Wiiリモコン

出典: Wiili

このページは、Wiiリモコン（英語では**Wii Remote** または **Wiimote**）に関する技術的な説明を行う。Wiiリモコンの一般的な説明に関してはWikipedia の当該項目を参照。新しい情報についてはWiiリモコンドライバに追加され、随時このページに掲載する。

翻訳元（英語）

Wii Remote

目次

Communication

The Wiimote communicates with the Wii via a Bluetooth wireless link. The Bluetooth controller is a Broadcom 2042 chip, which is designed to be used with devices which follow

the Bluetooth Human Interface Device (HID) standard, such as keyboards and mice. The Bluetooth HID is directly based upon the USB HID standard, and much of the same documentation applies.

When queried with the Bluetooth Service Discovery Protocol (SDP), the Wiimote reports back a great deal of information, listed at Wii_bluetooth_specs#spd info. In particular it reports:

Name	Nintendo RVL-CNT-01
Vendor ID	0x057e
Product ID	0x0306

The Wiimote does not appear to require any of the authentication or encryption features of the Bluetooth standard. In order to interface with it, one must first put the controller into *discoverable* mode by either pressing the 1 and 2 buttons at the same time, or by pressing the red sync button under the battery cover. Once in this mode, the Wiimote can be queried by the Bluetooth HID driver on the host. If the HID driver on the host does not connect to the Wiimote within 20 seconds, the Wiimote will turn itself off. Holding down the 1 and 2 buttons continuously will force the Wiimote to stay in discoverable mode without turning off. This does not work with the sync button, however. When in discoverable mode, the Player LEDs will blink. The number that blink will correspond to the remaining battery life, similar to the meter on the Wii home menu where # of bars = # of lights.

HID Interface

The HID standard allows devices to be self-describing, using a HID descriptor block. This block includes an enumeration of *reports* that the device understands. A report can be thought of similar to a network port assigned to a particular service. Reports are unidirectional however, and the HID descriptor lists for each port the direction (Input or Output) and the payload size for each port. Like all Bluetooth HID devices, the Wiimote reports its HID descriptor block when queried using the SDP protocol. A human-readable version of the block is shown at Wii_bluetooth_specs#HID_Descriptor, and is summarized in the following table:

Output

Report ID	Payload Size	Known Functions
0x11	1	Player LEDs, Force Feedback
0x12	2	Report type / ID
0x13	1	IR Sensor Enable
0x14	1	Enable speaker
0x15	1	Controller status
0x16	21	Write data

Input

Report ID	Payload Size	Known Functions
0x20	6	Expansion Port
0x21	21	Read data
0x22	4	Write data
0x30	2	Buttons only
0x31	5	Buttons | Motion Sensing Report
0x32	16	Buttons | IR??

0x17	6	Read data
0x18	21	Speaker data
0x19	1	Mute speaker
0x1a	1	IR Sensor Enable 2

0x33	17	Buttons | Motion Sensing Report
0x34	21	Buttons | IR??
0x35	21	Buttons | Motion Sensing Report
0x36	21	Buttons | IR??
0x37	21	Buttons | Motion Sensing Report
0x38	21	Buttons | Motion Sensing Report | IR
0x3d	21	Buttons | IR??
0x3e	21	Buttons | Motion Sensing Report | IR??
0x3f	21	Buttons | Motion Sensing Report | IR??

Note that "Ouput" refers to packets are sent from the host to the Wiimote, and "Input" refers to packets that are sent from the Wiimote to the host. For clarity, the convention in this document is to show packets including the Bluetooth header (in parentheses), report ID (also called *channel ID* in some places), and payload, as described in sections 7.3 and 7.4 of the Bluetooth HID specification. Each byte is written out in hexidecimal, without the 0x prefix, separated by spaces. For example

```
(a1) 30 00 00
```

is a DATA input packet (0xa1), on channel 0x30, with the two byte payload 0x00, 0x00.

It actually seems that Force Feedback is accessible through ALL output channels the same way.

Note: On some wii-motes, IR wont start transmitting until a read-report 0x38 has been sent.

Linux Notes

On Linux hosts, the BlueZ bluetooth drivers, included in recent 2.4 and 2.6 kernels, allow communication with Bluetooth devices. The BlueZ stack includes a kernel module, hidp, which connects to Bluetooth HID-capable devices. To instruct the kernel to connect to the Wiimote, you will need the bluez-utils package, which includes the hidd daemon. First put the Wiimote into discoverable mode, by either pressing buttons 1 and 2 on the Wiimote or using the switch hidden near the batteries. Then:

```
hidd --search
```

This will scan for all HID-capable devices and connect to them. After receiving the

connection, the Player LEDs will continue to blink, but the Wiimote will not power down until the connection is broken. By default, hidd will maintain the connection for 30 minutes beyond the last packet exchange. Received packets can be observed using the hcidump command:

```
hcidump -X
```

Sending packets to the Wiimote requires access to the control and interrupt Bluetooth sockets established by hidd. A modified version of hidd is needed to retain these sockets, and a customized version for the Wiimote will be posted soon.

It should be noted that the HIDP driver creates a /dev/input/eventX when the Wiimote connection is established (see dmesg for a message reporting which event device it is). Preliminary tests suggest this interface is unsuitable for accessing the Wiimote because of the unusual HID information it reports. This is why a lower-level userspace daemon is thought to be required to communicate with the Wiimote directly via Bluetooth sockets.

MacOS X Notes

DarwiinRemote 0.3a works pretty well - it allows the Wiimote to pair with OSX, and control the mouse (shakily). It supports IR sensors and rotation, and, although it claims to operate the LEDs and rumble, a quick go on my powerbook didn't seem to work.

See the blog, 0.3a annoucement and The sourceforge page for details.

Windows XP Notes

There is a Windows "driver" called GlovePIE[1]. It can read the buttons and accelerations, and it can set the leds and the rumble.

