Users Manual for IRLib.
A Library for Receiving, Decoding and Sending Infrared Signals Using Arduino.

(Revised June 2014)

This library is designed for using Arduino controllers for receiving, decoding and sending infrared signals such as those used by infrared remotes on consumer electronics. The work is based on earlier library called IRremote which was created by Ken Shirriff. This library is a major rewrite of that system. The original library defined only decoding and sending classes. The decoder class included the code for receiving the raw signals and measuring time intervals in them.

In this rewrite we separated the receiving from the decoding initially to make the code easier to understand. Our thinking was that receiving portion of the code was extremely hardware oriented and required knowledge of hardware interrupts, hardware times and other technical issues. Whereas the decoding portion of the process dealt with protocols used by IR devices. It was more likely that users want to implement new protocols and they would not want to have to deal with the technical details.

Overall I wanted to take a “black box” approach that would isolate as many of the internal details as possible so that the end-user did not have to deal with them. The separation of receiving and decoding also allowed for the possibility that the receiver could be reset to receive another sequence of signals while the decoding process was going on (although decoding is not generally very time intensive). This design decision to split receiving from decoding also facilitated the later decision in version 1.3 of the library to implement other methods of receiving signals.

This manual is divided into three major sections. First is a complete reference of all the classes, structures and methods included in the library. The reference section is designed for those who want to make maximum use of the facilities of the library. However you may not need to understand everything in this section in order to use it. We suggest that novices proceed to part two which is the tutorials section. There you will find some examples of the basic use of the library. Finally you can move on to the third section of this documentation which explains how to add additional protocols to the decoding and sending sections.

Note also that the code is well documented and skilled programmers who are familiar with infrared protocols and Arduino hardware or anyone interested should simply browse through the code itself for useful information.

1. IRLib Reference

This section is intended to be a complete reference to the classes, methods, and structures used in the library. It is divided into three sections: the receiver classes, the decoder classes, and the sending classes.

1.1 Receiver Classes

IR data is received on an input pin of the Arduino as a square wave of pulses of varying lengths. The length of the on and off pulses encodes the data. It is the job of the receiver class to record the timing of the pulses in the spaces between in an array and pass that information to the decoder class. This section contains an overview discussion of the receiving process, with an explanation of the base receiver class and three derived classes each with its own unique characteristics. In addition to the receiver classes which record basic pulse widths, there is a new additional receiver class for measuring the modulation frequency of an IR signal.

1.1.1 Receiver Overview

Infrared signals are received by an IR receiver device such as the first TSOP4438 or TSO58438 or similar. See the Hardware Considerations section for details. Although IR signals are sent as a series of modulated pulses, these receivers filter out the modulations and send you a signal in the form of a clean square wave. The output of the receiver is connected to one of the digital input pins of your Arduino. It is the job the receiver class to monitor this signal and to determine the length of the pulses known as “marks” in the length of the intervening idle periods known as “spaces”. The hardware receiver devices typically are normally high and the low when a signal is received. However the code compensates for this sort is useful to think of a “mark” as being “on” and a “space” as being “off”.

The duration of marks and spaces is measured in microseconds (1/1,000,000 of a second). Note however that the software makes use of the built-in Arduino function “micros(void)” which returns results in four microssecond granularity. Which means that values are always a multiple of four. This is for typical 16 MHz clocks on most Arduino models. On 8 MHz models results are in eight microsecond granularity. So whatever results we achieve are going to be no more accurate than that.

The receiver software is organized as a base class which of itself is not functional but contains methods which are common to all of the classes. The classes are:

- IRrecvBase Abstract base class
- IRrecv Original receiver class which uses 50 µs interrupt sampling
- IRrecvPCI Uses pin change interrupts to measure pulse duration
- IRrecvLoop Uses a tight code loop to poll the input pin

The data is stored in an array of unsigned int values. The first element of the array is the number of microseconds from the time the receiver is enabled until the time the first mark is received. This gap is typically discarded by the decoding routines. From there on, odd-numbered elements of the array contain the duration of the marks and even-numbered elements contain the duration of the spaces.

Most of the receiver classes are interrupt driven which means once they are initialized you can go off and do whatever you want until a complete sequence has been received. You determine when the sequence is complete by polling the method GetResults and when it returns true you can then obtain the results. Once a complete sequence has been received, the receiver class ignores any subsequent signals until it is reinitialized by you. However the IRrecvLoop is not interrupt driven and so when you call GetResults it sits in a tight loop and does not release control back to you until the complete signal has been received.

Although the library implements three different extensions to the base receiver class, you should have one and only one instance of a receiver class in your program because all the classes make use of global data and methods that are common to all of the classes. The data must be declared globally in the library because it is accessed by interrupt service routines. The structure of an interrupt service routines does not allow for the passing of any parameters. Therefore any data that the ISR accesses must be global.

You should read the next section about the base class because most of the methods work identically regardless of which derived class you actually instantiate. You should then read the section on the class which you are going to use so that you will understand the class specific issues.

Here is an extremely simple sample sketch showing the use of the IRrecv class. Either of the other two classes could be substituted in this example. You may wish to refer to this sample code while reading the description of the methods.

```
#include <IRLib.h>
IRdecode My_Decoder;
IRrecv My_Receiver(11); //Create the receiver. Use pin 11
void setup()
{
  Serial.begin(9600);
  My_Receiver.enableIRIn();//start receiving
}

void loop() {
  if (My_Receiver.GetResults(&My_Decoder)) {//wait till it returns true
    My_Decoder.decode();
    My_Decoder.DumpResults();
    My_Receiver.resume();     //restart the receiver
  }
}
```

1.1.2. The IRrecvBase class

This base class is an abstract class which in and of itself is not functional but contains common code for the other receiver classes. The prototype of the class is in "IRLib.h" and the code is implemented in "IRLib.cpp". The prototype is…

```
class IRrecvBase
{
public:
  IRrecvBase(void) {};
  IRrecvBase(unsigned char recvpin);
  void No_Output(void);
  void blink13(bool blinkflag);
  bool GetResults(IRdecodeBase *decoder, const unsigned int
Time_per_Ticks=1);
  void enableIRIn(void);
  virtual void resume(void);
  unsigned char getPinNum(void);
```

```
  unsigned char Mark_Excess;
protected:
  void Init(void);
};
```

The constructor is

```
  IRrecvBase(unsigned char recvpin);
```

The value passed is usually the pin number to which the receiver is connected. Although the single parameter is called "recvpin", in the case of the IRrecvPCI class, the value passed constructor is not the receiver pin number but is the interrupt number. See IRrcevPCI for details.

Creating an instance of the receiver class does not enable the receiving of data. You must enable the receiver by calling

```
void enableIRIn(void)
```

This method sets the input pin to input mode and resets the index into the array of data values to zero. On the interrupt driven classes this enables interrupts and the receiver starts recording intervals. However when using the IRrecvLoop class, recording of signals does not begin until you call GetResults.

