

IRsmallDecoder

A small, fast and reliable infrared signals decoder, for controlling Arduino projects with remote controls.

This is a library for receiving and decoding IR signals from remote controls. It's ideal for Arduino projects that require a fast, simple, and reliable decoder, but do not need to handle multiple protocols simultaneously or transmit IR signals.

Table of Contents

- [Main features](#)
- [Supported protocols](#)
- [Supported boards](#)
- [Connecting the IR receiver](#)
 - [Arduino boards](#)
 - [ESP boards](#)
 - [Pin Mapping Considerations](#)
- [Installing the library](#)
- [Using the library](#)
 - [With a standard Arduino board](#)
 - [With ESP boards](#)
 - [Address check](#)
 - [Methods and data](#)
- [Possible improvements](#)
- [Contributions](#)
- [Contact information](#)
- [License](#)
- [Appendix A - Details about this library](#)
- [Appendix B - IR receiver connection details](#)

Main features

- The signals are fully decoded and the data is divided into separate variables.
- Initial repetition codes are ignored, effectively reducing the chance of detecting multiple codes when only one is expected.
- Held buttons are detected and processed in a more practical and useful way.
- Signal tolerances are generous, allowing for a high detection rate without compromising reliability.
- Redundant signal data (when present) is used solely for error checking.

- Very low SRAM and Flash memory usage.
- Decoding is performed asynchronously without requiring timers, allowing them to be used for other purposes.
- No conflicts with timer-related functionalities such as *tone()*, servos, *analogWrite()*, etc.
- Signal acquisition and processing are performed on-the-fly via a hardware (external) interrupt.
- Compatible with a wide range of Arduino and non-Arduino boards, including ATtiny, ESP8266, and ESP32.

Supported protocols

- NEC
- NECx
- Philips RC5 and RC5x (simultaneously)
- Sony SIRC 12, 15 and 20 bits (individually or simultaneously)
- SAMSUNG old standard
- SAMSUNG 32 bits (16 of which are for error detection)

Supported boards

Because it uses no hardware-specific instructions, this library should work on most Arduino boards and many other microcontrollers that support Arduino libraries.

I've only tested it thoroughly on an Arduino Uno, an Arduino Mega, an Arduino Nano ESP32, a NodeMCU ESP8266, and a NodeMCU ESP32.

ATtiny 25/45/85/24/44/84 microcontrollers are also supported.

If you experience issues using this library on a particular board, please submit an issue [here](#) or [contact me](#).

Connecting the IR receiver

Arduino boards

The receiver's output must be connected to one of the Arduino's digital pins that support external interrupts — and it must also work with the CHANGE mode if the intended protocol requires it. One example of a board that lacks CHANGE mode on some interrupt pins is the Arduino 101, and one protocol that requires that mode is RC5. You can check the required interrupt mode for each protocol [here](#).

The following table (adapted from the [Arduino Reference](#)) lists the digital pins that can be used to connect an IR receiver to an Arduino board:

Board or microcontroller	Digital pins usable for interrupts
UNO R3, Nano, Mini, other 328-based	2, 3
UNO R4 Minima, UNO R4 WiFi	2, 3
Uno WiFi Rev.2, Nano Every	all digital pins
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21 ^[1]
Micro, Leonardo, other 32u4-based	0, 1, 2, 3, 7
Zero	all digital pins, except pin 4
MKR Family boards	0, 1, 4, 5, 6, 7, 8, 9, A1, A2
Nano 33 IoT	2, 3, 9, 10, 11, 13, A1, A5, A7
Nano 33 BLE, Nano 33 BLE Sense (Rev 1 & 2)	all pins
Nano RP2040 Connect	0-13, A0-A5
Nano ESP32	all pins ^[2]
GIGA R1 WiFi	all pins
Due	all digital pins
101 with CHANGE mode	2, 5, 7, 8, 10, 11, 12, 13
101 with other modes	all digital pins
ATtiny 25/45/85	2 ^[3]
ATtiny 24/44/84	8 ^[3]

1. In the Mega family, pins 20 and 21 cannot be used for interrupts when they are configured for I2C communication.

2. If you connect the receiver to D13, the board's built-in LED will remain slightly lit (depending on the receiver's pull-up resistor) and flicker when an IR signal is detected.