The wiimote is accessible exactly like a USB HID device: using Delphi and the TJvHidDeviceController component. You can access all the info with the demo program (ReadWriteDemo\SimpleHIDWrite.dpr). I am using W2K.

The wiimote can be paired with Windows XP and a generic driver is installed. It appears in the game controller's panel but cannot be configured in any way.

Some people have had trouble sending reports to the Wiimote. Some people have got around this by using a different bluetooth stack. Being unable to send reports to the wiimote makes it impossible to read the acceleration data. The Bluesoleil bluetooth stack is most recommended to fix these problems. The problem of sending reports occurs with the TJvHidDeviceController component.

This is probably a bug of the Bluetooth stack. Alternatively it could be the bug of the HID DLL that writing reports have to use the size of the largest report declared (the correct amount of bytes is written though). Robert Marquardt 02:14, 19 December 2006 (EST)

Inputs

Buttons

There are 12 buttons on the Wiimote. Four of them are arranged into a directional pad, and the rest are spread over the controller.

By default, whenever a button is pressed or released, a packet is sent to the host via HID input report 30H, with a payload containing a 2-byte bitmask with the current state of all the buttons.

The button state also seems to be included in the first two bytes of all other input reports.

Some of the bits in the first two bytes don't seem to be directly related to button presses, and are somewhat unknown.

For example, when the A button is pressed, this HID DATA input packet is received:

```
(a1) 30 00 08
```

and when it is released, this is packet received:

```
(a1) 30 00 00
```

The bit assignments (in big endian order) for the buttons are:

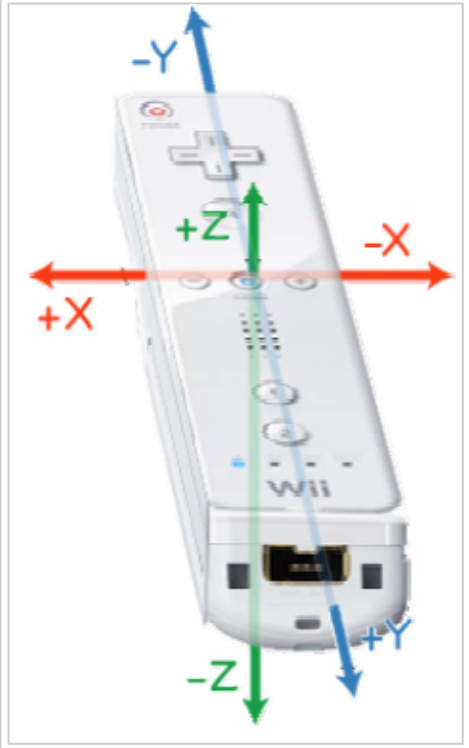
Button	Number (dec)	Value (hex)
Two	1	0x0001
One	2	0x0002
B	3	0x0004
A	4	0x0008
Minus	5	0x0010
Z Acceleration, bit 6 (report 3E) or bit 2 (report 3F)	6	0x0020
Z Acceleration, bit 7 (report 3E) or bit 3 (report 3F)	7	0x0040
Home	8	0x0080
Left	9	0x0100
Right	10	0x0200
Down	11	0x0400
Up	12	0x0800
Plus	13	0x1000
Z Acceleration, bit 4 (report 3E) or bit 0 (report 3F)	14	0x2000
Z Acceleration, bit 5 (report 3E) or bit 1 (report 3F)	15	0x4000

? Unknown ?	16	0x8000

The power button is unusual in that it sends a disconnect request when held down instead of emitting a normal button code.

Motion Sensor

The motion of the remote is sensed by a 3-axis linear accelerometer located slightly left of the large A button. The integrated circuit is the ADXL330 (data sheet), manufactured by Analog Devices. This device is physically rated to measure accelerations over a range of at least +/- 3g with 10% sensitivity.



Coordinate system used by Wiimote

Inside the chip is a small micromechanical structure which is supported by springs built out of silicon. Differential capacitance measurements allow the net displacement of the tiny mass to be converted to a voltage, which is then digitized. It is important to note that the sensor does not measure the acceleration of the Wiimote, but rather the force exerted by the test mass on its supporting springs. Due to the sign convention used, this quantity is proportional to the net force exerted by the player's hand on the Wiimote when holding it. Thus, at rest on a flat table, the accelerometer reports vertical force of +g (the mass can be normalized away into the aribitrary units), and when dropped reports a force of nearly zero.

The sensor uses a right-handed coordinate system with the positive X-axis to the left, and the positive Z-axis pointing upward, when the remote is held horizontally, as shown in the diagram at the right. Forces on each axis are digitized to 8 bit unsigned integers, with the zero scale set to 0x80. Manufacturing and calibration defects of course cause some intrinsic zero offsets. However, the Earth's gravitational field and a flat, level surface upon which to rest the remote, in principle, allow any offsets or chip skew to be measured and calibrated away in software. The calibration values are stored near the start of the Wiimotes flash RAM. Decomposing the force measured by the sensor into rotation and linear components is tricky, and discussed further on the Motion analysis page.

The Wiimote does not normally report motion sensor readings to the host, but can be requested by sending SET_REPORT request to channel 0x12:

```
(52) 12 00 31
```

The 3rd byte is a bitmask. 0x01 turns on the rumble, and 0x04 turns on continuous output. If 0x04 is not set, packets are only output when the values change (almost always when the

motion sensor is enabled). If 0x04 is set, values are output continuously (it's obvious when channel/mode 0x30 is chosen, which disables motion readback. With byte3=0x04, the button values are output multiple times a second. With byte3=0x00, they are output only when you press or release a button). The 4th byte specifies which HID channel to which log sensor output should be sent. After receiving this command once, the Wiimote will send back stream of DATA INPUT packets on the requested channel/mode (0x31 in the above example), where in reports 0x31, 0x33, 0x35, and 0x37 the 5th, 6th and 7th bytes contain the X, Y, and Z readings of the accelerometer. A sample packet when the Wiimote is at rest, face up, on a table is:

```
(a1) 31 40 20 86 8a a5
```

where 0x86 is the X-axis measurement, 0x8a is the Y-axis measurement, and 0xa5 is the Z-axis measurement. The first two bytes of the payload, 0x40 and 0x20, are the button values and some other bits. These bits are important in reports 0x3E and 0x3F. The button values are still in the same place and have the same meanings as in the Buttons section.