If you are only receiving signals in your program, you need only call this method one time. However if your program sends and receives IR signals, it cannot do both simultaneously. The sending of an IR signal disables the receiver. After you have sent a code you must reenable the receiver class using enableIRIn().

Once the receiver is running you then need to poll the class to see if a complete sequence has been received. You do this by repeatedly calling…

```
bool GetResults(IRdecodeBase *decoder,
       const unsigned int Time_per_Ticks=1);
```

This method will return "true" when a complete sequence has been received and will return "false" otherwise. The first parameter is a pointer to a decoder class. This facilitates passing data from the receiver to the decoder.

The second parameter is optional. Is a multiplier which converts the recorded data into microseconds. The IRrecv class produces results in 50 µs ticks. It passes the number 50 as the second parameter so that GetResults will multiply the values in the array by 50 to convert them into actual microseconds. The other two receiver classes use the default multiplier 1 because they record actual microseconds do not need converting.

This method passes data into your decoder class. You can configure your decoder to use the same global array that is used by the receiver classes or your decoder can define its own extra buffer in which case the interval data is copied from the receiver's array to the decoder's array. The method also passes the index of the last array element used.

When a complete sequence has been received by the class it does not continue recording signals. It also does not resume recording once you have called GetResults because your decoder may be using the same array as the receiver and you do not want the receiver overwriting the array before you get it decoded.

To resume receiving data you must call the method…

virtual void resume(void);

That method is declared “virtual” because it is called by the IRrecvBase::enableIRIn method in the derived classes. This allows the derived classes to do things they need to do for themselves when resume is called.

In summary, you would typically call enableIRIn once at the beginning of your program and if your program uses the send classes you would need to use it after each sending of an IR signal. You would use “resume()” to resume reception after decoding or immediately after GetResults if your decoder uses its own array. You need not call “resume()” if you have called “enableIRIn()” because it will call it for you.

The remaining methods are not essential to the operation of the receiver but are useful.

For debugging purposes you may want to know if your Arduino is receiving a signal. If you initially call

```
void blink13(true)
```

Pin 13 will be blinked on every time a mark is received and blinked off when a space is received. The default is off and blinking will not occur.

Some users create custom boards for input and output of IR signals and those boards are connected to their Arduino even in the case where there really a sketch that only does input. It is theoretically possible that when running an “input only” sketch that the output pin could initialize high and your output LED would be on all the time. LED driver circuits are sometimes designed to overdrive the LED because it is used only intermittently. If it were to be accidentally left on continuously, it could burn out your circuit. If you want to ensure that this does not happen you can call.

```
void No_Output(void);
```

This will ensure that your output pin is set low. Note that in this library the output pin is determined at the time that the library is compiled and is based on the choice of internal hardware timer in the “IRLibTimer.h” file. That is why the receiver team can know what output pin to drive low.

As noted earlier, the constructor for IRrecvPCI is passed an interrupt number rather than a pin number. The pin number is computed based upon the type of hardware you are using and interrupt chosen. You can return the value of this pin number using…

```
unsigned char getPinNum(void);
```

This method also works for the other receiver classes although you probably already know which pin you are using because you sent it that information in the constructor.

Depending on the type of IR receiver hardware you are using, the length of a mark pulse is over reported and the length of a space is underreported. Based on tests performed by Ken Shirriff who wrote the original IRremote library upon which this library is based, the length of a received mark is about 100µs too long and a space is 100µs too short. That is the default value used by the receiver classes. However my own experience is that 50µs is a better value. In order to maintain backwards compatibility with earlier versions of the library we have maintained the 100µs default however you can now change that value based on your own experiences by changing the value of

```
unsigned char Mark_Excess;
```

You can examine or change this variable as desired. It is applied by adding that value to all odd-numbered elements of the interval array and is subtracted from even-numbered elements when the data is passed to your decoder by GetResults.

There is one protected method which is used for internal use.

```
void Init(void);
```

Is called by the constructors and it initializes the blink flag and the default mark excess values. There is also a global function used by the receiver class.

```
void do_Blink(void);
```

This performs the actual blinking of pin 13. It cannot be a method of this class because it is called from within an interrupt service routine. This function is used internally only.

1.1.3. IRrecv Class

This receiver class is based on the original receiver created by Ken Shirriff in his library IRremote upon which this library is based. It uses a hardware timer to trigger an interrupt every 50µs. Inside the interrupt routine it counts the number of these 50µs ticks while the pin is a mark and when the state changes it counts how many ticks in the space. When it receives an extraordinarily long space it presumes that the sequence has ended. It sets an internal flag noting that the sequence has been received. It stops recording time intervals and when the user calls GetResults the next time it will return true.

The internal hardware timer used is controlled by settings in the “IRLibTimer.h” file. Each type of Arduino platform such as Arduino Uno, Leonardo, Mega etc. has a choice of different timers. For example the Uno uses TIMER2 while the Leonardo uses TIMER1. You may need to change the default timer in the event of a conflict with some other library. For example the Servo library makes use of TIMER1 so if you’re using a Leonardo with a servo you would need to change the value in “IRLibTimer.h” to use a different timer.

The prototype of the class is in “IRLib.h” and the code is implemented in “IRLib.cpp”. The prototype is…

```
class IRrecv: public IRrecvBase
{
public:
  IRrecv(unsigned char recvpin):IRrecvBase(recvpin){};
  bool GetResults(IRdecodeBase *decoder);
```

```
  void enableIRIn(void);
  void resume(void);
};
```

As previously noted in the discussion of the base class, the constructor is passed the pin number of the input pin to which you have connected your receiver. There are no restrictions and any digital input pin can be used.

Note that when calling GetResults you only use the first parameter which is a pointer to your decoder class. Although the prototype only lists the constructor and three other methods, all of the methods available to the base class are available in this class.

You will need to enable input by calling enableIRIn() and will need to poll the GetResults method in your loop until it turns true. Although in our example code we called GetResults in a very small loop, because this class is interrupt driven you can do just about anything else inside your main loop function and only call GetResults when you are ready.

A reminder that when a complete stream has been received, no additional measurements are taken until you call the “resume()” method. If your decoder does not use its own buffer and relies upon the global buffer for the library, be sure not to call resume until your decoding is complete when you no longer have need of the buffer.

Because the interrupt service routine for this class must be declared globally, it is always created even if you do not use the IRrecv class. If you are using a timer interrupt for some other purpose, it can create a conflict because there are duplicate ISR functions declared. There is a conditional compile flag available in IRLib.h that effectively removes this entire class and the ISR from the library. If you need to remove the class you should comment out the line that reads

```
#define USE_IRRECV
```

There is no similar define to disable IRrecvPCI or IRfrequency classes because they use a different type of ISR with a unique name that will not conflict with other libraries or code. Also the IRrecvLoop is not interrupt driven so it will never conflict and does not need to be disabled.

Because this receiver only samples the input every 50µs there is a chance that it could sample at inopportune times and be as much as 98µs off when measuring intervals. If you are decoding a known protocol, this margin of error is usually acceptable. The decoder functions typically use +/-25% tolerance and that produces acceptable results. However if you are trying to analyze an unknown protocol you would be better suited to use either the IRrecvPCI or IRrecvLoop receiver class instead.