3. Assuming you're using [damellis' ATtiny core](#) or [SpenceKonde's ATtinyCore](#). Other cores may have different pin assignments.

ESP boards

This library also works on ESP8266 and ESP32-based boards (in addition to the Arduino Nano ESP32 mentioned above), but you should be mindful of the pin you connect the IR module to.

ESP8266

The GPIO pins that are safe to use on the ESP8266 NodeMCU are: 4, 5, 12, 13, and 14. The following pins can be used but may have some limitations: 0, 2, 9, and 10. (GPIO pins 0 and 2 can cause boot failure if they are pulled LOW during startup and pins 2, 9, and 10 are HIGH during boot).

ESP 32

The GPIO pins that are safe to use on the ESP32 NodeMCU are: 4, 5, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 25, 26, 27, 32, and 33. The following pins can be used but may have some limitations: 12, 34, 35, 36, and 39. (GPIO pin 12 may cause a boot failure and the last 4 do not have pull up resistors, though they may still work if the IR module already has one.)

Pin Mapping Considerations

On most ESP32 and ESP8266 boards, the pin you specify corresponds directly to the microcontroller's GPIO number. However, when using an Arduino Nano ESP32 with the default pin numbering, the specified pin does not match the actual GPIO. For example, pin 5 (or D5) maps to GPIO 8. ([Click here](#) for more information).

On some boards, such as the ESP8266 NodeMCU, you can also refer to pins using their digital labels — for example, GPIO 5 is labeled as D1.

Once you've figured out the correct pin to use, you can refer to the [IR receiver connection details](#) at the end of this document if you're unsure how to physically wire the IR receiver to the board.

Installing the library

With the Library Manager

- Run Arduino IDE and go to *tools > Manage Libraries...* or *Sketch > Include Library > Manage Libraries...*;
- Search for IRsmallDecoder and install.

Manually

- Navigate to the [Releases page](#);
- Download the latest release (zip file);
- Run Arduino IDE and go to *Sketch > Include Library > Add .ZIP Library*;
- Or, instead of using Arduino IDE, extract the zip file and move the extracted folder to your libraries directory.

Using the library

This library can be initialized in a few different ways, and the following "standard" way will not work with ESP8266 / ESP32 based boards. For those microcontrollers, jump ahead to the [With ESP boards](#) section.

With a standard Arduino board

In the INO file, **one** of the following directives must be used:

```
#define IR_SMALLD_NEC
#define IR_SMALLD_NECx
#define IR_SMALLD_RC5
```



```

#define IR_SMALLD_SIRC12
#define IR_SMALLD_SIRC15
#define IR_SMALLD_SIRC20
#define IR_SMALLD_SIRC
#define IR_SMALLD_SAMSUNG
#define IR_SMALLD_SAMSUNG32

```

before the

```
#include <IRsmallDecoder.h>
```

Then you need to create **one** decoder object with the correct digital pin:

```
IRsmallDecoder irDecoder(2); //IR receiver connected to pin 2 in this example
```

And also a decoder data structure:

```
irSmallD_t irData;
```

Inside the loop(), check if the decoder has new data available. If so, do something with it:

```

void loop() {
  if(irDecoder.dataAvailable(irData)) {
    Serial.println(irData.cmd, HEX);
  }
}

```

A full example:

```

#define IR_SMALLD_NEC
#include <IRsmallDecoder.h>
IRsmallDecoder irDecoder(2);
irSmallD_t irData;

void setup() {
  Serial.begin(115200);
  Serial.println("Waiting for a NEC remote control IR signal...");
  Serial.println("held \t addr \t cmd");
}

void loop() {
  if(irDecoder.dataAvailable(irData)) {
    Serial.print(irData.keyHeld);
    Serial.print("\t ");
    Serial.print(irData.addr, HEX);
    Serial.print("\t ");
    Serial.println(irData.cmd, HEX);
  }
}

```


With ESP boards

If you're using an ESP8266 or an ESP32 board, the decoder cannot be initialized before the `setup()` function, because it depends on hardware resources that are not initialized until that function is executed. One workaround is to declare the decoder as a static variable inside the `loop()` function; alternatively, you can dynamically allocate it inside `setup()`.