Other channels can be selected for motion sensor reports, including 0x31, 0x33, 0x35, 0x37, 0x3e and 0x3f. If either 0x3e or 0x3f are selected then sensor readings will alternate between the two channels. 0x3E contains the X Acceleration in the third byte of the payload, wheras 0x3F contains the Y Acceleration in the third byte of the payload. The Z Acceleration is stored in the button flags, as shown in the button table above. 0x3E and 0x3F use the remaining 18 bytes for what we think is IR data.

The length of the report payload will depend upon the channel, as show in the table in HID Interface section. Channels which have payloads longer than needed for the motion sensor will apparently pad the end of the packet with 0xff bytes:

```
(a1) 33 40 00 86 8a a5 ff ff ff ff ff ff ff ff ff ff ff ff ff
```

Motion sensor reports can be stopped by setting the output channel to 0x30:

```
(52) 12 00 30
```

It seems that the channel mode is actually a mode selection / bitmask. Button output is always enabled. Mode 0x30 is just buttons, 0x31 is motion sensor, 0x32 is IR camera (???), and 0x33 is both IR camera and motion sensor. The two sets of data are tacked onto each other, first the button values, then the three motion sensor bytes, then the camera bytes. Other modes seem to include motion data and / or IR data in different ways.

Calibration

The zero points, and gravity values, for the three accelerometer axes, are stored near the start of the Wiimote's flash memory. See the Flash Memory section below for details.

Calibration Data

Calibration data for the onboard accelerometer is stored in the Wii's memory, starting at address 0x16. This information is repeated at 0x20.

```
0x16      zero point for X axis
0x17      zero point for Y axis
0x18      zero point for Z axis
0x19      unknown
0x1A      +1G point for X axis
0x1B      +1G point for Y axis
0x1C      +1G point for Z axis
0x1D      unknown
0x1E-0x1F checksum?
```

IR Sensor

During R&D, Nintendo discovered the motion sensors were not accurate enough to use the remote to control an on-screen cursor. To correct this, they augmented the remote with an infrared image sensor on the front designed to locate two IR beacons within the controller's field of view. The beacons are housed within a device misleadingly called the *sensor bar*. The sensor bar is powered by the Wii base unit, and contains 2 groups of IR LEDs, spaced 7.5 inches apart. Each group is composed of 5 LEDs, but homemade sensor bars have been effective with fewer LEDs, so long as the intensity is sufficient. The cable from the Wii to the sensor bar only carries power. No information is passed either to or from the sensor bar, and the intensity is not modulated in any way.

These two sources of IR light are tracked by a PixArt sensor in the front of the Wiimote housing. By tracking the locations of these two points in the sensors 2D field of view, the system can derive more accurate pointing information. Not much is known about this feature yet, but circumstantial evidence from the Nintendo/PixArt press release suggests that Nintendo is using a PixArt System-on-a-Chip to process the images on-board the Wiimote and sends the minimum information needed for tracking back to the base unit. Transmitting full 2D images constantly would require a prohibitive amount of bandwidth, especially when multiple remotes are in use.

Wiimote detects and transfers up to four IR hotspots back to the host. Various amounts of data can be requested, from position values only, position and size, to position, size and pixel value. The amount of different configurations are quite numerous, and also ties in with the connected peripheral device.

marcan's info

Here are the reports you need to send to activate IR transmission:

```
(52) 12 00 33
(52) 13 04
(52) 1A 04
(52) 16 04 B0 00 30 01 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
(52) 16 04 B0 00 06 01 90 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
(52) 16 04 B0 00 08 01 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
(52) 16 04 B0 00 1A 01 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
(52) 16 04 B0 00 33 01 33 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Python / pseudocode follows.

```
FEATURE_DISABLE = 0x00
FEATURE_ENABLE = 0x04

IR_MODE_OFF = 0
IR_MODE_STD = 1
IR_MODE_EXP = 3
IR_MODE_FULL = 5

CMD_SET_REPORT = 0x52

RID_LEDS = 0x11
RID_MODE = 0x12
RID_IR_EN = 0x13
RID_SPK_EN = 0x14
RID_STATUS = 0x15
RID_WMEM = 0x16
RID_RMEM = 0x17
RID_SPK = 0x18
RID_SPK_MUTE = 0x19
RID_IR_EN2 = 0x1A

MODE_BASIC = 0x30
MODE_ACC = 0x31
MODE_ACC_IR = 0x33
MODE_FULL = 0x3e

def send(cmd,report,data):
    Send cmd, report, data to Wiimote. If cmd is CMD_SET_REPORT, this amounts to sending data to th
    Interface code to Wiimote goes here.

def setmode(mode,cont):
        aux = 0
        if rmbl:
                aux |= 0x01
        if cont:
                aux |= 0x04
        send(CMD_SET_REPORT,RID_MODE,[aux,mode])

# size here is redundant, since we can just use len(data) if we want.
def senddata(data,offset,size): # see writing to data: [[#On-board Memory].
        of1 = offset >> 24 & 0xFF #extract offset bytes
        of2 = offset >> 16 & 0xFF
        of3 = offset >> 8 & 0xFF
        of4 = offset & 0xFF
        data2 = data + [0]*(16-len(data)) # append zeros to pad data if less than 16 bytes
        if len(data2) > 16:
                data2 = data2[:16] # crop data if we have too much
        # format is [OFFSET (BIGENDIAN),SIZE,DATA (16bytes)]
        send(CMD_SET_REPORT,RID_WMEM,[of1,of2,of3,of4,size]+data2)

