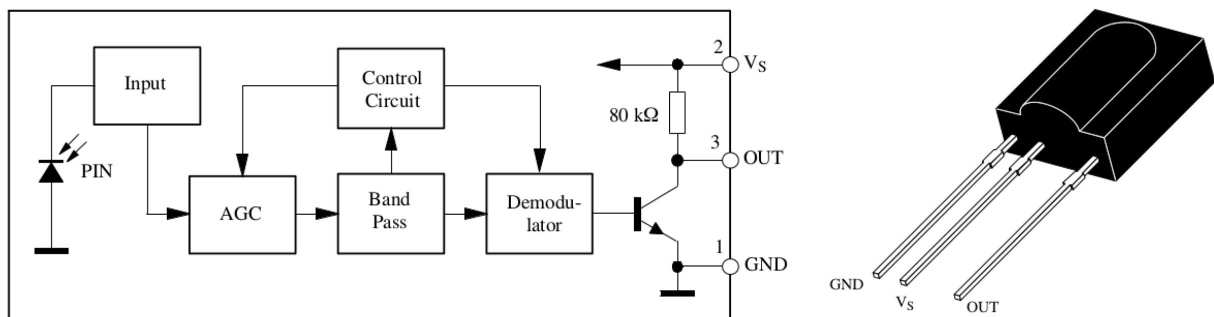
Receiving IR telecommand signals

Links

http://www.righto.com/2009/08/multi-protocol-infrared-remote-library.html
http://z3t0.github.io/Arduino-IRremote/
https://arduino-info.wikispaces.com/IR-RemoteControl
https://en.wikipedia.org/wiki/RC-5

Hardware

Sensor: TSOP1736, connected to Arduino Uno



This sensor responds to signals with 36kHz carrier.

Connections:

Vs - +5V
OUT – Pin 11 (or any other digital pin)
GND – GND

Eventually a low pass filter 100Ω / 4.7µF can be inserted into the Vs line.

The supply voltage must be at least 4.5V, so when connected to a 3.3V microcontroller like the WEMOS the 5V supply must be used and the OUT signal must be reduced to max. +3.3V (for example with zener diodes).

Basic firmware

The standard library for this is Irremote.

Example by Ken Shirrif, the developer of the library:

```
#include <IRremote.h>
int RECV_PIN = 11;
IRrecv irrecv(RECV_PIN);
decode_results results;
```

```
void setup(){
  Serial.begin(115200);
  Serial.println("Enabling IRin");
  irrecv.enableIRIn(); // Start the receiver
  Serial.println("Enabled IRin");
}

void loop() {
  if (irrecv.decode(&results)) {
    Serial.println(results.value, HEX);
    irrecv.resume(); // Receive the next value
  }
  //delay(10);
}
```

This gives the hex value of the received code.

Notes:

By having a look at the source code, you see what the data structure decode_results is made of:

```
class decode_results
{
	public:
		decode_type_t		decode_type;	// UNKNOWN, NEC, SONY, RC5, ...
		unsigned int		address;	// Used by Panasonic & Sharp [16-bits]
		unsigned long		value;		// Decoded value [max 32-bits]
		int			bits;		// Number of bits in decoded value
		volatile unsigned int	*rawbuf;	// Raw intervals in 50uS ticks
		int			rawlen;		// Number of records in rawbuf
		int			overflow;	// true iff IR raw code too long
};
```

The most important in this structure is value, which is an unsigned long.

If we want to put the result into a variable for further use:

```
irreceived = results.value;
```

the variable irreceived must be unsigned long, otherwise we will loose some information!
(For RC5 with 14 bits. Word would be enough, but for RC6 with 20 bits, we would lose the higher bits)

RC5 code

This code is mainly used by Philips.