The static way

```
// Define which protocol you want:
#define IR_SMALLD_NEC
// Include the library:
#include <IRsmallDecoder.h>

void setup() {
  Serial.begin(115200);
  while (!Serial);
  Serial.println();
  Serial.println("Waiting for a remote control IR signal...");
  Serial.println("held\tAddr\tCmd");
}

void loop() {
  // Create one static decoder "listening" on a supported GPIO pin:
  static IRsmallDecoder irDecoder(5);
  // Create a decoder data structure:
  irSmallD_t irData;
  // Check if a signal was decoded:
  if (irDecoder.dataAvailable(irData)) {
    // Use the decoded data:
    Serial.print("» ");
    Serial.print(irData.keyHeld);
    Serial.print("\t");
    Serial.print(irData.addr, HEX);
    Serial.print("\t");
    Serial.println(irData.cmd, HEX);
  }
}
```

The Dynamic way

```
// Define which protocol you want:
#define IR_SMALLD_NEC
// Include the library:
#include <IRsmallDecoder.h>
// Declare a pointer to the decoder:
IRsmallDecoder* irDecoder;
// Create a decoder data structure:
irSmallD_t irData;

void setup() {
  Serial.begin(115200);
  while (!Serial);
  Serial.println();
  Serial.println("Waiting for a remote control IR signal...");
}
```



```

Serial.println("held\tAddr\tCmd");
// Dynamically allocate the decoder, using a supported GPIO pin:
irDecoder = new IRsmallDecoder(5);
}

void loop() {
// Check if a signal was decoded (using arrow operator):
if (irDecoder->dataAvailable(irData)) {
// Use the decoded data:
Serial.print("» ");
Serial.print(irData.keyHeld);
Serial.print("\t");
Serial.print(irData.addr, HEX);
Serial.print("\t");
Serial.println(irData.cmd, HEX);
}
}

```

Other examples

I've included an example in this library — called [ESP_IR_Decoder](#) — designed specifically for ESP8266 and ESP32 microcontrollers. It uses the static approach and is set up to test any of the supported protocols.

As for the other examples, you can also run them on ESP-based MCUs, but you'll need to make a few adjustments. The simplest way to make them work is by moving the global declaration of the decoder into the beginning of the `loop()` function and declaring it as static. The decoder data structure can remain as a global variable.

Address check

if you are using multiple remotes, with the same protocol, but different addresses, in the vicinity of your project, you should also verify the address. For example, you can do something like this:

```

if (irData.addr == theRightAddr) {
switch (irData.cmd) {
case someCmd:
// do something here
break;
case someOtherCmd:
// do some other things
break;
//etc.
}
}

```

Methods and data

The multifunctional *dataAvailable()* method

The `dataAvailable(irData)` method combines the functionality of 3 "fictitious" functions: *isDataAvailable()*, *getData()* and *setDataUnavailable()*.

If there is some data available — already decoded — when `irDecoder.dataAvailable(irData)` is called:

- The data is copied to the specified data structure (`irData` in this example);
- The original data is marked as unavailable;
- And, finally, it returns `true` .

If there's no new data, it simply returns `false` .

Note: this library does not use data buffering. If a new signal is decoded before the available data is retrieved, the previous data is discarded. This may happen if the loop takes too long to check for new data. So, if you want to use repetition codes, try to keep the loop duration below 100ms (for NEC and RC5) and avoid using delays. Despite the fact that they don't interfere with decoding, their use is discouraged.

If you just want to check if any button was pressed and don't care about the data, you can call the `dataAvailable()` method without any parameters. Keep in mind that, if there's new data available, this method will discard it, before returning `true` . The [ToggleLED](#) example demonstrates this functionality.

Disabling the decoder

If you have a time-critical function, that should not be frequently interrupted, you can disable the decoder before calling that function using the `disable()` method. After the function completes, re-enable the decoder by calling the `enable()` method.