# this seems to be the minimal code to get it to work
setmode(MODE_ACC_IR,0)
send(CMD_SET_REPORT,RID_IR_EN,[FEATURE_ENABLE])
send(CMD_SET_REPORT,RID_IR_EN2,[FEATURE_ENABLE])
senddata([8],0x04B00030,1) # enable IR data out
senddata([0x90],0x04B00006,1) # sensitivity constants (guessed, Cliff seems to have more data, but
senddata([0xC0],0x04B00008,1)
senddata([0x40],0x04B0001A,1)
senddata([IR_MODE_EXP],0x04B00033,1) # enable IR output with specified data format
```

```
# this is Cliff's version pythonified, probably more accurate as far as sensitivity. Works pretty n
setmode(MODE_ACC_IR,0)
send(CMD_SET_REPORT,RID_IR_EN,[FEATURE_ENABLE])
send(CMD_SET_REPORT,RID_IR_EN2,[FEATURE_ENABLE])
senddata([1],0x04B00030,1) # seems to enable the IR peripheral
senddata([0x02, 0x00, 0x00, 0x71, 0x01, 0x00, 0xaa, 0x00, 0x64],0x04B00000,9)
senddata([0x63, 0x03],0x04B0001A,2)
# this seems incorrect - for FULL IR mode, we must use FULL wiimote mode (0x3e).
# otherwise the data is probably garbled.
senddata([IR_MODE_FULL],0x04B00033,1)
senddata([8],0x04B00030,1) # Enable data output. Can be specified first it seems, we don't really n
```

Output format EXP is three bytes per dot recognized. Bytes 0 and 1 are X and Y, byte 2 is the LSBs of X and Y and a size value. In binary:

```
xxxxxxxx yyyyyyyy yyxxssss
```

No dot is indicated by 0xff data. This means you will not see any change in the output data even if you did do all the initialization correctly, unless you actually point the Wiimote at a (powered) sensor bar!

Output format for FULL is unknown, but we know the data is interleaved. Data sent to report 0x3e contains the data for the first two dots recognized, data sent to 0x3f contains the data for the next two dots recognized, if any.

Note that the original hidd patch has trouble with 0x00 bytes in commands. Make sure you're sending the right data.

Cliff's info

Set the report 0x12 to a report number such as 0x33.

You need to start by setting reports 0x13 and 0x1a to 0x04 to enable transmission.

Then to activate the IR Sensor you need to write to the wiimote's memory. Set 0x04b00030 to 1, then set the sensitivity, then set 0x04b00033 to 5, then set 0x04b00030 to 8.

The sensitivity data is 9 bytes at 0x04b00000 and 2 bytes at 0x04b0001a. For midrange sensitivity, it's {0x02, 0x00, 0x00, 0x71, 0x01, 0x00, 0xaa, 0x00, 0x64}, and {0x63, 0x03}.

Outputs

Player LEDs

The bottom edge of the remote body contains 4 blue LEDs. These LEDs are used during normal play to indicate that the remote is in Bluetooth discoverable mode (all blinking), or to indicate the player number of the controller (one light illuminated). The LEDs are independently controllable, however, using SET_REPORT ouput packet to channel 11, which has a payload size of 1. The most-significant 4 bits control each LED, with bit 4 corresponding to the player 1 LED. Channel 11 also can control the rumble feature, so it is best to keep the 4 least-significant bits zero in order to avoid vibrating the controller. For

example, this packet turns on the player 1 LED only:

```
(52) 11 10
```

Since each LED has its own bit, any combination of LEDs can be illuminated.

```
[ ][ ][ ][ ] = (52) 11 00
[♦][ ][ ][ ] = (52) 11 10
[ ][♦][ ][ ] = (52) 11 20
[♦][♦][ ][ ] = (52) 11 30
[ ][ ][♦][ ] = (52) 11 40
[♦][ ][♦][ ] = (52) 11 50
[ ][♦][♦][ ] = (52) 11 60
[♦][♦][♦][ ] = (52) 11 70
[ ][ ][ ][♦] = (52) 11 80
[♦][ ][ ][♦] = (52) 11 90
[ ][♦][ ][♦] = (52) 11 A0
[♦][♦][ ][♦] = (52) 11 B0
[ ][ ][♦][♦] = (52) 11 C0
[♦][ ][♦][♦] = (52) 11 D0
[ ][♦][♦][♦] = (52) 11 E0
[♦][♦][♦][♦] = (52) 11 F0
```

Force Feedback

Force feedback is provided via a "rumble pack" style device inside the Wiimote body. It is roughly an off-center weight connected to a motor which can be activated to cause the controller to vibrate. The motor can be activated by sending a SET_REPORT output packet to channels 0x11, 0x13, 0x14, 0x15, 0x19 or 0x1a with the least significant bit set:

```
(52) 13 01
```

And the vibration can be turned off by clearing the bit:

```
(52) 13 00
```

So far, it appears all channels are equivalent, though using channel 0x11 is not advised because it also controls the player LEDs.

Speaker

A small speaker is embedded in the top face of the controller to provide audio feedback. The control method of this speaker is still somewhat unknown.

Marcan and DesktopMan have discovered the protocol used for sending sound to the

Wiimote. As far as I know, they have not documented it yet.

I have asked about their efforts, and this is what I have learned: The bottom line is, they haven't been able to play a real sound yet. Wiimote sound is not presently usable, and their working assumption is that it's a jitter problem.

Basically, the Wiimote speaker plays back 4-bit, 6kHz sound. The sound is split in 20-byte frames that are sent one at a time, 150 times a second.

It appears that precise timing is of the utmost importance, and that so far, jitter issues have made it impossible to play real sounds.

They have only done limited testing thus far, and I understand that either they are not yet confident enough in their method to publish results, or they have not had the time to do so.

As for me, my working theory is that there could be a buffering system. After all, the 8051 microcontroller built into the Broadcom BCM2042 chip that is at the core of the Wiimote has 22kB of RAM, some of which could be used as an audio buffer. Wiimote audio only requires 3kB per second, after all. Hernick 13:17, 12 December 2006 (EST)

Onboard Memory

Flash Memory

On a blank wiimote, which was bought seperately and has never been connected to a Wii, and has never received a report other than 0x17 (read memory) with the first byte as 00; the memory is structured like this:

Addresses 0x0000 to 0x003F:

```
A1 AA 8B 99 AE 9E 78 30 A7 74 D3 A1 AA 8B 99 AE
9E 78 30 A7 74 D3 82 82 82 15 9C 9C 9E 38 40 3E
82 82 82 15 9C 9C 9E 38 40 3E 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

This is basically two sequences, each repeated twice:

```
A1 AA 8B 99 AE 9E 78 30 A7 74 D3
A1 AA 8B 99 AE 9E 78 30 A7 74 D3
82 82 82 15 9C 9C 9E 38 40 3E
82 82 82 15 9C 9C 9E 38 40 3E
```

The three bytes starting at 0x16 and 0x20 (the first 3 bytes of the 3rd and 4th line above) store the calibrated zero offsets for the accelerometer. Presumably the 9C 9C 9E stores the force of gravity on those axes.

Addresses 0x0040 to 0x0FC9: all zeros on a newly bought Wiimote.

Address 0x0FCA to 0x12B9: Mii Data block 1, all zeros on a newly bought Wiimote.

Address 0x12BA to 0x15A9: Mii Data block 2, all zeros on a newly bought Wiimote.

Address 0x15AA to 0x15FF: all zeros on a newly bought Wiimote.

Addresses 0x1600 to 0xFFFF: Don't exist. They return an error if you try to read from them. You won't get the error if you start reading from at or before the 0x15FF boundary, but you will just get zeroes for the invalid bytes in that case.

For the Flash memory the address is only 2 bytes. So address 0x010000 is the treated the same as address 0x0000. This is true all the way up to 0xFF0000. That byte (0x00FF0000) is always ignored unless the most significant byte (0xFF000000) has bit 2 (0x04) set.

Control Registers

The control registers have bit 2 (0x04) set in the first byte. Bit 0 (0x01) in the first byte is the rumble flag, and is not considered part of the address, so 0x05A20000 is the same address as 0x04A20000. Only registers 0x04A20000 to 0x04A30000 are readable. But they don't seem to provide any useful information on a fresh new Wiimote.

Registers 0x04000000 - 0x49FFFFF return error 7 (instead of error 8 like normal nonexistant memory). Perhaps it is write only, or perhaps it doesn't exist.

Registers 0x04A00000 - 0x4A1FFFF don't exist because they return error 8 (like normal non-existant memory)

Registers 0x04A20000 - 0x4A30000 are readable!! But are all the byte 0xFF. Except registers whose address ends in 0xFF, which is the byte 0x00.

Registers 0x04B00000 - 0x4BFFFFF return error 7 when you read them, but some of them are known to be settable, and to control the IR camera in the Wiimote. The first 9 bytes are part of the IR sensitivity settings. Bytes 0x04B0001A and 0x04B0001B are also part of the IR sensitivity settings. Byte 0x04B00030 turns IR on and off (8 is on, 1 is off in order to set sensitivity). Byte 0x04B00033 sets the IR mode (0x33 tells it to return IR data as 4 lots of 3 bytes).

Registers 0x04C00000 - 0x4FFFFFF return error 7 when you read them.

Reading and Writing

We can now read data. Here's the command:

```
(52) 17 FF FF FF FF SS SS
```

FF FF FF FF is the offset (big-endian format). SS SS (big-endian format) is the size in bytes.

It was once thought that this was the correct format:

```
(52) 17 0r 00 FF FF SS SS
```

The low bit of the first byte (of the address) is the usual rumble flag. The rumble flag is not part of the address. You should always set this bit to whatever the current rumble state should be. It does not affect the address. If you don't set this bit to the current rumble state you will accidentally change the rumble state just by reading (or writing) the memory.

There is only 5.5K of Flash RAM on the Wiimote, which is addressed between 0x0000 and 0x15FF. But there are also internal control registers which can be set, but only some of them can be read. The control registers begin with 0x04. The returned packets store addresses as only two bytes. And addresses wrap around after 0xFFFF.

The responses look like this:

```
       btns? SE FF FF data
       vvvvv vv vv vv v------>
(a1) 21 80 00 f0 11 f0 80 6c 8c c7 c2 5d 2e bd 40 00 4e 00 99 80 08 b1
```

E is the error flag. E is 8 if you try to read from bytes that don't exist (are greater than 5.5K, or 0x15FF), and 7 if you try to read from write-only registers, or 0 if no error. S (the high nibble, needs to be shifted right 4 bits) is the size in bytes, minus one, for the current packet. FF is the offset of the current packet (big-endian). Everything else is the data (16 bytes max, if there is an incomplete trailer S is set to something other than 0xF and the data is padded out with zeros.) If you requested more than 16 bytes, then you will receive multiple packets, unless you get an error where E is 8.

Writing data is as follows:

```
       FF FF FF FF SS data
       vv vv vv vv vv v------>
(52) 16 00 00 00 00 10 57 69 69 57 69 6c 6c 52 6f 63 6b 59 6f 75 21 21
```

FF,SS same meaning as reading (you can only write 16 bytes at a time here, so SS is only one byte). 16 bytes of data follow. It seems we get some kind of acknowledge on Input 0x22.