https://en.wikipedia.org/wiki/RC-5

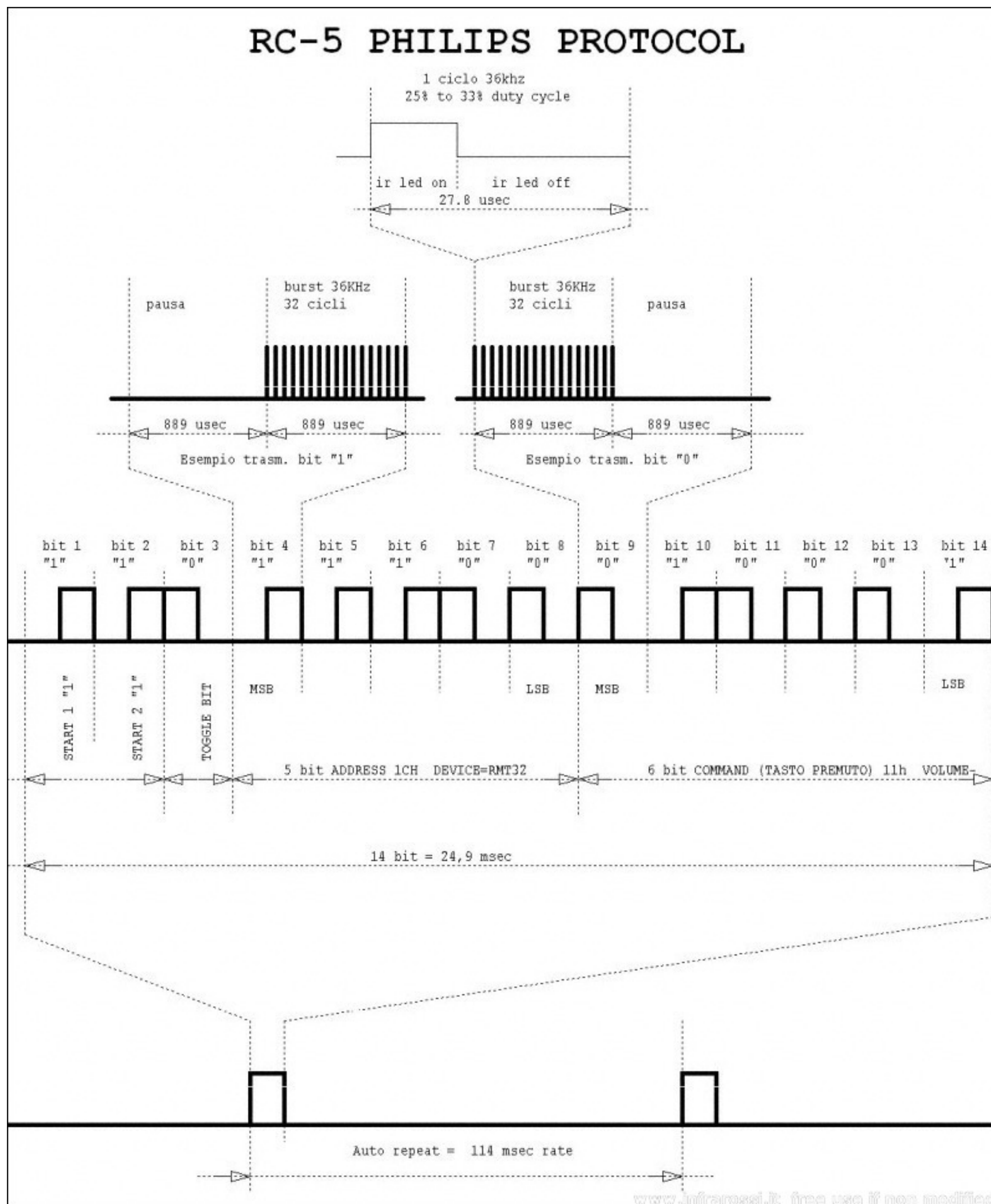
From Wikipedia:

„The command comprises 14 bits:

- A start bit, which is always logic 1 and allows the receiving IC to set the proper gain.
- A field bit, which denotes whether the command sent is in the lower field (logic 1 = 0 to 63 decimal) or the upper field (logic 0 = 64 to 127 decimal). Originally a second start bit, the field bit was added later by Philips when it was realized that 64 commands per device were insufficient. Many devices still use the original system.
- A control bit, which toggles with each button press. This allows the receiving device to distinguish between two successive button presses (such as "1", "1" for "11") as opposed to the user simply holding down the button and the repeating commands being interrupted by a person walking by, for example.
- A five-bit system address, that selects one of 32 possible systems.

- A six-bit command, that (in conjunction with the field bit) represents one of the 128 possible RC-5 commands.

The 36 kHz carrier frequency was chosen to render the system immune to interference from TV scan lines. Since the repetition of the 36 kHz carrier is 27.778 µs and the duty factor is 25%, the carrier pulse duration is 6.944 µs. Each bit of the RC-5 code word contains 32 carrier pulses, and an equal duration of silence, so the bit time is 64×27.778 µs = 1.778 ms, and the 14 symbols (bits) of a complete RC-5 code word take 24.889 ms to transmit. The code word is repeated every 113.778 ms (4096 / 36 kHz) as long as a key remains pressed. (Again, please note that these timings are not strictly followed by all manufacturers, due to a lack of widespread distribution of accurate information on the RC-5 protocol.)“