You can also use it to disable the decoder when it's not needed, as shown in the [TemporaryDisable](#) example.

The `enable()` method also works as a reset method. This additional functionality may be useful if you chose not to use the [timeout](#) feature, and need to temporarily disable all interrupts (or are using a library that does that). Just call the `enable()` method after re-enabling all interrupts, even if the decoder is already enabled. This will reset the decoder and prevent the next IR command from being discarded (in case the receiver detected a signal while interrupts were disabled, resulting in a pending interrupt).

Protocol data structures

The protocol data structure is not the same for all protocols, but they all have two common member variables:

- **cmd** - the button command code (one byte);
- **addr** - the address code (usually the same for all buttons on a single remote).

Most decoders also have the **keyHeld** variable (which is set to `true` when a button is being held), and two of the SIRC decoders include the **ext** variable (see [notes](#) for more details).

The following table shows the number of bits used by each protocol and the data types of the structure's member variables:

Protocol	keyHeld	cmd	addr	ext
NEC	bool	8/uint8_t	8/uint8_t	--
NECx	bool	8/uint8_t	16/uint16_t	--
RC5	bool	7/uint8_t	5/uint8_t	--
SIRC12	--	7/uint8_t	5/uint8_t	--
SIRC15	--	7/uint8_t	8/uint8_t	--
SIRC20	--	7/uint8_t	5/uint8_t	8/uint8_t
SIRC	bool	7/uint8_t	8/uint8_t	8/uint8_t
SAMSUNG	bool	8/uint8_t	12/uint16_t	--
SAMSUNG32	bool	8/uint8_t	8/uint8_t	--

Notes

- Only one protocol can be compiled at a time, however:
 - NECx also decodes NEC, but the address will contain redundant data;
 - The RC5 implementation also decodes the extended protocol version, which includes a field bit that is used as an extra command bit (resulting in 7 bits total);
 - SIRC12 will detect signals from SIRC15 and SIRC20, but the decoded codes will not be correct;
 - Similarly, SIRC15 will detect signals from SIRC20, but not from SIRC12.
- SIRC handles 12-, 15-, or 20-bit signals, by taking advantage of the fact that most Sony remotes send three frames per button press. It uses triple-frame verification, checks for held keys, and ignores initial repetition codes;
- SIRC12, SIRC15 and SIRC20 use a basic (slightly smaller and faster) implementation, without the triple frame verification and without the **keyHeld** check.
- The SIRC20 protocol has an **ext** variable which holds extended data.
- The SIRC decoder also has an **ext** variable, but it's only used for 20-bit codes — otherwise, it is set to 0.
- Unlike the other decoders, the RC5 decoder is unable to handle closely spaced signals. If you press a button multiple times in a short period, it may interpret them as a single invalid signal.

Possible improvements

- I might add a few more IR protocols to this library (there are a lot of them out there);
- The keyHeld initial delay is hard-coded — I could make it configurable (via constructor) or even changeable (via method);
- I believe it may be possible to increase the number of usable pins by using NicoHood's PinChangeInterrupt Library;
- SIRC12, SIRC15 and SIRC20 do not include the keyHeld feature. The SIRC decoder fills that gap, but requires three signal frames per key press;
- The SIRC decoder could also return the number of detected bits (12, 15 or 20).

Contributions

So far, these releases have been made without significant contributions from other developers. However, I must say that this work was inspired by several existing IR libraries: [Arduino-IRremote](#), [IRLib2](#), [IRReadOnlyRemote](#), [Infrared4Arduino](#) and especially [IRLremote](#), which was almost what I was looking for — but not quite. So I decided to create my own NEC protocol decoder, and it worked so well that I decided to publish it, believing others might find it useful. But not before implementing a few more decoders and packaging everything into an Arduino-compatible library.

In addition, I should thank corvin78 for helping with testing on a Digispark (ATtiny85) and Kristof Toth for making me realize the importance of the [timeout](#) feature.

Contact information

If you wish to report an issue related to this library (and don't want to do it on GitHub) you may send an e-mail to: lumica@outlook.com. Suggestions and comments are also welcome.