1.1.4. IRrecvPCI Class

This receiver class makes use of the hardware interrupt available on some pins of Arduino microcontrollers. It was created because it gives more accurate timings than the original IRrecv class which only samples the input every 50µs. The code is loosely based upon and inspired by the work by the developers of the AnalysIR program. AnalysIR is a Windows-based application which allows you to graphically analyze IR input signals through and Arduino, Raspberry Pi or other microcontrollers systems. Many thanks to the developers of that software for their assistance and input into the development of this class. You can find more about their software at http://analysir.com

The class sets up the pin change hardware interrupt which calls the interrupt service routine every time the input pin switches from low to high or high to low. At each change, the code calls the built in function “micros()” and that value is subtracted from the timestamp of the previous change. Because the micros() function is only accurate to 4µs on 16 MHz systems or 8µs on 8 MHz systems, that is the limitation of accuracy of this method.

While it is much more accurate than the original IRrecv class which only had 50µs or worse accuracy, it may not be suitable for everyday use. The class has difficulty determining when a sequence has ended. Normally we assume a sequence has ended when a space interval is longer than a certain amount. But we cannot know how long the final trailing space is until the first mark of the next sequence begins. The code attempts to compensate for this by checking for an extremely long space each time that the user calls GetResults. However unless you call that routine extremely frequently, it is more likely that the next sequence will begin. While that does not adversely affect the reception and subsequent decoding of an initial sequence, if the next sequence comes quickly then the receiver may miss it or may start reception in the middle of a sequence thus giving only partial therefore jumbled results. To avoid this problem we suggest that you not resume the receiver immediately but instead put a “delay(500);” or longer statement after you have finished decoding.

Although this class does not use internal hardware timers, it is still recommended that you call “enableIRIn()” after using any sender routines.

As with the IRrecv class, the interrupt service routine is disabled once a completed sequence has been detected and is not re-enabled until you call either “enableIRIn()” or “resume()”.

The prototype of the class is in “IRLib.h” and the code is implemented in “IRLib.cpp”. The prototype is…

```
class IRrecvPCI: public IRrecvBase
{
public:
  IRrecvPCI(unsigned char inum);
  bool GetResults(IRdecodeBase *decoder);
  void resume(void);
private:
  unsigned char intrnum;
};
```

Note that the parameter passed to the constructor is an interrupt number and not a pin number. The value you specify is passed to the built in “attachInterrupt()” function. The table below shows which interrupts are available on various types of Arduino hardware and the resulting pin numbers used by those interrupts. **NOTE: these interrupt numbers which are passed to “attachInterrupt()” are not necessarily identical to the interrupt numbers in the datasheet of the processor chip you are using. These interrupt numbers are a system unique to the “attachInterrupt()” Arduino function.** For more information on attachInterrupt see http://arduino.cc/en/Reference/AttachInterrupt

Board	int.0	int.1	int.2	int.3	int.4	int.5	Int.6

Uno, Ethernet	2	3					
Mega2560	2	3	21	20	19	18	
Pinoccio	4	5	SCL(15)	SDA(16)	RX1(13)	TX1(14)	7
Leonardo	3	2	0	1	7		
Due	(see below)						

The Arduino Due board has powerful interrupt capabilities that allow you to attach an interrupt function on any available pins. This library has not been tested using this platform but theoretically should work.

Note this table is also used by the IRfrequency class and is implemented by the following global function which is implemented in IRLib.cpp and whose prototype is in IRLib.h..

```
unsigned char Pin_from_Intr(unsigned char inum);
```

The constructor takes the interrupt number which you pass and computes the pin number. If you wish to verify which pin number it has chosen based on your interrupt you can use the getPinNum() function to make sure that the hardware detection is working properly.

Although the prototype of this class only lists 2 methods, all of the methods of the base class are available.

1.1.5. IRrecvLoop Class

This version of the receiver class uses a tight internal loop to poll the input pin. It makes no use of hardware interrupts or internal timers to compute the intervals. It does use the “micros()” function to compute time intervals so it has the limitations of that function. Specifically the function is only accurate to 4µs on 16 MHz systems or 8µs on 8 MHz systems. Although we have attempted code the loop as tightly as possible there still is some amount of overhead in the loop itself which could affect the accuracy. However it is still much more accurate than the 50µs off the original IRrecv. This makes it a good choice for analyzing unknown protocols.

Because no interrupts are involved, the class does not begin recording intervals when you initially call “enableIRIn()” as is the case with other classes. This class only samples input when you call GetResults. That method then takes over control of your program and does not relinquish it until it has received a complete sequence. The function always returns true. Because it takes over control of your program and does not allow you to do other things while it is looking for a sequence, this version of the class may not be practical for everyday use.

It is still recommended that you follow all of the procedures for when to call enableIRIn() and or resume() methods as you would with other versions of the class.

The prototype of the class is in “IRLib.h” and the code is implemented in “IRLib.cpp”. The prototype is…

```
class IRrecvLoop: public IRrecvBase
{
public:
  IRrecvLoop(unsigned char recvpin):IRrecvBase(recvpin){};
```

```
  bool GetResults(IRdecodeBase *decoder);
};
```

Although the prototype only lists the constructor and GetResults, all of the methods of the base class are available. Like IRrecv, the value passed to the constructor is the number of the receiver pin. Any available digital input pin can be used.

1.1.6. Global Receiver Structures

In addition to the base class and the three derived classes that are used for receiving IR signals, there is a globally defined structure that is used by the classes. Good object-oriented design would have us put all of the data associated with a class inside the class itself. However much of the data we use needs to be accessible from an interrupt service routine. A limitation of an ISR is that we cannot pass it any parameters and it cannot be part of a class. So all of the data used by the ISR must be in some globally available location. Unless you're going to implement your own receiver class or create custom decoder classes need access to this data, you need not deal with the structure. It is solely for internal use. The file "IRLibRData.h" contains the following definitions

```
enum rcvstate_t {STATE_UNKNOWN, STATE_IDLE, STATE_MARK, STATE_SPACE,
STATE_STOP, STATE_RUNNING};
typedef struct {
  unsigned char recvpin;    // pin for IR data from detector
  rcvstate_t rcvstate;       // state machine
  bool blinkflag;         // TRUE to enable blinking of pin 13 on IR
processing
  unsigned long timer;     // state timer, counts 50uS ticks.(and other uses)
  unsigned int rawbuf[RAWBUF]; // raw data
  unsigned char rawlen;         // counter of entries in rawbuf
}
irparams_t;
extern volatile irparams_t irparams;
```

The enum values are the "states" in which the interrupt service routine keeps track of the receiving process. They are used by IRrecv and IRrecvPCI.

The structure called irparams gives Jack of the input pin, the state of the receiver, the flag used by the pin 13 blinker, a variable to count the number of ticks in the interval, the array which contains the length of sequence of marks and spaces and the index into that array which will contain the length of the sequence when it is complete.