Note that only bit 2 (0x04) in the first byte (of payload) is considered part of the address. It causes the Wiimote to write to registers instead of flash memory. Bit 0 (0x01) of the first byte should be set to the current state of Rumble, otherwise it will accidentally turn rumble on or off. The second byte is only used for register addresses. Writing to flash memory ignores this byte, so 0x010000 is the same as 0x0000.

Cadex' findings

I tried to read all addresses in the 32 bits of address space - once with, once without Nunchuk attached - and here is what I found out:

For read access, the 32 bits of the address seem the have the following meaning:

```
----MM-F CCCCCCC- AAAAAAAA AAAAAAAA
Meaning:
- : Don't care (= can be 0 or 1 without changing the result)
MM : Mode:
     00 : Read flash memory
     01 and 10: Read control registers
     11 : ???, always returns errorcode 6, except for addresses 0x0CA0---- which return errorcode 8
F : Force Feedback bit - Enable (1) or disable (0) Force Feedback
CCCCCCCC : Don't care when reading Flash Memory (MM == 11)
           Relevant when reading control registers (MM == 01 or 10) when upper four bits must be 10
AAAAAAAA AAAAAAAA : Lower 16 Bits of the address
```

Control registers: Access to the control registers is selected by setting MM (Bits 2 and 3 of Byte 1) to 01 or 10 (-> Byte 1 == 0x04 or 0x08). Furthermore, the upper 4 bits of Byte 2 must be set to 1010 (0xA), otherwise errorcode 7 is returned. Since the lowest bit of Byte 2 doesn't seem to care for read access, this leaves the following address spaces:

0x04A0---- / 0x08A0---- : Return errorcode 8

0x04A2---- / 0x08A2---- : All 0xFF except for addresses 0x04A2xxFF / 0x08A2xxFF, which have value 0x00

0x04A4---- / 0x08A4---- : Nunchuk attached: 0xFF / Nunchuk not attached: Errorcode 7

0x04A6---- / 0x08A6---- : Errorcode 7

0x04A8---- / 0x08A8---- : Errorcode 7

0x04AA---- / 0x08AA---- : Errorcode 7

0x04AC---- / 0x08AC---- : Errorcode 7

0x04AE---- / 0x08AE---- : Errorcode 7

So it seems that only the following regions of the address space are interesting for read access; all other address ranges return an error when trying to read them, or are mapped to these:

0x00000000 - 0x000016FF : Flash memory content

- In contrast to the information in section "Flash Memory", the readable area of the flash memory of MY Wiimote doesn't end at 0x15FF. Instead, it ends at address 0x16FF, and this is a hexdump of that extra region in my Wiimote:

```
00001600:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
(00001610 - 000016d0: all zeros)
000016d0:  00 00 00 FF 11 EE 00 00 33 CC 44 BB 00 00 66 99
000016e0:  77 88 00 00 2B 01 E8 13 00 00 00 00 00 00 00 00
000016f0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

0x04A20000 - 0x04A2FFFF : Data registers of the Wiimote, default 0xFF (except for addresses ending on FF which contain value 0x00), meaning not yet known

0x04A40000 - 0x04A4FFFF : Data registers of the Nunchuk (if attached!), default 0xFF, meaning not yet known

Communication with the Nunchuk

I found the first piece of the expansion port communication puzzle, I'm able to get at least parts of the status of an attached Nunchuk controller. This is really just a first step, but by telling you what I found out, probably other people will soon find out more:

As mentioned in the previous section, reading addresses in the range 0x04A40000-0x04A4FFFF will only work when a Nunchuk is attached (in which case 0xFF values are returned), whereas errorcode 7 is returned if no Nunchuk is present. This made me believe that addresses 0x04A4---- are used to receive data from the Nunchuk.

Next step I did was trying to write a single byte to all addresses 0x04--0000, once with and once without Nunchuk attached, and see what errorcodes I get back.

With the Nunchuk attached, I got back errorcode 0 (= OK) when writing to 0x04A40000 and 0x04A50000, without Nunchuk I got errorcode 7. This made me believe that addresses 0x04A4---- are not only used to receive data from, but also send data to the Nunchuk. I suppose it doesn't make a difference if you use address 0x04A4---- or 0x04A5----, my previous experiments showed that at least for reading data, the bit that makes the difference seems to get ignored. Probably the same applies for writing data.

Next step I did was simply writing 0x01 bytes to addresses in the 0x04A4---- region, then reading that region again and seeing if anything changed. When writing 0x01 to address 0x04A40040, suddenly the data read was no longer 0xFF values only:

```
04a40000:  FF FF FF FF FF FF FF FF 7B 71 86 D0 62 32 FE FE
04a40010:  FE FE FE 5F FE 5E FF FF FF FF 5E FF 5E FF FF FE
04a40020:  FE FE FE 5F FE 5D FE FE FE FE FE 5F FE 5D FE FE
04a40030:  FE FE FE 5F FE 5D FE FE FE FE FE 5F FE 5D FE FD
04a40040:  FF FF FF 5E FF 5E FF FF FF FF FF 5E FF 5E FF FF
04a40050:  FF FF FF 5E FF 5E FF FF FF FF FF 5E FF 5E FF FF
04a40060:  FF FF FF 5E FF 5E FF FF FF FF FF 5E FF 5E FF FF
04a40070:  FF FF FF 5E FF 5E FF FF FF FF FF 5E FF 5E FF FF
04a40080:  FF FF FF 5E FF 5E FF FF FF FF FF 5E FF 5E FF FF
04a40090:  FF FF FF 5E FF 5E FF FF FF FF FF 5E FF 5E FF FF
04a400a0:  FF FF FF 5E FF 5E FF FF FF FF FF 5E FF 5E FF FF
04a400b0:  FF FF FF 5E FF 5E FF FF FF FF FF 5E FF 5E FF FF
04a400c0:  FF FF FF 5E FF 5E FF FF FF FF FF 5E FF 5E FF FF
04a400d0:  FF FF FF 5E FF 5E FF FF FF FF FF 5E FF 5E FF FF
04a400e0:  FF FF FF 5E FF 5E FF FF FF FF FF 5E FF 5E FF FF
04a400f0:  84 FE FE 5F FE 5D FE FE FE FE FE 5F 9A 7D FE FE
04a40100:  FF FF FF FF FF FF FF FF 7B 71 85 D0 61 BA FE FE
04a40110:  FE FE FE 5F FE 5E FF FF FF FF 5E FF 5E FF FF FE
04a40120:  FE FE FE 5F FE 5D FE FE FE FE FE 5F FE 5D FE FE
04a40130:  FE FE FE 5F FE 5D FE FE FE FE FE 5F FE 5D FE FD
```

What I figured out so far:

- The data at 0x04A4000A-0x04A4000C seems to be acceleration data (values "85 D9 61" in the hexdump above). These values seem to be similar, but still a bit different from the acceleration values provided by the Wiimote (values seem to jump when certain boundaries are reached)
- Bit 0 of the byte at 0x04A4000D shows the state of the "Z"-button; the state of the "C"-button does not seem to be contained in that byte though.
- The byte at 0x04A40008 seems to contain the X value of the Nunchuks joystick, 0x04A40009 the Y value
- Data seems to get repeated every 0x100 bytes (see example), so probably bits 8-15 of the address (= the third byte) is ignored when reading from the Nunchuk, only addresses 0x04A40000-0x04A400FF are relevant (see example)
- It seems to make a difference at which address one starts to read the data. Reading 0x10 bytes starting at 0x04A40000 will contain the interesting values, whereas reading 0x10 (or 0x0F) bytes starting at 0x04A40001 will return completely different data.
- Writing an other value than 0x01 or writing to a different address in the 0x04A4004-region changes the middle part of the data being returned. For example, this is what was returned when writing 0x02 instead:

```
04a40000:  FF FF FF FF FF FF FF FF 7B 71 85 D9 61 B3 FE FE
04a40010:  FE FE FE 56 FE 57 FF FF FF FF 57 FF 57 FF FF FE
04a40020:  FE FE FE 56 FE 56 FE FE FE FE FE 56 FE 56 FE FE
04a40030:  FE FE FE 56 FE 56 FE FE FE FE FE 56 FE 56 FE FD
04a40040:  FF FF FF 57 FF 57 FF FF FF FF FF 57 FF 57 FF FF
04a40050:  FF FF FF 57 FF 57 FF FF FF FF FF 57 FF 57 FF FF
04a40060:  FF FF FF 57 FF 57 FF FF FF FF FF 57 FF 57 FF FF
04a40070:  FF FF FF 57 FF 57 FF FF FF FF FF 57 FF 57 FF FF
04a40080:  FF FF FF 57 FF 57 FF FF FF FF FF 57 FF 57 FF FF
04a40090:  FF FF FF 57 FF 57 FF FF FF FF FF 57 FF 57 FF FF
04a400a0:  FF FF FF 57 FF 57 FF FF FF FF FF 57 FF 57 FF FF
04a400b0:  FF FF FF 57 FF 57 FF FF FF FF FF 57 FF 57 FF FF
04a400c0:  FF FF FF 57 FF 57 FF FF FF FF FF 57 FF 57 FF FF
04a400d0:  FF FF FF 57 FF 57 FF FF FF FF FF 57 FF 57 FF FF
04a400e0:  FF FF FF 57 FF 57 FF FF FF FF FF 57 FF 57 FF FF
04a400f0:  84 FE FE 56 FE 56 FE FE FE FE FE 56 9A 76 FE FE
04a40100:  FF FF FF FF FF FF FF FF 7B 71 85 D9 61 BF FE FE
04a40110:  FE FE FE 56 FE 57 FF FF FF FF 57 FF 57 FF FF FE
04a40120:  FE FE FE 56 FE 56 FE FE FE FE FE 56 FE 56 FE FE
04a40130:  FE FE FE 56 FE 56 FE FE FE FE FE 56 FE 56 FE FD
```

I made a small Python script (requires PyBlueZ) which does nothing else than connect to the

Wiimote and repeatedly dump the 6 bytes of which I think they represent the Nunchuk's state: Nunchuktest

Other findings:

- It appears that by XORing the bytes at 0x04A40008-0x04A4000D with the values E8 17 E8 B6 E8 B6, you can get a more stable reading from both the accelerometers and the joystick. Also, bit 1 of the byte at 0x04A4000D looks to be the status of the C button.
- The classic controller is also readable using this technique, altough the button values haven't been worked out yet.

Carl Kenner's Findings

Some of the bits count backwards and some count forwards. To get proper acceleration values for X and Z, and to get proper joystick values, I use the following (Pascal) function:

```
function NunchukFixByte(a: Byte): Byte;
begin
  Result:=(a and 232) + 7-(a and 7) + 16-(a and 16);
end;
```

Afterwards I subtract 105 from the X, and 108 from the Z. This gives me the RawForceX and RawForceY values in GlovePIE. Note that GlovePIE uses the Direct3D axes centred on 0, so you will need to negate or add 128 or whatever for your system.

The Y axis (that GlovePIE calls Z) is different, I use this function instead (note this function returns a value roughly between -50 and 50, with +50 being forwards like in Direct3D):