License

Copyright (c) 2020 Luis Carvalho
This library is licensed under the MIT license.
See the [LICENSE file](#) for details.

Appendix A - Details about this library

Size

The size of this library is, as the name implies, small (about 940 bytes on average, for the Arduino UNO board) and the memory usage is also reduced (around 30 bytes). Keep in mind that these values vary depending on the selected protocol and the selected board.

Program memory and static data used (in SRAM) on an Arduino UNO (in bytes)*:

Protocol	Program memory	Static data
NEC	918	29
NECx	914	31
RC5	1140	33
SIRC12	768	23
SIRC15	744	23
SIRC20	828	27
SIRC	1324	38
SAMSUNG	940	30
SAMSUNG32	914	30

[*] - If you disable the timeout functionality, you'll save 58 bytes of program memory and around 1.3µs per loop cycle (on an Arduino UNO @16MHz). It's negligible, but if you really need a few extra bytes and don't mind the occasional discarded codes, you have the option to do so. More information can be found in the [Timeout](#) section of this document.

To keep track of this library's memory usage, I created a couple of test sketches based on the ToggleLED example — one version without the library (serving as the baseline), and another with the library. By compiling each of the supported protocols and comparing the resulting memory sizes to the reference sketch, I was able to determine how much program memory and RAM each protocol uses.

For example, compiling the NEC protocol on an Arduino UNO R3 yielded the following results:

- The reference sketch (without the library) uses 766 bytes of program memory and 11 bytes of RAM.
- With the NEC protocol decoder included, the sketch uses 1684 bytes of program memory and 40 bytes of RAM.

That's an increase of 918 bytes in flash and 29 bytes in RAM for the NEC implementation.

Here are the two sketches used for this comparison:

Reference sketch	With NEC protocol decoder
<pre> // #define IR_SMALLD_NEC // #include <IRsmallDecoder.h> // IRsmallDecoder irDecoder(2); // irSmallD_t irData; int ledState=LOW; void setup() { pinMode(LED_BUILTIN, OUTPUT); } void loop() { //if(irDecoder.dataAvailable(irData)){ ledState=(ledState==LOW)? HIGH:LOW; digitalWrite(LED_BUILTIN,ledState); //} } </pre>	<pre> #define IR_SMALLD_NEC #include <IRsmallDecoder.h> IRsmallDecoder irDecoder(2); irSmallD_t irData; int ledState=LOW; void setup() { pinMode(LED_BUILTIN, OUTPUT); } void loop() { if(irDecoder.dataAvailable(irData)){ ledState=(ledState==LOW)? HIGH:LOW; digitalWrite(LED_BUILTIN,ledState); } } </pre>

Speed

Although my main goals are functionality and small size, I believe this library is reasonably fast. I haven't compared it to other libraries (it's not easy to do so), but I was able to compare the speed of the different protocols that I've implemented so far:

Protocol Speed comparisons:

Protocol	Interrupt Mode	Avg. Interrupt Time	Max. Interrupt Time	Interrupts per Keypress	Signal Duration
NEC	RISING	11.33 μ s	14 μ s	34	67.5ms
NECx	RISING	10.92 μ s	13 μ s	34	67.5ms
RC5	CHANGE	10.99 μ s	19 μ s	14 to 28	24.9ms
SIRC12	RISING	10.04 μ s	14 μ s	3*13	3*(17.4 to 24.6)ms
SIRC15	RISING	10.50 μ s	12 μ s	3*16	3*(21 to 30)ms
SIRC20	RISING	11.10 μ s	15 μ s	3*21	3*(27 to 39)ms
SIRC	RISING	11.75 μ s	18 μ s	39, 48 or 63	3*(17.4 to 39)ms
SAMSUNG	FALLING	10.98 μ s	13 μ s	2*22	2*(32.1 to 54.6)ms
SAMSUNG32	FALLING	10.97 μ s	14 μ s	34	(54.6 to 72.6)ms

Notes:

- Signal Duration refers to the effective signal duration, not the signal period;
- Tests were conducted on an Arduino Uno @ 16MHz with the [timeout](#) feature enabled;
- With the timeout disabled, the average and maximum values are slightly lower, but the difference is insignificant;
- To get the number of clock cycles used by an interrupt, multiply the time (in μ s) by 16;

- The decoding is done partially while the signal is being received. Once a signal is fully received, the final stage of decoding is executed, and that's when the interrupt takes more time to run.