Note that the entire structure is declared "volatile" which informs the compiler to always immediately store results in their location and immediately retrieve them when needed because they may get changed by the ISR in the middle of the calculation.

The decoder class contains a pointer to an array. The default is that that pointer is initialized to "irparams.rawbuf" which is the same array used by the receiver class. You have the option to define your own array for use by the decoder by changing that pointer to a different array. The receiver method GetResults will copy the values either to itself if you did not specify a different array or it will copy the results to your array if you have changed that pointer. See the section on IRdecodeBase::UseExtnBuf(void *P); for details. The GetResults method also copies rawlen to a similarly named variable in the decoder class.

1.1.7 IRfrequency Class

Normally you would use a TSOPxxxx IR receiver device to detect IR signals. These devices demodulate the signal into square waves measured in hundreds or thousands of microseconds. However the actual IR signals are modulated at frequencies from 36 kHz as high as 57 kHz depending on the protocol. If you have an unknown protocol, you can typically receive such a signal using a device that has been tuned to 38 kHz however if you then want to accurately retransmit the signal you need to know the actual frequency. This requires a different type of receiver such as a TSMP58000 that passes the modulated signal through directly. See the section Hardware Considerations for more info on connecting these devices.

Because the signals are so short (17.5 µs to 27.7 µs) it is difficult to accurately measure them. The only way is to use a hardware interrupt pin and an extremely fast interrupt service routine. However these measurements are limited to the accuracy of the micros() function which is only accurate to 4 µs on 16 MHz systems and 8 µs µs on 8 MHz systems. The accuracy of the measurement can be slightly improved by taking several hundred measurements and averaging them.

The IRfrequency class implements an interrupt driven system for measuring frequency.

See the examples folder for sample code using this class. The IRrecvDumpFreq example requires both a TSOP and TSMP device connected to 2 different pins. It detects both the frequency using IRfrequency and the pattern of pulses using IRrecvPCI or IRecvLoop receivers. You could also use the original IRrecv receiver class but it degrades the accuracy of the frequency calculation because the interrupt every 50 µs interferes. The other sample code is IRfreq and it uses only the TSMP device to measure frequency and it does nothing else. Measuring frequency by itself is the most accurate method but there may be applications where you need to measure frequency and protocol simultaneously.

The prototype of the class is in “IRLib.h” and the code is implemented in “IRLib.cpp”. The prototype is…

```
class IRfrequency
{
public:
  //Note this is interrupt number, not pin number
  IRfrequency(unsigned char inum);
  void enableFreqDetect(void);
  bool HaveData(void);      //detective data received
  void disableFreqDetect(void);
  void ComputeFreq(void);     //computes but does not print results
  void DumpResults(void);     //computes and prints results
  unsigned char getPinNum(void);//get value computed from interrupt number
  double Results; //results in kHz
  unsigned char Samples; //number of samples used in computation
private:
  volatile unsigned char Time_Stamp[256];
  unsigned char intrnum, index, pin;
  unsigned long Sum;
```

```
};
```

The parameter to the constructor is the interrupt number you will use for detecting the frequency. Note this is the same system of interrupt numbers used by the IRrecvPCI class. It is not the actual pin number for connecting the device. See the table in the section on IRrecvPCI. You can also use the getPinNum() method to retrieve the pin number once the class has been created. Note that although some of the methods of this class have the same name as the IRrecvBase class, it is not a derived class from that base. As with other receiver classes, you should instantiate only one copy of this class.

The class does not begin collecting frequency data until you call enableFreqDetect() method. This attaches the interrupt routine and begins waiting for a signal. The built in interrupt service routine calls micros() each time a signal is detected. It stores the least significant byte of that value in a 256 byte array. It continually detects data in the array and when it fills, the index wraps around. This overflow is very likely to occur because an IR stream will usually contain thousands of pulses. Therefore the measurement is only based upon the most recent 256 data points.

If you are simultaneously receiving a signal using a TSOPxxxx device and one of the three available receiver classes as in the IRrecvDumpFreq example, you can continue to operate the frequency detector until a complete sequence has been detected. However if you are measuring frequency alone as with the IRfreq example, you need to know when to quit. You can call the HaveData() method repeatedly in your main loop. It will return true if at least one buffer full of data has been collected.

When you are finished collecting data, you should call the disableFreqDetect() method. This will detach the interrupt and no further data is collected until you call enableFreqDetect() again. You then have a choice of either calling ComputeFreq which computes the frequency based on the timestamp data or DumpResults which will call ComputeFreq for you and display the results using Serial.print().

The computation measures the difference between successive timestamps. While the majority of the timestamps will be the time between pulses of the modulated signal, some of them will be the gaps caused by the spaces in the protocol. These gaps will be on the order of hundreds of microseconds rather than tens of microseconds. The computation discards these intervals and only considers those which are likely to be from the modulated signal itself. Because we’re only storing the least significant byte of data, some of the values will be inaccurate. The code attempts to eliminate anything that is unusually large or unusually small. It then averages the usable samples and computes a frequency measured in kHz. ComputeFreq stores the results in a float double value Results. It also tells you the number of actual samples used in the computation in the unsigned char Samples. If the Samples value is extremely small, you might wish to discard that data and remeasure.

The text strings of the DumpResults method are stored in program memory rather than RAM memory. However if you are not using the DumpResults method and you are running out of program memory may wish to disable this function. In IRLib.h there is a define

```
#define USE_DUMP
```

You can comment out that line was to save program memory. Note that this parameter also affects the DumpResults method of the decoder classes as well.

1.2 Decoder Classes

IR signals are encoded as a stream of on and off pulses of varying lengths. The receiver classes only record the length of pulses which we call "marks" and the intervals between them which we call "spaces". However it is the decoder class which identifies the protocol used and extracts the data. It provides the user with the type of protocol it found if any, the value received, and the number of bits in that value. We implement the decoder as an abstract base class and nine additional derived classes. Seven of the classes are for the seven protocols which are built into the library. An eighth class consolidates the seven other classes into one omnibus class that decodes all seven protocols. A ninth class turned the data into a 32-bit hash code which can be used for deriving a value for data from an unknown protocol. Note however that the hash code is only good for detecting signals and cannot be used to re-create the signals for sending again.

Note: additional protocols are demonstrated in various example sketches provided with this library. These examples illustrate how to extend the library using object-oriented programming techniques without having to recompile it. It will be our policy for the foreseeable future not to directly add additional protocols to the library but to implement future protocols as example sketches. We will not accept any pull requests which add additional protocols to the main library. We will also not accept additional examples which do not implement both sending and decoding classes.

1.2.1. Decoding Overview

The data from the receiver class is in the form of an array of time intervals of the marks and spaces that constitute the data stream. That stream typically begins with some sort of header followed by the encoded bits of data which could be from 8 up to 32 or more bits followed by some trailer. Occasionally there are other sequences in the middle of the stream that are not actually part of the data that serve to separate the data into different sections. In order to make good use of the information we need a decoder which will take this data and convert it into a single binary value which identifies the particular function the remote is using.

