

Apple iPod Remote Control Protocol

Generations 2 & 3

I reverse-engineered a second-generation iPod remote (the version that has the touch-sensitive scroll wheel). The remote probably works exactly as-is with the first generation iPod (with the mechanical scroll wheel). The newest third-generation ipods (thinner, rounder, and with 4 buttons in-line) have a similar remote with an identical circuit board, but with a different square 4-pin remote connector -- the new pinout hasn't been completely figured out, and it's not known if the same protocol is used.



medium size
large size (100kB)



medium size
large size (100kB)

Remote is labeled "MITSUMI" and "66-6616A". The cable has 6 pins: 3 analog signals are fed directly to the headphone jack, 3 go to a small microcontroller and handle the remote-control functions. To reduce noise, they do not share a common ground in the remote (but I can still hear the remote affecting the sound).

Disassembly

The shiny exterior that looks like shiny metal really is - don't try to bend it! I thought it was just a metalized plastic, but it's nice to see such a nice quality case. I first considered this a cover that could be snapped off, but it's really the body of the remote; remove the white plastic portion - the clip and back of the remote - by prying it out of the metal case by "b" in the phrase "assembled in china". The pc board (including the headphone jack) will remain in the case. Probably the easiest way to remove the plastic is to slip a thin screwdriver on the same side as the hold switch, just past the green dot (away from the switch, not on top of it). A picture would be helpful, but I haven't tried to put it back together yet - that looks like a challenge. There are two pins in the metal case that hold the pc board; these don't bend and make removing the pc board somewhat difficult.

Pinout

The generation 2 remote connector is compatible with standard 1/8th inch headphone jack, and also looks like the same connector used in the iBook AV cable (no confirmation on this, though).

It contains six unshielded wires, which are soldered to the PC board.

morganw has kindly provided the information (and pictures) of the generation 3 connector. All of the remote functionality has been moved to a unique 4-pin square connector, and two additional contacts may have new signals on them.

Generation 2



medium size
medium size, color-coded
large size (100kB)

I'd love to see this in a gallery!

Generation 3



large size (100 kB)

These connectors are so photogenic!

Connections

Wire Color	Generation 2	Generation 3	Signal
black	Tip	Tip	Audio, left
white	Ring (Next-to-tip)	Ring	Audio, right
red	Ring 2 (3rd ring from end)	Ring 2	Audio ground
green	Sleeve (4th ring from end)	pin 1	Data to iPod
blue	inner ring	pin 4	Power to remote, +3.3v
yellow	outermost ring	pin 3	Digital ground
		pin 2	no connect / unknown
		Sleeve	no connect / unknown

Generation 3

The third generation iPod seems to use the same remote (internally and externally) as the second generation, but with a different connector. It's still unknown if the microcontroller contains different firmware (and so it may use a different protocol). Here's a picture of morganw's remote:



large size (400 kB)

Remote Control IC

The remote uses a general-purpose Microchip 12C508A microcontroller ([datasheet](#)):



medium size
large size (100kB)

The chip has 8 pins: 2 for power, 6 are general purpose I/O. Reset and a 4 MHz oscillator are internal to the chip. It runs on 3.0 - 5.5 volts and has 512 words of program memory, 25 bytes of RAM, and a 1 microsecond instruction time. The only peripheral (other than I/O) is a timer. This may be used, but it appears that the external clock on pin 5 (T0CKI) is used as a general purpose I/O.

Two other unpopulated 5-pin IC's (IC2 and IC3) are on the button side of the PCB; not sure what the purpose of these would be.

Schematic

I didn't trace the whole button-side of the schematic, but the rest of it is pretty simple. Because we have the datasheet of the microcontroller, the data output (TP7) and power (TP1) connections are well defined electrically.

Testpoints:

TP1 = IC1 pin 1, VDD (3.3v), connected to blue wire.

TP2 = (pulled up to VDD by R3)

TP3 = (pulled up to VDD by R2)

TP4 = SW2 (pulled up to VDD by R1)

TP5 = IC1 pin 5 (pulled up to VDD by R5)

TP6 = IC1 pin 6 (pulled up to VDD by R4)

TP7 = IC1 pin 7. Connected via 1K ohm resistor R7 to the green wire, the data out to the iPod.

TP8 = IC1 pin 8, VSS (ground), connected to yellow wire.

TP9 = [probably hold switch, SW1] (pulled up to VDD by R6)

IC2 (unpopulated) - connects to SW2 & SW4

IC3 (unpopulated) - connects to TP5, SW5

Communications Protocol

Electrical:

digital CMOS output (low = 0v, high = 3.3v) through a 1K ohm resistor, normally low.

Logical:

I've only measured the second generation protocol. I assume the 3rd generation is the same, but the code in the processor could be totally different.

I measured about 108.75 uSec/bit, but my equipment wasn't the most capable; 9600 baud is 104 uSec/bit and it's a safe bet that this is actually what's used, and jives with other peoples' measurements. Also, since the microcontroller relies on an RC oscillator, it won't be totally spot-on. My measurement could also have been messed up by a non-integral bit time between words (which I couldn't detect too well).

Upon pressing the button, the output goes high for about 45msec and then sends the first version of the code. This high signal is probably used to wake the iPod when sleeping. After that, the remote repeats a pattern of waiting about 26msec with a low output, and then sending the second version of the pattern (4msec), until the button is released. I wasn't able to see if something different happened when the button was released.