Unwanted initial repetition codes

Remote control keys do not "bounce", but the remotes do tend to send more codes than desired when a button is pressed. That's because, after a very short interval, they start sending repeat codes. To avoid those unwanted initial repetitions, this library ignores a few of those repetition codes before confirming that the button is actually being held.

Data separation

The data sent by remotes is decoded according to the protocol specifications and separated into different variables. On most remotes, only the 8-bit command matters, so you don't have to work with 16- or 32-bit codes. This reduces both code size and memory usage.

Simplicity

As you've probably seen above — or if you've already tried one of the 'Hello...' examples — this library is very simple to use and not overloaded with rarely needed features. That simplicity is one of the reasons it's compact and resource-efficient, though careful design and optimization also contribute to its small size and speed. Additional features may be added in the future, but only if they do not significantly impact performance or memory usage.

How it works

The decoding is done asynchronously, which means that it doesn't rely on a timer to receive and process signals. Instead, it uses a hardware interrupt to drive the Finite State Machines that perform the decoding. In fact, these are Statechart Machines (David Harel type) operating in an asynchronous mode.

Most of the Statechart Machines are implemented using *switch* statements, but I also use the "labels as values" GCC extension (also known as "computed gotos") to implement some of the more complex statecharts. It's not part of C++ standard, but it should work with all IDEs that use the GCC (such as the Arduino IDE). If you encounter problems compiling any of the protocols that use the "labels as values" extension, please submit an issue [here](#) or [contact me](#).

I can't say these decoders are easy to understand — some of the Statechart Machines turned out to be quite tricky. But if you're still interested in taking a look at the statechart diagrams, they can be found [here](#). Note that they may not be exact representations of what I've actually implemented, but they're a good starting point.

Timeout

All decoders have a timeout feature that resets them after a few milliseconds without receiving IR signals. This prevents them from remaining in a waiting state due to the detection of extra pulses caused by interference or the detection of other unsupported IR signals in the vicinity. Without this mechanism, the decoder might occasionally fail to detect valid IR signals.

This feature can be disabled, by including the line `#define IR_SMALLD_NO_TIMEOUT` before the `#include <IRsmallDecoder.h>` directive. This saves you a few bytes of program memory and some CPU cycles per loop. The savings are almost negligible, but the option is available in case someone needs all the extra program space and doesn't mind occasional decoding failures.

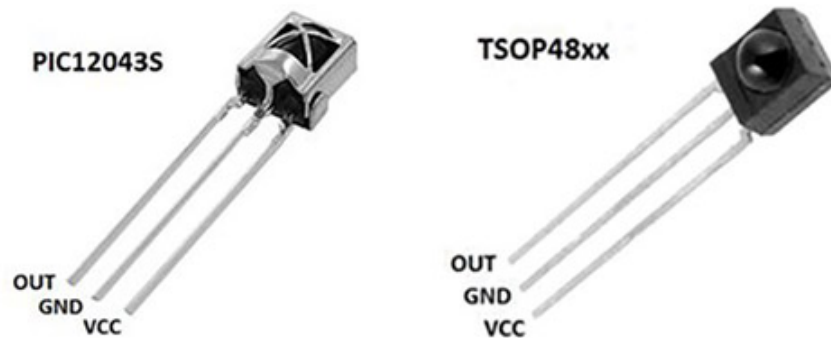
It's worth noting that this timeout does not use a timer. It's polling-based, relying entirely on calls to the `dataAvailable()` method in the `loop()` function.