The data sent by a remote often contains information such as a device number,-sub-device, function number, sub-function and occasionally information that designates that this is a repeated signal. The philosophy of this library is to not care about what the data represents. We take the philosophy that "You push the button and this is the stream of data that you get." Our job is to get you that binary number usually expressed in hexadecimal and it's up to you to decide what to do with it.

If you are using a supported protocol, that hexadecimal number can then be fed into a send class which will output the IR signal identical to the one that you received. There is one exception in that one of the decoders used for unknown protocols creates a 32-bit hash code from the input sequence. The hash code is extremely likely to be a unique representation of the original stream but there is no way to reverse that and re-create the stream from the hash code.

Different manufacturers use different protocols for encoding this data. That is what allows you to have a universal remote that can operate devices by different manufacturers and not have the signals get mixed up. That creates a problem for us because we need different programs to decode each different protocol. This library supports seven of the most common protocols. The example programs included with the library show how to decode and encode a number of additional protocols. We have seen references online to dozens of others which we do not support yet.

The library has a base decoder class and 10 additional extended classes. Seven of those classes are for the seven protocols we support directly. An eighth class creates a hash code out of that raw data which can turn the data into a unique binary number. Note that the hash code cannot then be used to re-create the original sequence for sending the data out again. The ninth class is an abstract class which defines common methods used by both RC5 and RC6 protocols. Finally, the tenth class combines the seven supported protocols into a single decoder.

1.2.2. IRdecodeBase Class

The library defines a base decoding class that is an abstract class which does not in and of itself do anything. All other decoder classes are extensions of this class. The prototype is in "IRLib.h' and the code itself is in "IRLib.cpp". The prototype for the class is…

```
typedef char IRTYPES; //formerly was an enum
class IRdecodeBase
{
public:
  IRdecodeBase(void);
  IRTYPES decode_type;              // NEC, SONY, RC5, UNKNOWN etc.
  unsigned long value;              // Decoded value
  unsigned char bits;               // Number of bits in decoded value
  volatile unsigned int *rawbuf;    // Raw intervals in microseconds
  unsigned char rawlen;             // Number of records in rawbuf.
  bool IgnoreHeader;             // Relaxed header detection allows AGC to
settle
  virtual void Reset(void);         // Initializes the decoder
  virtual bool decode(void);        // This base routine always returns
                                    // false override with your routine
  bool decodeGeneric(unsigned char Raw_Count, unsigned int Head_Mark,
            unsigned int Head_Space, unsigned int Mark_One,
            unsigned int Mark_Zero, unsigned int Space_One,
            unsigned int Space_Zero);
  virtual void DumpResults (void);
  void UseExtnBuf(void *P);   //Normally uses same rawbuf as IRrecv.
                              //Use this to define your own buffer.
  void copyBuf (IRdecodeBase *source);    //copies rawbuf and rawlen from
                                          //one decoder to another
protected:
  unsigned char index;           // Index into rawbuf used various places
};
```

The constructors for the base class and for any of its derived classes take no input. Unlike the receiver classes there is no problem creating an instance of multiple decoder objects because each has its own internal data associated with it.

Data is passed from the receiver class to the decoder class when you call GetResults and pass it a pointer to your decoder. Get Results starts by calling your decoders “Reset()” method to clear out any previous data. It resets the decode_type to UNKNOWN and zeros out other values. It then copies the interval timing data into the array rawbuf and sets rawlen to the index of the last array element used. It is unlikely you would ever have need to call the Reset() method yourself but it is available if you need it.

User should then call the “decode()” method. It will analyze the raw data and if it is a recognized protocol it will set the “decode_type” value as one of the following values found in IRLib.h.

```
#define UNKNOWN 0
#define NEC 1
#define SONY 2
#define RC5 3
#define RC6 4
#define PANASONIC_OLD 5
#define JVC 6
#define NECX 7
#define HASH_CODE 8
#define LAST_PROTOCOL HASH_CODE
```

That “decode()” method also sets “value” to the binary value it decoded and sets “bits” to the number of bits of data. Most protocols have a fixed number of bits but some protocols such as Sony have different versions of the same protocol using different numbers of bits.

The method “DumpResults()” can be called after “decode()” to dump information about the received data to the serial monitor. You will have had to initialize serial output using “Serial.begin (int)” prior to calling it. Here is an example of typical data output by this method

```
Decoded Sony(2): Value:74BCA (20 bits)
Raw samples(42): Gap:11950
  Head: m2300  s700
0:m550 s600 1:m1200 s600            2:m1150 s650     3:m1150 s600
4:m600 s600 5:m1150 s650            6:m600 s600      7:m550 s600
8:m1200 s600      9:m550 s650       10:m1150 s650    11:m1150 s600
12:m1150 s650     13:m1100 s650           14:m600 s600     15:m600 s600


16:m1150 s650     17:m550 s650            18:m1150 s600    19:m550
Extent=32650
Mark  min:550      max:1200
Space min:600      max:650
```

The output identifies this a Sony protocol which is protocol “2”. The received value in hexadecimal is 0x74BCA and is 20 bits long. The stream contained 42 intervals (that is the value rawlen). It then dumps out all of the values from rawbuf. The first element of that array is the amount of time between initializing of the receiver and the first received mark. This gap is ignored by the decoder. The next two values are the length of the mark and space of the header sequence. The remaining values are the lengths of the marks and spaces of the data bits. Each mark is preceded by the letter “m” and spaces are “s”. The values are in microseconds. Because we used the IRrecv receiver you will note that all of the values are in 50µs increments.

At the end of the data bits, the method also reports the sum total of all of the intervals from the header through the stop bit. That information is significant for some protocols. It also tells you the maximum and minimum values of mark and space for data bits which can be useful for analyzing unknown protocols. Additional data analysis of unknown protocols can be found in the IRanalyze example sketch in the examples folder.

The text strings of the DumpResults method are stored in program memory rather than RAM memory. However if you are not using the DumpResults method and you are running out of program memory may wish to disable this function. In IRLib.h there is a define

```
#define USE_DUMP
```

You can comment out that line was to save program memory. Note that this parameter also affects the DumpResults method of the IRfrequency class as well.

When a decoder is created the pointer rawbuf is set equal to the global value irparams.rawbuf which is the global buffer used by the receiver class. In some circumstances you may wish to have your decoder use its own buffer so that you can resume receiving data while working with the previously received data in your decoder. You can change the buffer used by your decoder with the method

```
  void UseExtnBuf(void *P);
```

You can pass a pointer to your new buffer and then GetResults will copy the raw data into this buffer instead of retaining it in the global buffer.

On some occasions you might wish to have more than one decoder and want to share the data between different decoders. In that case you would use…

```
  void copyBuf (IRdecodeBase *source);
```

Note that the pointer you pass to that method is the pointer to the entire decoder class and not just the buffer. This method copies not only the values from rawbuf but it also copies rawlen.

The only remaining method in the base class is “genericDecode”. It is a general-purpose decoding routine used by many of the other protocols that share common attributes. It will be documented in the section on adding additional protocols to the library.