The first version of the code (transmitted when button is first pressed) is:

(...1111) 0111111111 0101111111 0xxx01111111 (0000....)

where the xxx depends on which button is pressed. The second version of the code follows this format:

(...0000) 1111 0111111111 0101111111 0xxx0111111 1 (0000....)

where xxx depends on the button:

VOL+	SW6	010
VOL-	SW2	110
SKIP>>	SW3	001
<<SKIP	SW4	101
PLAY/PAUSE	SW5	100
(button release)		000

Although my limited test equipment wasn't able to capture it, other people report (thanks!) that a button release code is sent upon (guess what) button release.

This looks like the standard asynchronous protocol implementing a 8-bit, no parity, 2 stop bit (8N2) protocol (or it could be also defined as 8-bit, mark parity, 1 stop bit depending on how you look at it). This is similar to the often-used 8N1 protocol, except there is an extra '1' between characters; it's not known if the iPod actually needs this bit. Each group of 11 bits above is classified composed of one start bit (a 0), followed by 8 data bits (least significant bit first), then two hard-to-classify ones. These ones could be either two stop bits, a stop bit and a mark parity bit, or a stop bit and one just-for-fun mark bit.

So, restated, the 3-byte message sent is:

VOL+	0xFF 0xFD 0xF2
VOL-	0xFF 0xFD 0xF3
SKIP>>	0xFF 0xFD 0xF4
<<SKIP	0xFF 0xFD 0xF5
PLAY/PAUSE	0xFF 0xFD 0xF1

(button release)

0xFF 0xFD 0xF0

There is an interesting quirk brought on by the fact that usual serial protocols require the data line to be signaling high during the time inbetween characters, but the iPod lowers the signal inbetween the 3-byte packets (although it does keep it high inbetween bytes). At least 4 high high bits precede valid data, which seems to be enough to satisfy the iPod's UART. But, after the three bytes are sent, the data line quickly goes low. This isn't usual, and the iPod UART seems to interpret this string of zeros as any other UART would: a start bit, 8 bits of low data (0x00), and then a missing stop bit (a missing 1). The iPod firmware apparently depends on receiving this 0x00, but it's not known if it actually checks to see that the stop bit is missing. If emulating the remote with a serial port, you should send the 0x00.

Example Microcontroller Code

I made a remote emulator using an Atmel AVR microcontroller. I know that doesn't sound like a reasonable thing to do -- after all, the iPod comes with a perfectly suitable microcontroller -- but it's a good start at the design of a protocol translator. This only implements two buttons - forward and reverse - but it's easily expandable.

The chip I picked is the Atmel ATtiny12-8PI. This is a small 8-pin device with 6 usable I/O pins, built-in 1.2 MHz RC oscillator (factory calibrated to $\pm 1\%$ accuracy), 1K of flash memory, 32 bytes of RAM, and 64 bytes of EEPROM. You probably noticed the lack of UART - instead I'm just bit-banging the output. It's ok for this single use, but when I eventually try to simultaneously receive data from my car's bus, I'll have to rewrite this code. This version also runs off of 5 volts and won't run at 3.3v, so it requires some interface circuitry - you may want to use the 3.3v version, especially if you want to use the iPod's power, or it's power output as a "connected" signal.

The schematic is pretty simple; I'll just describe how the 8 pins of the microcontroller connect:

- pin 1 - Reset - pull up to +5v with a 50K ohm resistor (you can remove the resistor if you program this as an input, but I haven't done that yet)
- pin 2 - PB3 - reverse button. Connect with a normally-open pushbutton to ground. No pullup needed.
- pin 3 - PB4 - forward button. Connect with a normally-open pushbutton to ground. No pullup needed.
- pin 4 - ground. Besides the power supply, connecting this to the metal back of the iPod worked (instead of connecting to the digital ground in the remote connector).
- pin 5 - PB0 - not used
- pin 6 - PB1 - not used
- pin 7 - PB2 - output. This needs to be reduced to 3.3 volts, so I used a series divider - this connects to a 220 ohm resistor, which goes to the center of the divider, which connects through a 330 ohm resistor to ground. I used a 10k ohm resistor from the center to the actual ipod input. This divider is far from optimal - it draws waay too much current (9 milliamps!), which is almost enough to pull the output down to 3.7 volts, almost eliminating the need for a divider -- but those were the spare parts I had laying around.
- pin 8 - +5 volts

[Source code is here](#). Read the notes in the code describing how to program the chip so that the RC oscillator is calibrated.

Others' Work

Bernard Leach has been tackling the iPod side of the remote interface. His [ipod port of linux](#) includes a driver for the remote, where he controls a UART. He sees 4 interrupts, one for each byte including the messed-up 0x00 terminator. When four bytes are accumulated (there seems to be no synchronization

rule in software that specifies which byte is the first; hopefully the iPod firmware does this), the bytes are passed to this translation code: (version 0.3 shown)

```
static int ipod_remote_buttons[] = {KEY_1, KEY_2, KEY_3, KEY_4, KEY_5};
static void ipod_remote_process_input(struct ipod_remote *ipod)
{
    static int last_button = 0;

    if ( ipod->data[2] == 0xf0 ) {
        input_report_key(&ipod->dev, last_button, 0);
    }
    else {
        int pos = ipod->data[2] - 0xf1;
        if ( pos >= 0 && pos < 5 ) {
            last_button = ipod_remote_buttons[pos];
            input_report_key(&ipod->dev, last_button, 1);
        }
    }
}
```

He uses the defines KEY_1 ... KEY_5, arbitrarily from the HP-HIL protocol rather than defining his own incompatible values.

Next...

Interfacing it to my car! (current state of that project: I wrote software to read the BMW I-BUS, but I haven't tested it long-term to ensure that it handles noise and packet collisions gracefully)

[back to my homepage](#)

[you might be interested in my sega vmu page \(as bender would say, "it gives me street cred" -- it's higher rated in google than even Sega's site!\)](#)