Part 2

lesson 13

IR Receiver Module

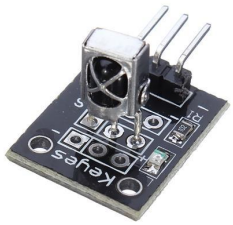
Overview

Using an IR Remote is a great way to have wireless control of your project.

Infrared remotes are simple and easy to use. In this tutorial we will be connecting the IR receiver to the ESP32, and then use a Library that was designed for this particular sensor.

In our sketch we will have all the IR Hexadecimal codes that are available on this remote, we will also detect if the code was recognized and also if we are holding down a key.

Component Required:



(1) x Elegoo ESP32

(1) x IR receiver module

(1) x IR remote

(3) x F-M wires (Female to Male DuPont wires)

Component Introduction

IR RECEIVER SENSOR:

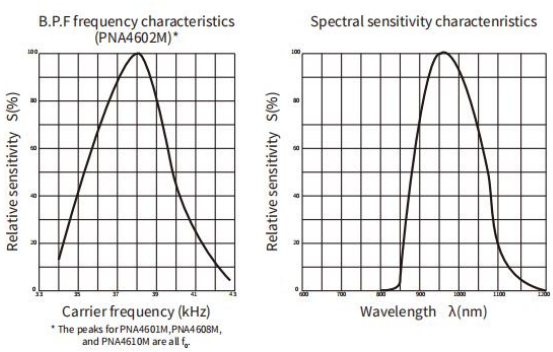
IR detectors are little microchips with a photocell that are tuned to listen to infrared light. They are almost always used for remote control detection - every TV and DVD player has one of these in the front to listen for the IR signal from the clicker. Inside the remote control is a matching IR LED, which emits IR pulses to tell the TV to turn on, off or change channels. IR light is not visible to the human eye, which means it takes a little more work to test a setup.

There are a few difference between these and say a CdS Photocells:
IR detectors are specially filtered for IR light, they are not good at detecting visible light. On the other hand, photocells are good at detecting yellow/green visible light, and are not good at IR light.

IR detectors have a demodulator inside that looks for modulated IR at 38 KHz. Just shining an IR LED won't be detected, it has to be PWM blinking at 38KHz. Photocells do not have any sort of demodulator and can detect any frequency (including DC) within the response speed of the photocell (which is about 1KHz)

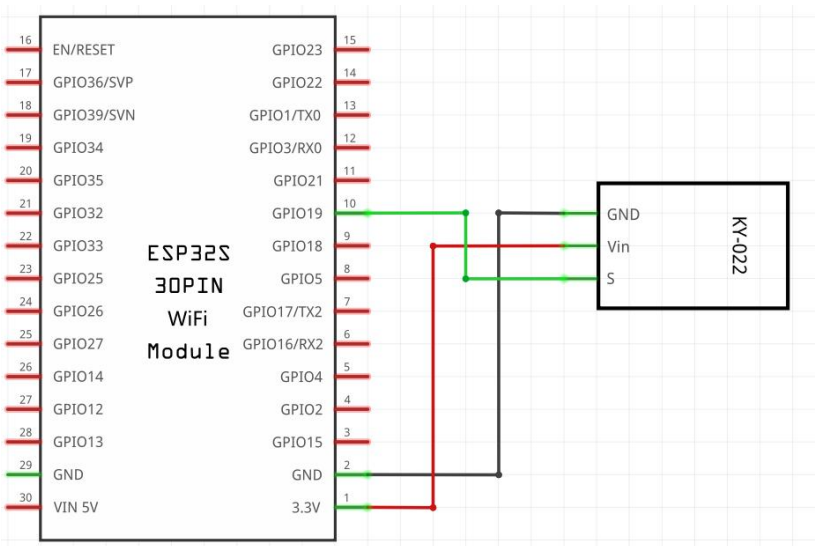
IR detectors are digital out - either they detect 38KHz IR signal and output low (0V) or they do not detect any and output high (5V). Photocells act like resistors, the resistance changes depending on how much light they are exposed to.

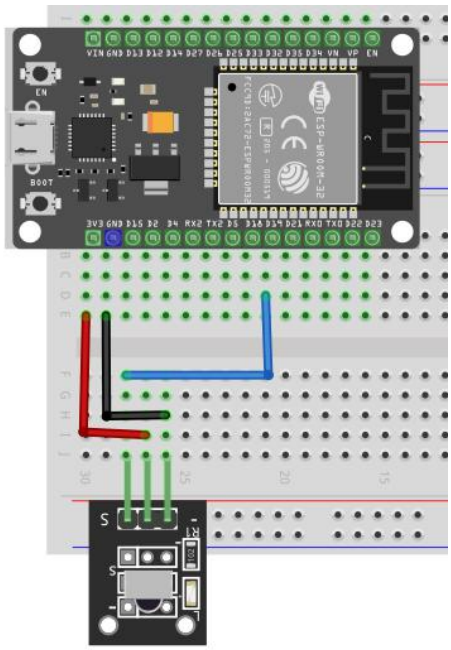
As you can see from these datasheet graphs, the peak frequency detection is at 38 KHz and the peak LED color is 940 nm. You can use from about 35 KHz to 41 KHz but the sensitivity will drop off so that it won't detect as well from afar. Likewise, you can use 850 to 1100 nm LEDs but they won't work as well as 900 to 1000nm so make sure to get matching LEDs! Check the datasheet for your IR LED to verify the wavelength.



Try to get a 940nm - remember that 940nm is not visible light!

Connection Schematic





There are 3 connections to the IR Receiver.
The connections are: Signal, Voltage and Ground.
The“G”is the Ground,“Y”is signal, and“R”is Voltage 3.3V.

