    struct cwiid_state state;
    enum rw_status rw_status;
    cwiid_mesg_callback_t *mesg_callback;
    pthread_mutex_t state_mutex;
    pthread_mutex_t rw_mutex;
    pthread_mutex_t rpt_mutex;
    int id;
    const void *data;
};
```

Once these two structures are defined, Wiimote programmer can move on to associating them with an actual devices, which

is done by a call to cwiid_connect() function whose prototype is as follows:

```
cwiid_wiimote_t *cwiid_connect(bdaddr_t *bdaddr, int flags);
```

The purpose of this function is to establish a Bluetooth connection with a physical Wiimote device. The address of a bdaddr_t structure set to *BDADDR_ANY and passed as the parameter bdaddr in order to connect to any (one) available Wiimote. Flags parameter represents option flags that can also subsequently be enabled with cwiid_enable. Possible values which can be combined using bitwise-OR operation and their meanings are as follows:

- CWIID_FLAG_MESG_IFC: Enable the message based interfaces (message callback and cwiid_get_mesg).
- CWIID_FLAG_CONTINUOUS: Enable continuous wiimote reports
- CWIID_FLAG_REPEAT_BTN: Deliver a button message for each button value received, even if it hasn't changed.
- CWIID_FLAG_NONBLOCK: Causes cwiid_get_mesg to fail instead of block if no messages are ready.

On success, the function will return a cwiimote_handle to be used in later calls and bdaddr parameter will contain the address of the connected device. In order to initiate Bluetooth scan by the Wiimote device, you should press buttons 1 and 2. Therefore a typical usage of this function is as follows:

```
  ...
printf("Press buttons 1 and 2 on the Wiimote to
connect... ");
fflush(stdout);

/* Establish a continous and non-blocking connection
*/
g_wiimote = cwiid_connect(&g_bdaddr,
CWIID_FLAG_CONTINUOUS|CWIID_FLAG_NONBLOCK);
```

On success g_wiimote should contain an address of a handle

that can be used in further calls, on failure it will contain a NULL value.

Once the connection established, a few check ups can be performed to confirm proper operation of the device. These can be done using cwiid_command whose prototype is as follows:

```
int cwiid_command(cwiid_wiimote_t *wiimote, enum cwiid_command
```

Here, wiimote parameter is a cwiid_wiimote handle that was previously obtained by cwiid_connect() call, command parameter is a command to be executed, and flags parameter is a flag associated with the command. Available commands and their associated flags are as follows:

- `CWIID_CMD_STATUS` - Request a status message (delivered to the message callback) (flags ignored)
- `CWIID_CMD_LED` - Set the LED state. The following flags may be bitwise ORed:
    - `CWIID_LED1_ON`
    - `CWIID_LED2_ON`
    - `CWIID_LED3_ON`
    - `CWIID_LED4_ON`
- `CWIID_CMD_RUMBLE` - Set the Rumble state. Set flags to 0 for off, anything else for on.
- `CWIID_CMD_RPT_MODE` - Set the reporting mode of the wiimote, which determines what wiimote peripherals are enabled, and what data is received by the host. The following flags may be bitwise ORed (Note that it may not be assumed that each flag is a single bit - specifically, `CWIID_RPT_EXT = CWIID_RPT_NUNCHUK | CWIID_RPT_CLASSIC)`:
    - `CWIID_RPT_STATUS`
    - `CWIID_RPT_BTN`
    - `CWIID_RPT_ACC`
    - `CWIID_RPT_IR`
    - `CWIID_RPT_NUNCHUK`
    - `CWIID_RPT_CLASSIC`
    - `CWIID_RPT_EXT`

The function returns zero on success and you can see the immediate effect like rumbling of a Wiimote, flashing leds, etc. or nonzero on error. Continuing with our example, the following code would power on leds 2 and 3:

```
cwiid_command(g_wiimote, CWIID_CMD_LED,
CWIID_LED2_ON|CWIID_LED3_ON);
```

In order to receive IR and accelerometer data respective devices should be enabled by a similar command:

```
cwiid_command(g_wiimote, CWIID_CMD_RPT_MODE,
CWIID_RPT_IR|CWIID_RPT_ACC|CWIID_RPT_BTN);
```

Finally, report about current state of Wiimote can be obtained by cwiid_get_state() call whose prototype is as follows:

```
int cwiid_get_state(cwiid_wiimote_t *wiimote, struct cwiid_sta
```

As in all previous calls, the wiimote parameter is a Wiimote handle obtained by cwiid_connect() call, second parameter is a structure that will be explained later. For now it suffices to say that upon success, it contains information that describes device's state. As usual, the function returns zero on success and nonzero on error. The following code will obtain and print the battery state of a device:

```
struct cwiid_state g_wii_state;
cwiid_get_state(g_wiimote, &g_wii_state);
printf("- Battery: %d%%\n\n", (int)(100.0 *
g_wii_state.battery / CWIID_BATTERY_MAX));
```

In order to make the examples less cluttered, none of the calls check for return values; please make sure that you check the return value of every call and act accordingly.

Now, important fields of the cwiid_state struct will be explained. It's defined in cwiid.h as follows:

```
struct cwiid_state {
```

```
    uint8_t rpt_mode;
    uint8_t led;
    uint8_t rumble;
    uint8_t battery;
    uint16_t buttons;
    uint8_t acc[3];
    struct cwiid_ir_src ir_src[CWIID_IR_SRC_COUNT];
    enum cwiid_ext_type ext_type;
    union ext_state ext;
    enum cwiid_error error;
};
```

The fields of interest to us, are ir_src and acc arrays. As the name suggests, ir_src array provides information about the IR sensors. The array size is a constant CWIID_IR_SRC_COUNT which is defined to be 4 in the header file, but with a regular sensor bar, we obtain two sources of infrared data, one from each of the sensor bar. Each element of the array is a struct cwiid_ir_src which is defined in the header as follows:

```
struct cwiid_ir_src {
    char valid;
    uint16_t pos[2];
    int8_t size;
};
```

Here the valid field contains whether the information provided in pos array is valid. Since a sensor may get out of range this field needs to be checked every time before the position data in pos array is used. pos array can be indexed using CWIID_X and CWIID_Y constants; as can be guessed, each index contains respective coordinate.

The acc array within the cwiid_state structure contains information obtained from the accelerometer. It can be used to calculate roll and pitch of a Wiimote. Before making use of this data, accelerometer calibration should be obtained using the following call:

```
struct acc_cal wm_cal;
cwiid_get_acc_cal(wiimote, CWIID_EXT_NONE, &wm_cal);
```

Now that wm_cal structure contains accelerometer calibration information we can proceed to obtaining current accelerometer readings and calculate roll and pitch as follows:

```
a_x = ((double)mesg->acc[CWIID_X] -
wm_cal.zero[CWIID_X]) /
      (wm_cal.one[CWIID_X] - wm_cal.zero[CWIID_X]);
a_y = ((double)mesg->acc[CWIID_Y] -
wm_cal.zero[CWIID_Y]) /
      (wm_cal.one[CWIID_Y] - wm_cal.zero[CWIID_Y]);
a_z = ((double)mesg->acc[CWIID_Z] -
wm_cal.zero[CWIID_Z]) /
      (wm_cal.one[CWIID_Z] - wm_cal.zero[CWIID_Z]);
a = sqrt(pow(a_x,2)+pow(a_y,2)+pow(a_z,2));

roll = atan(a_x/a_z);
if (a_z <= 0.0) {
    roll += PI * ((a_x > 0.0) ? 1 : -1);
}
roll *= -1;
pitch = atan(a_y/a_z*cos(roll));
```

Here we assume that a previous cwiid_get_state() call has been made and mesg actually is a pointer to a cwiid_state struct, hence mesg->acc is an array containing accelerometer information.

Finally, in order to disconnect from a Wiimote device and leave it in a proper state, cwiid_disconnect() function should be called.  It's defined as follows:

```
int cwiid_disconnect(cwiid_wiimote_t *wiimote);
```