Source: Wikipedia

If we are just interested in the command bits, we can mask the rest and use only the 6 LSB bytes and mask the rest:

```
...
if (irrecv.decode(&results)) {
    irreceived = results.value;
    command = irreceived & B111111;
```

```
...
```

whith the new variables

```
unsigned long irreceived;
byte command;
```

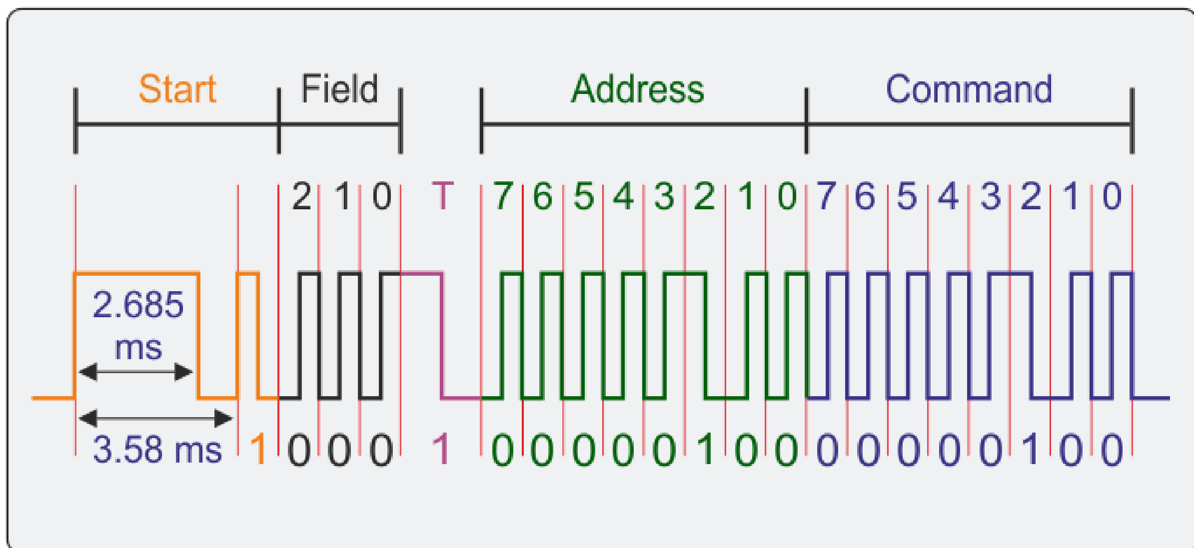
declared at the beginning of the program.

RC6 code

I was convinced that the Philips standard was RC5, until I used the IRrecvdump.ino example program included in the library. This showed me that the remote command I was experimenting with used the RC6 code, not RC5.

This code contains 20 bits of information, 8 of them for the address and 8 for the command. There seem to be more than one standard for RC6.

From http://www.pcbheaven.com/userpages/The_Philips_RC6_Protocol/:



Modulation type: Manchester code (bi-phase) - HIGH to LOW for ACE (1), LOW to HIGH for ZERO (0)
Carrier frequency: 36 KHz
Start bits: 1+1
Field bits: 3
Toggle bits: 1
Address bits: 8
Command bits: 8
Bit period (transmission clock): 895 uSec
Total signal duration: 23.27 mSec
Signal repetition interval: 83 mSec

Other codes

If in doubt which code the IR device uses, there is a fine example program program coming with the library: IRrecvdump.ino

This allows us to get a lot of information via the serial monitor

```
Decoded RC6: 10010 (20 bits)
Raw (40): 2750 -750 550 -750 600 -300 550 -350 1400 -1200 550 -350 550 -300 550
-350 550 -300 550 -350 550 -300 550 -350 550 -300 550 -350 550 -300 1000 -750
550 -350 550 -300 550 -350 550
9832D814

Unknown encoding: 9832D814 (32 bits)
Raw (24): 1800 -1700 950 -800 1850 -750 1000 -750 1000 -750 1000 -750 1000 -750
1000 -750 1000 -1650 950 -800 950 -800 1000
```

Using the WEMOS board with an ESP 8266 chip

There is a library for this chip: IRremoteESP8266.
It comes with some interesting examples, like IrrecvDumpV2.ino, that gives a lot more information than the original IRrecdump.ino program:

```
Timestamp : 000466.464
Encoding  : RC5
Code      : 942 (12 bits)
Library   : v2.3.0

Raw Timing[19]:
   +   988, -   796,     + 1000, -   784,     + 1920, -   760,     +   988, -  1696,
   +  1872, -  1696,     + 1918, -   764,     +  984, -   790,     +  1034, -   758,
   +   986, -  1698,     + 1882

uint16_t rawData[19] = {988, 796,  1000, 784,  1920, 760,  988, 1696,  1872, 1696,  1918, 764,  984,
790,  1034, 758,  986, 1698,  1882};  // RC5 942
uint32_t address = 0x5;
uint32_t command = 0x2;
uint64_t data = 0x942;
```

The ESP8266 works with 3.3V logic!
The IR receiver TSOP1736 however needs a supply voltage of +5V, it doesn't work with 3.3V.
I connected a 3V3 zener diode in parallel to the output of the TSOP1736, to protect the inputs of the controller (a resistance can be omitted as the internal pullup of the TSOP1736 has a high value of 80kΩ, so the current is very weak. Even too weak maybe, as the high level in this configuration was only 1.7V, a value just sufficient to be interpreted as logical 1).

For safer operation I added an external pullup resistor of 1 to 4.7kΩ that lifted the high level up to a value between 2.2V And 2.5V.