Wiring diagram

Code

After wiring, please open the program in the code folder- **IR_Receiver_Module** and click UPLOAD to upload the program. See Lesson 5 in part 1 for details about program uploading if there are any errors.

Non-blocking Pulse Measurement: The readPulse() function uses micros() (microsecond precision) to measure logic level durations, adapting to NEC's strict timing requirements (tolerance: ±20% for pulse durations).

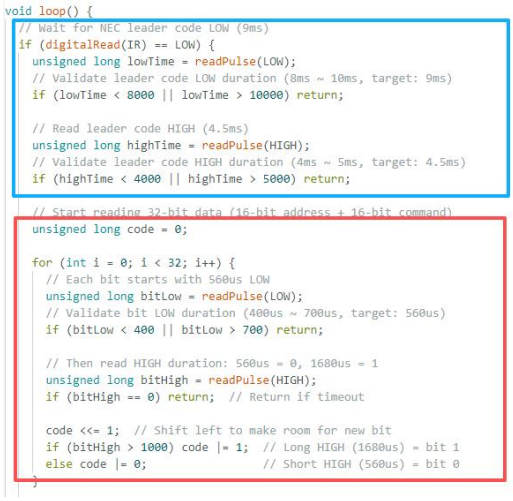
The readPulse() function is the foundation of NEC protocol decoding

```
unsigned long readPulse(int level, unsigned long timeout = 20000) {
  unsigned long start = micros();  // Record start time in microseconds
  // Wait until level changes or timeout
  while (digitalRead(IR) == level) {
    if (micros() - start > timeout) return 0;  // Return 0 on timeout
  }
  return micros() - start;  // Return the actual duration of the level
}
```

Wait for NEC Leader Code and Read 32-bit Data

The code in the blue box is mainly used to detect and validate the NEC protocol leader code —the synchronization signal that marks the beginning of an infrared frame.

The code in the red box is responsible for decoding the 32-bit NEC data frame, which consists of a 16-bit device address and a 16-bit command/button code.



```
void loop() {
  // Wait for NEC leader code LOW (9ms)
  if (digitalRead(IR) == LOW) {
    unsigned long lowTime = readPulse(LOW);
    // Validate leader code LOW duration (8ms ~ 10ms, target: 9ms)
    if (lowTime < 8000 || lowTime > 10000) return;

    // Read leader code HIGH (4.5ms)
    unsigned long highTime = readPulse(HIGH);
    // Validate leader code HIGH duration (4ms ~ 5ms, target: 4.5ms)
    if (highTime < 4000 || highTime > 5000) return;

    // Start reading 32-bit data (16-bit address + 16-bit command)
    unsigned long code = 0;

    for (int i = 0; i < 32; i++) {
      // Each bit starts with 560us LOW
      unsigned long bitLow = readPulse(LOW);
      // Validate bit LOW duration (400us ~ 700us, target: 560us)
      if (bitLow < 400 || bitLow > 700) return;

      // Then read HIGH duration: 560us = 0, 1680us = 1
      unsigned long bitHigh = readPulse(HIGH);
      if (bitHigh == 0) return;  // Return if timeout

      code <<= 1;  // Shift left to make room for new bit
      if (bitHigh > 1000) code |= 1;  // Long HIGH (1680us) = bit 1
      else code |= 0;                 // Short HIGH (560us) = bit 0
    }
```

Output the decoded 32-bit NEC code in hexadecimal format to the serial monitor.Hexadecimal is the standard format for representing infrared remote codes (each button has a unique hex code).

```
Serial.print("Received code: 0x");
Serial.println(code, HEX);
```

As shown in the figure on the right, its main function is to map the decoded hexadecimal code to a human-readable button name (e.g., "POWER", "VOL+") for easier interpretation.

```
// Match code to corresponding button
switch (code) {
  case 0xFFA25D: Serial.println("POWER"); break;
  case 0xFFE21D: Serial.println("FUNC/STOP"); break;
  case 0xFF629D: Serial.println("VOL+"); break;
  case 0xFF22DD: Serial.println("FAST BACK"); break;
  case 0xFF02FD: Serial.println("PAUSE"); break;
  case 0xFFC23D: Serial.println("FAST FORWARD"); break;
  case 0xFFE01F: Serial.println("DOWN"); break;
  case 0xFFA857: Serial.println("VOL-"); break;
  case 0xFF906F: Serial.println("UP"); break;
  case 0xFF9867: Serial.println("EQ"); break;
  case 0xFFB04F: Serial.println("ST/REPT"); break;
  case 0xFF6897: Serial.println("0"); break;
  case 0xFF30CF: Serial.println("1"); break;
  case 0xFF18E7: Serial.println("2"); break;
  case 0xFF7A85: Serial.println("3"); break;
  case 0xFF10EF: Serial.println("4"); break;
  case 0xFF38C7: Serial.println("5"); break;
  case 0xFF5AA5: Serial.println("6"); break;
  case 0xFF42BD: Serial.println("7"); break;
  case 0xFF4AB5: Serial.println("8"); break;
  case 0xFF52AD: Serial.println("9"); break;
  case 0xFFFFFFFF: Serial.println("REPEAT"); break;
  default: Serial.println("OTHER BUTTON"); break;
}

delay(150);  // Debounce delay to avoid duplicate readings
```

switch...case

[Control Structure]

Description

Like if statements, switch case controls the flow of programs by allowing programmers to specify different code that should be executed in various conditions. In particular, a switch statement compares the value of a variable to the values specified in case statements. When a case statement is found whose value matches that of the variable, the code in that case statement is run

The break keyword exits the switch statement, and is typically used at the end of each case. Without a break statement, the switch statement will continue executing the following expressions ("falling-through") until a break, or the end of the switch statement is reached.