```
function NunchukFixZAcc(a: Byte): SmallInt;
begin
  Result:=(a and 201) + 6-(a and 6) + 48-(a and 48);
  if Result<128 then Result:=Result+256;
  Result:=233-Result;
end;
```

For the buttons, bit 1 is the inverse of the C button, while bit 0 is the Z button. I assume that it has a similar inversion patter to the Y acceleration (that I call Z).

Joystick axes should be roughly between 0 and 100. Joystick and accelerometers both have a range of -50 to 50 (unlike the Wiimote).

Note that my Nunchuk has a lot of extra data in the middle of its block that Cadex's doesn't have. I haven't investigated what it means yet, I'm hoping it is calibration data.

I send a report to read those 16 bytes, at the start of Run Time, then whenever I receive the reply for that report (which looks exactly like a read of 0x0000 flash mem by the way), I send another read-memory report. When I receive report 0x20 to say that the Nunchuk has been plugged in again, I need to activate nunchuk mode again (by setting the required address to 1) and then send another read-memory report (and of course sending report 0x12

again too).

Expansion Port

The expansion port on the bottom of the unit is used to connect the remote to auxiliary controllers which augment the input options of the Wiimote. Auxiliary controllers use the Bluetooth interface of the remote to communicate with the Wii base unit, allowing them to be much simpler and cheaper to build. Currently available controllers are the Nunchuk and the Classic controller.

The expansion port itself is a ???-style connector with 6 contacts. Two of the contacts are slightly longer, which allows them to make contact first when a plug is inserted. Presumably these contacts carry Vcc and ground, much like standard USB connectors. The remaining 4 contacts most likely carry data, and the small number suggests that a serial interface of some sort is used. The connector is not suitable for charging the remote. More information is needed on the electrical specifications of this connector.

The status of the expansion port is indicated with report 20H. This report is sent whenever the status of the expansion port changes, or when the computer sends output report 15H to request the status.

After an attachment is plugged in or unplugged, no other reports are sent except 20H. The computer needs to send an output request (report 12H) to retrieve new input reports.

The format of input report 20H consists of the report number, then two bytes which are presumably the button state like in other reports, then a status flags byte, then two unknown 00 bytes, then the battery level.

Status byte flags

bit	Value (hex)	Flag
0	0x01	? 0 ?
1	0x02	any attachment plugged in
2	0x04	Speaker enabled ?
3	0x08	IR sensor enabled ?
4	0x10	LED 1
5	0x20	LED 2
6	0x40	LED 3
7	0x80	LED 4

Note that bit 2 (0x04) above is turned on by sending report 0x14 with the flag 0x04 in the payload. It is turned off by sending report 0x14 without the 0x04 flag. Bit 3 (0x08) above is turned on by sending EITHER report 0x13 with the flag 0x04 in the payload, OR report 0x1A with the flag 0x04 in the payload. You can turn it off by sending EITHER report 0x13 or 0x1A without that flag. You can, for example, turn it on with report 0x13 and then off

again with report 0x1A.

For example, when the classic controller or nunchuk is plugged in, input report 20H is sent:

```
(a1) 20 00 00 02 00 00 C0
```

The 02H indicates the classic controller or nunchuk is plugged in. The C0H is the battery level, and has nothing to do with the nunchuk or classic controller. It will slowly decrease the more you use up the batteries, but sometimes it increases.

When the classic controller or nunchuk is unplugged, input report 20H is sent again, but with 00 instead of the 02:

```
(a1) 20 00 00 00 00 00 C0
```

Batteries

You can read the battery level with report 0x20. It is received when something is plugged in, or unplugged from, the expansion port. Or you can request it by sending report 0x15 with the payload set to anything without bit 1 (rumble) set.

```
(a1) 20 00 00 02 00 00 C0
```

The 0xC0 at the end is the battery level (in this case, for the brand new alkaline batteries that come with it).

Note that you may receive this message unsolicited. Whether this occurs or you request this message, you will need to send report 0x12 with the report number again before you will receive more data.

Random Findings

Sending a byte to ID 0x15 returns the Extension status report (at 0x20, see above), besides setting rumble. Seems everything sets rumble if bit 0 of the first byte is set.

The third byte in the extension status report (00/02 above) can take other values. It seems only that bit is the status, other bits can take on other values depending on the bytes sent to 0x13 and 0x14. For example, sending 0x04 to ID 0x13 makes the values in byte 3 be 08/0a instead of 00/02.

By directly communicating with the Wiimote on the low level layer of L2CAP connections (using channels 0x11 (=output to the Wiimote) and 0x13 (=input from the Wiimote)) instead of using the higher level HID-layer, one can send command sequence "52 1A 00 ??" (0x52 = SET_REPORT, 0x1A = "IR sensor enable 2", 0x00, ?? = any byte value) to the Wiimote. "IR

sensor enable 2" actually only has a 1 byte payload, but when a 2 byte payload is sent instead and the second byte is ??, then from that point on the Wiimote precedes all messages it sends with HID transaction header byte ?? instead of the usual 0xA1 (0xA1 = DATA_INPUT). This might be caused by a buffer overflow and might even prohibit proper communication of that Wiimote with the Wii, seeming like the Wiimote is broken (to be confirmed, I don't have a Wii yet so I can't try - See Talk:Wiimote for some more information on this). Removing and reinserting the batteries won't help, but normal behaviour can be restored by simply sending "52 1A 00 A1" which sets the normal value of 0xA1 again. Besides being a potential security problem (it's probably possible to send even more than only this 1 byte of additional data, so one might be able to change other stuff as well by sending even more data), this probably isn't of any help though.

It also seems to be possible to use HID command DATA_OUTPUT (0xA2) instead of SET_REPORT (0x52). for communication with the Wiimote, so sending sequence "(A2) 15 00" for example (0xA2 = DATA_OUTPUT, 0x15 = "Controller status", 0x00) is equivalent to using "(52) 15 00" (0x52 = SET_REPORT, 0x15 = "Controller status", 0x00).

関連項目

- Wiiリモコンリバースエンジニアリング
- モーションセンサー
- Wiiリモコンドライバ
- ヌンチャク
- クラシックコントローラ
- Wiimote/Hardware

外部リンク

- Ch0p のミラーページ。wiili.org がダウンした場合はこのミラーを編集し、復旧後マージすること。

カテゴリ: 開発