No hardware specific instructions

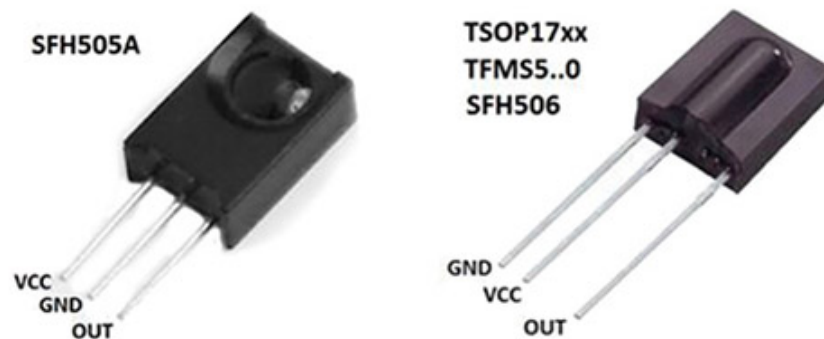
In order to make this library compatible with most Arduino boards, I didn't use any hardware specific instructions. However, I did use a programming technique that assumes the microcontroller's endianness is Little-Endian. On some boards, you may receive a warning related to this, but the code should work regardless.

Appendix B - IR receiver connection details

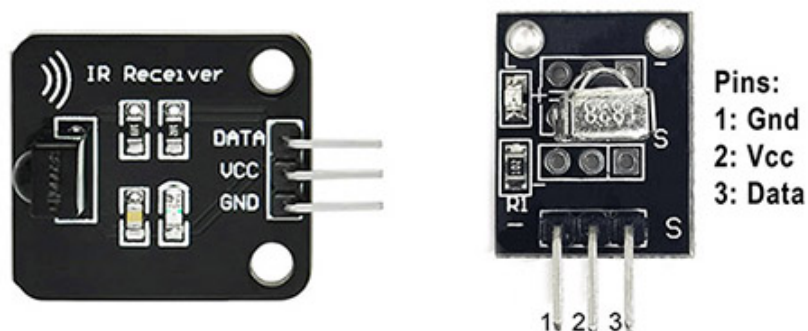
If you are using a simple IR receiver IC, the pinout order will most likely be OUT-GND-VCC, as in the following examples:



But beware, there are other IR receivers with different pinouts, such as the following examples:



If you are using an IR receiver module (designed for prototyping), the pinout is usually labeled on it, and the OUT pin may sometimes be marked as DATA, DAT, or S.

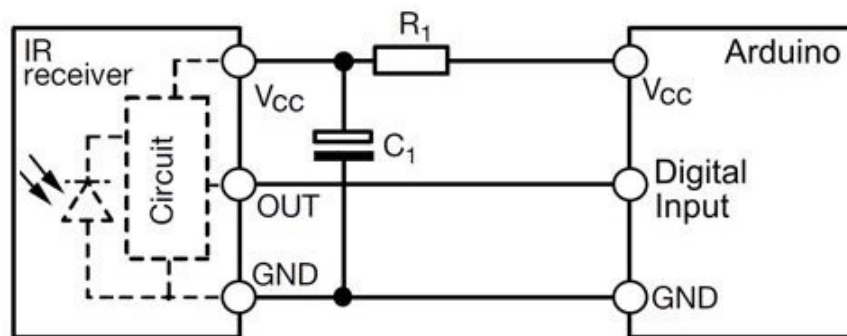


The connection to the Arduino is very straightforward. Just connect:

- OUT (or DAT or S) to one of the Arduino's digital pins that has interrupt capability^[1];
- VCC to the Arduino's +5V or (+3.3V if you are using a board with a lower operating voltage^[2]);
- GND to one of the Arduino's Ground pins.

1. Go to [Connecting the IR receiver](#) for more information.
2. Keep in mind that not all IR receivers can operate at low voltages.

Nearly all IR receiver datasheets recommend using an RC filter (R1 and C1) on the power input, to suppress power supply disturbances. While this improves reliability, it's generally not essential during early development stages, but it's worth including in final designs:



① Notes:

- Many IR receiver prototyping modules already include this RC filter.
- When adding it manually to a simple IR receiver, typical values are a 100Ω resistor and a 4.7μF capacitor, placed as close as possible to the module's power pins.
- Not all datasheets recommend the same values for the RC filter, so it's always best to consult the receiver's documentation for proper component ratings and layout recommendations.