In addition to the base decoder class library also defines a function Pnames(IRTYPES type); which returns a pointer to a string containing the name of the protocol such as “Sony”, “NEC” etc. These strings are stored in flash memory or program memory so that it does not use valuable data RAM memory. It is used by DumpResults and may be useful to the end user as well.

Most protocols begin with an exceptionally long “mark” pulse which serves a number of purposes. One of the reasons it is generally long is that receiver devices have a special circuit called Automatic Gain Control or AGC that adaptively boosts the signal received so that it can be cleanly decoded. It has been our experience that when conditions are marginal such as a signal at an odd angle or a weak signal, then it takes time for the AGC to adjust itself. The result is the length of the opening “mark” is shortened. Many failed decodes occur because the header is too short. Consumer electronics devices such as TVs, DVD’s etc. only need to support a single protocol. So they do not need to concern themselves with the duration of the header in order to decide what to do. However our decoder classes all require that the received header pulse be reasonably close to the specification. You can turn off the requirement that the initial mark of the header is within normal tolerances. You do so by setting the value IgnoreHeader=true; This can improve the ability to decode weak signals by allowing the AGC to address.

Warning however if you’re using multiple protocols, it can give you an accurate results of the protocol type. Specifically the NEC and NECx protocols are identical except for the length of the initial header mark. Therefore an NECx signal would be reported as NEC when the IgnoreHeader parameter is true. Unfortunately the data values are unaffected. There may be other as yet unsupported pairs of protocols which cannot be distinguished from one another except by their header.

1.2.3. Specific Protocol Decoder Classes

The seven protocols shown below are each supported with their own decoder class. If you only need to use one protocol you should create an instance of one of these classes. The prototypes are in “IRLib.h” and the code itself is in “IRLib.cpp”. Each contains only one method “decode()” but of course they also have access to the data values and methods of the base class. The “decode()” method returns true if the data received could be successfully decode as that protocols and it returns false otherwise. The classes are…

```
class IRdecodeNEC: public virtual IRdecodeBase
      {public: virtual bool decode(void);};
class IRdecodeSony: public virtual IRdecodeBase
      {public: virtual bool decode(void);};
class IRdecodeRC5: public virtual IRdecodeRC
      {public: virtual bool decode(void);};
class IRdecodeRC6: public virtual IRdecodeRC
      {public: virtual bool decode(void);};
class IRdecodePanasonic_Old: public virtual IRdecodeBase
      {public: virtual bool decode(void);};
class IRdecodeJVC: public virtual IRdecodeBase
      {public: virtual bool decode(void);};
class IRdecodeNECx: public virtual IRdecodeBase
      {public: virtual bool decode(void);};
```

The NEC, Sony and JVC protocols are used by those brands of equipment. The NECx is an extension of NEC used by them and other manufacturers. The RC5 and RC6 protocols were developed by Phillips and used by a variety of manufacturers. Panasonic_Old is used by cable boxes by Scientific Atlanta and Cisco mostly used by Time Warner and BrightHouse cable systems. The example code included with the library has similar routines which support DirecTV, Samsung36 and GIcable the last of which is used by Motorola cable boxes. Also the rcmm used by AT&T U-Verse is in the examples folder.

The NEC protocol uses a special sequence as a repeat code. With the NEC decoder detects a repeat pattern that will return the value 0xffffffff. Other protocols especially RC5 and RC6 makes use of a toggle bit. So if you decode successive presses of the same button, the value you will receive will have one bit inverted each time. However if you hold the same button down, the remote sends repeated codes and the bit will not change. This library has no built-in features for dealing with toggle bits. It treats those bits just like any other bit in the stream. It generally does not hurt to leave the toggle bit either set or reset all of the time.

There is an additional abstract class called IRdecodeRC which is the base class for both RC5 and RC6. Unless we find another protocol based upon those, you will probably never had of it directly.

Library also includes a hash code decoder class. It takes the array of data values and attempts to create a unique 32-bit code based on whether each value is higher, lower, or equal to the previous element of the array. It allows you to come up with an identifying number for an unknown protocol function. However there is no way to re-create the original data stream from that hash code. But for some applications using an unknown protocol it can be useful.

There is an overall IRdecode class which combines all seven protocols into a single class. It is defined as follows…

```
class IRdecode:
public virtual IRdecodeNEC,
public virtual IRdecodeSony,
public virtual IRdecodeRC5,
public virtual IRdecodeRC6,
public virtual IRdecodePanasonic_Old,
public virtual IRdecodeJVC,
public virtual IRdecodeNECx
      {public: virtual bool decode(void);};
```

It's decode method calls the decode methods of each of the seven defined protocols in that order and if it finds any of them returning true then it returns true as well. If all of the routines fail then it returns false. While this is useful when using multiple protocols, it creates overhead because all of the code for each of the protocols gets included in your sketch. If you are only using two or three of these seven protocols then you are wasting a lot of code space. It is recommended you create your own custom omnibus decode class that combines only the protocols which you are using. You can either comment out the unused protocols in this given IRdecode class or create your own custom class by copying its definition and getting rid of the unnecessary protocols.

Note that the linking loader is smart enough not to include code from a protocol which appears nowhere in your sketch. So if you do not create an instance of this omnibus decoder, you should include only the protocols you actually use.

1.3 Sending Classes

The decoder classes defined by this library all return a binary data value, the number of bits in that data value, and the type of protocol received. That information can then be used to re-create the original signal and send it out using an infrared LED. The sending classes allow you to re-create the signals that are sent by remote controls using the protocols supported by this library.

1.3.1. Sending Overview

The decoder classes defined by this library all return a binary data value, the number of bits in that data value, and the type of protocol received. That information can then be used to re-create the original signal and send it out using an infrared LED. There are a number of online references and other software such as LIRC which provide lists of binary values for different kinds of remotes and different functions. However they may have their own unique way of encoding the data that may or may not be compatible with the values used by this library. Therefore you should use the receiving and decoding classes to detect the signal from a remote and obtain the value that this library creates. Then you can use that value and the sending classes provided to re-create that stream and control some device.

You should read the decoding overview section of this reference to familiarize yourself with the protocols. We will not re-create that discussion here.

The library consists of an abstract base sending class and nine derived classes. Seven of them are for the seven supported protocols. One of them sends raw data out of the rawbuf array for use with unknown protocols and the ninth class is an omnibus class which combines all seven of the supported protocols into a single sending class.

Note: additional protocols are demonstrated in various example sketches provided with this library. These examples illustrate how to extend the library using object-oriented programming techniques without having to recompile it. It will be our policy for the foreseeable future not to directly add additional protocols to the library but to implement future protocols as example sketches. We will not accept any pull requests which add additional protocols to the main library. We will also not accept additional examples which do not implement both sending and decoding classes.

1.3.2. IRsendBase Class

The prototypes for the sending classes are in “IRLib.h” and the actual code is contained in “IRLib.cpp”. The prototype is…

```
class IRsendBase
{
public:
  IRsendBase();
  void sendGeneric(unsigned long data, unsigned char Num_Bits,
            unsigned int Head_Mark, unsigned int Head_Space,
            unsigned int Mark_One, unsigned int Mark_Zero,
            unsigned int Space_One, unsigned int Space_Zero,
            unsigned char kHz, bool Stop_Bits,
            unsigned long Max_Extent=0);
protected:
  void enableIROut(unsigned char khz);
  VIRTUAL void mark(unsigned int usec);
  VIRTUAL void space(unsigned int usec);
  unsigned long Extent;
};
```

The base class is completely abstract. The constructor takes no parameters. The methods are all for internal use or for the creation of derived classes.

1.3.3. Specific Protocol Sending Classes

The seven protocols shown below are each supported with their own sender class. If you only need to use one protocol you should create an instance of one of these classes. The prototypes are in “IRLib.h” and the code itself is in “IRLib.cpp”. Each contains only one method “send()”. Although these derived classes have access to the methods and data of the base class, the end-user has no need of the other methods in the base class unless you are implementing an unsupported protocol. They are all for internal use. The classes are…

```
class IRsendNEC: public virtual IRsendBase
      {public: void send(unsigned long data);};
class IRsendSony: public virtual IRsendBase
      {public: void send(unsigned long data, int nbits);};
class IRsendRC5: public virtual IRsendBase
      {public: void send(unsigned long data);};
class IRsendRC6: public virtual IRsendBase
      {public: void send(unsigned long data, unsigned char nbits);};
class IRsendPanasonic_Old: public virtual IRsendBase
      {public: void send(unsigned long data);};
class IRsendJVC: public virtual IRsendBase
      {public: void send(unsigned long data, bool First);};
class IRsendNECx: public virtual IRsendBase
      {public: void send(unsigned long data);};
```

Each class contains only one method a “send()” method. The first parameter is the data value to be sent. Some protocols have an additional parameter.

Note that there is no “IRsendHash” class because there is no way to re-create the data stream from the hash code created by the “IRdecodeHash” decoder class.

Sony and RC6 have different versions which use different numbers of bits. Their second parameter is the number of bits.

The JVC protocol sends data differently whether it is the first code sent or it is a repeat of the same code. In our experience you must send JVC codes twice. The first time as an original code and then again as a repeated code. The second parameter is a bool that should be true for the first sending and false for the second sending. See the ”IRsendJVC” example program to see how.

The NEC protocol uses a special sequence as a repeat code. With the NEC decoder detects a repeat pattern that will return the value 0xffffffff. If you pass that value to the send method of the NEC class it will send a special repeat pattern used by that protocol.

Other protocols especially RC5 and RC6 makes use of a toggle bit. So if you decode successive presses of the same button, the value received will have one bit inverted each time however if you hold the same button down, the remote sends repeated codes and the bit will not change. This library has no built-in features for dealing with toggle bits. It treats those bits just like any other bit in the stream. It generally does not hurt to leave the toggle bit either set or reset all of the time. The toggle bit for RC5 is 0x0800 and for RC6 is 0x10000. You would do a bitwise exclusive or with data to toggle that bit. See the “IRrecord” example code which shows how to send any values on any of the supported protocols and it includes support for toggle bits.

That example also shows how to use an additional class to send raw data values for protocols that are unsupported. If you receive an unknown protocol you can copy all of the values from the rawbuf array and save them. They can then re-create the original data stream using the IRsendRaw::send class and method. The prototype is shown here

```
class IRsendRaw: public virtual IRsendBase
      {public: void send(unsigned int buf[],
            unsigned char len, unsigned char khz);};
```

The first parameter points to the array of values. You also need to specify the number of intervals. The third parameter is the modulation frequency to use. If you do not know the modulation frequency then we recommend that you use 38 for this value. The receiver and decoder classes cannot normally detect the modulation frequency. See the section on the IRfrequency class to see how to detect modulation frequency. For more information see the section on implementing your own protocols.

Final class is an omnibus class that will send any of the supported protocols. Its prototype is…

```
class IRsend:
public virtual IRsendNEC,
public virtual IRsendSony,
public virtual IRsendRaw,
public virtual IRsendRC5,
public virtual IRsendRC6,
public virtual IRsendPanasonic_Old,
public virtual IRsendJVC,
public virtual IRsendNECx
{
public:
  void send(IRTYPES Type, unsigned long data, unsigned int data2);
};
```

Its send method takes three parameters. The first is the protocol type. The second is the binary data to be sent. The third value serves multiple purposes. For some protocols is the number of bits. For other protocols it might be the first flag. In the example code the Samsung36 example uses the third parameter to contain part of the actual received data. That is because it is a 36 bit protocol and we can only use at most 32 bits in an integer variable. For protocols that did not need this third parameter you should set it to zero.

1.4 Hardware Considerations

Obviously for this library to be of any use you will need to add some hardware to your Arduino. Specifically an IR LED possibly with a driver circuit for output and some sort of infrared receiver for input purposes.

Most of the features of this library make use of various built-in hardware specific features of the Arduino platforms. They make use of hardware interrupts and built in the hardware timers. Because different Arduino platforms use different base chips, even something as simple as “What pin do I connect to?” Can be a difficult issue to resolve. Additionally more and more platforms beyond the traditional 8-bit Atmel are being labeled with the term “Arduino compatible” which makes the hardware issues even more complicated. We will address all of these issues in the following sections.

1.4.1. Interrupts and Timers

The hardware specific portions of the library are based on the fact that it uses hardware timers and hardware interrupts.

The IRrecvPCI and IRfrequency receiver classes make use of pin change hardware interrupts. This is a hardware feature which causes control of your program to branch immediately to a routine that you specify whenever a specific input pin changes. Different platforms have different numbers and locations available input pins which support this type of interrupt. A chart showing which pins are available on which Arduino platforms can be found in the section 1.1.4. IRrecvPCI Class. These interrupts are all handled by the built in “attachInterrupt()” function. It is possible if to use hardware interrupts to detect pin changes on a variety of other input pins however this involves much more complicated setup procedure that directly manipulates various processor pork registers. We have decided not to implement such an extent capability.

The IRrecvLoop receiver class uses no interrupts or timers so it allows you to get away from such hardware dependency. It is not possible to write a looped version of IRfrequency that is fast enough to measure frequency with reasonable accuracy. Therefore the only way to measure input frequency is by use of the pin change interrupts.

Another area of hardware dependency is the use of built in hardware timers. On the receiving end, the original IRrecv receiving class uses the hardware timer to generate an interrupt every 50 µs. The interrupt service routine then polls the input pin to determine its high or low state. While any input pin can be used for this purpose, you must specify an available hardware timer to generate this timed interrupt.

Atmel processors like those used in Arduino boards can have up to six hardware timers numbered 0 through 5. Note that some processors have a special high-speed version of TIMER 4 which they call TIMER 4HS. TIMER 0 is always reserved by the system for the “delay()”, “micros()” and “mills()” functions so it is never used. This library attempts to detect the type of processor you are using and it selects a default hardware timer.

In addition to using a hardware timer to generate the 50 µs interrupts on the IRrecv receiver class, the same timer is used to control the frequency of PWM output for sending signals. We send a PWM signal to an output IR LED or an output driver circuit. While most uses of PWM signals on Arduino platforms have a fixed frequency and a variable duty cycle, our needs are for a fixed duty cycle of about 33% on with a variable frequency. There are no built-in Arduino functions for manipulating the frequency so we have to modify various hardware registers directly. We need to use frequencies ranging from around 30 kHz to 57 kHz. The most accurate way to generate these frequencies is with a hardware timer.

Because the hardware timer driven PWM features are limited to specific output pins, you have a limited choice of pins to use for output. We can overcome this limitation using a method called “bit-bang”. Bit-bang software sits in a tight loop and directly turns the output pin on and off. This allows you to use any available output pin. There are limitations to the bit-bang method which we will discuss later.

The original IRremote library upon which this library is based only supported Arduino Uno and similar devices which had the hardware TIMER 2 available. Later more platforms and timers were added in the following branch of the original library.

https://github.com/TKJElectronics/Arduino-IRremote/

The support for these additional timers and platforms became the basis of the IRLibTimer.h file of this library. Although we had not personally tested all the platforms supported in this file, we were confident they should work because they were almost an exact copy of the branch which originally implemented them. However the changes which were made to separate the use of receiving timers from sending timers have not been tested on all of the platforms mentioned in this file. There is a greater possibility these modifications could have broken something no matter how careful we were in making these changes. We are especially interested in hearing from users of various Teensy platforms to ensure that our changes have not adversely affected your use of the code.

The table below shows the output pin number that you should use for output based upon the platform, chip and timer. If a cell is blank it means that that particular timer is not available on that chip.

Platform	Chip		Timer				
		1	2	3	4	4HS	5
Arduino Mega	ATmega1280 & ATmega2560	11	9*	5	6		46
Arduino Leonardo	ATmega32U4	9*		5		13	
Arduino Uno and older	ATmega328	9	3*				
Teensy 1.0	AT90USB162	17*					
Teensy 2.0	ATmega32U4	14		9		10*	
Teensy++ 1.0 & 2.0	AT90USB646 & AT90USB1286	25	1*	14			
Saguino	ATmega644P & ATmega644	13	14*				
Pinoccio	ATmega256RFR2			D3*			

Entries marked with * are the defaults. Note you will probably not need to change any of the timer specifications unless you need to use particular pins or unless you have a conflict with another library which makes use of hardware timers. For example the default timer for Arduino Leonardo is TIMER1 which also happens to be used by the Servo library. In such a circumstance you would need to change to TIMER3 or TIMER4_HS.

1.4.2 Changing Defaults

If you are using one of the supported platforms and can use the default output specified in the table above, then you can skip to section 1.4.3 and following which discusses schematics for IR LED drivers and the connection of IR receivers. However if you want to use a different timer, different output pin, or enable bit-bang output then continue with this section of the documentation which discusses how to change the defaults for timers and interrupts.

As mentioned previously, typically both the IRrecv 50 µs interrupt timing and the PWM output timing make use of the same hardware timer. However the implementation of bit-bang meant that we had to split these two functions. While the default still remains that they will use the same timer, it is theoretically possible use different hardware timers for input and output. However given that timers are such a limited resource (some platforms only have one or two) it is unlikely and we have not tested it.

We will begin by setting the IR_SEND_TIMERxx definitions. The IR_RECV_TIMERxx will be automatically set based on the sending timer. We will then show you how to override that automatic selection if you wish to do so. We will also show you later how to specify bit-bang sending side of timer sending. However because the selection of the send timer also controls the selection of the receiving timer, you should still choose a sending timer even if you're going to override with bit-bang.

You can change the default IR_SEND_TIMERxx by commenting or uncommenting the proper lines in IRLibTimer.h. The section which allows you to change timers begins at approximately line 50. It attempts to detect your Arduino platform. For example if you have an Arduino Mega and want to use something other than the default TIMER 2 you would look for this section of the file…

```
/* Arduino Mega */
#if defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)
      //#define IR_SEND_TIMER1      11
      #define IR_SEND_TIMER2        9
      //#define IR_SEND_TIMER3      5
      //#define IR_SEND_TIMER4      6
      //#define IR_SEND_TIMER5      46
```

You would put double slashes in front of the line for TIMER2 and remove the double slashes from in front of one of the other timer definitions. The number specified in the define is the pin number that you must use for output. While you can choose which of the supported timers you which to use by adding or removing slashes, do not change the numbers because they are required for those particular timers.

If you are modifying this library for unsupported platforms, you should attempt to implement hardware detection code in this section and specify the parameters for your platform here.

If you intend to use the same timer for input as for output, then you need do nothing else. The code will automatically set various IR_RECV_TIMERxx definitions for you. If you wish to override which timer is used for receiving and make it different than the sending timer, look at the IRLibTimer.h at approximately line 125 for the following section.

```
//#define IR_RECV_TIMER_OVERRIDE
//#define IR_RECV_TIMER1
//#define IR_RECV_TIMER2
//#define IR_RECV_TIMER3
//#define IR_RECV_TIMER4
```

```
//#define IR_RECV_TIMER4_HS
//#define IR_RECV_TIMER5
```

You should un-comment the first line to specify that you want to override and you should un-comment one and only one of the other lines. Note that there is no testing done to ensure that one of these timers is available on your particular platform. You need to look at the section for various IR_SEND_TIMERxx specifications or look at the chart in the previous section to see what is available for your platform. Use at your own risk.

An alternative to using a hardware timer to control the output PWM frequency is the previously discussed method known as “bit-bang”. We require frequencies from approximately 30 kHz to 57 kHz. Because we are dealing with relatively high frequencies, this method is very difficult to implement and is not as accurate as using the built in hardware timers. Most IR protocols specify frequencies within 0.1 kHz such as NEC protocol which uses 38.4 kHz. However even when using the built-in timer hardware, we only support integer frequencies so we rounded that down to even 38.0 kHz.

The bit-bang implementation is less accurate than that. Our limited tests show that our bit-bang implementation might produce results as much as +/-5% off of the target value. However our experience is that the specification of frequency isn’t that critical. We were still able to control TVs, VCRs, and cable boxes using bit-bang. We also discovered there are numerous factors such as hardware interrupts that can interfere with your results. We recommend you only use bit-bang on unsupported platforms or if appropriate timers and PWM pins are unavailable.

You specify use of bit-bang output in the file IRLibTimer.h in a section at approximately line 100. You will find the following two lines of defines.

```
//#define IR_SEND_BIT_BANG  3
#define IR_BIT_BANG_OVERHEAD 10
```

If the IR_SEND_BIT_BANG definition is commented out as it is shown here, then the library will use hardware timer driven PWM by default. If you remove the slashes to un-comment the definition, it will use bit-bang output on the pin number that you specify. In this example the default is pin number 3. The other definition is a timing “fudge factor” that you may need to specify. You need not comment it out when not using bit-bang. You can leave it alone.

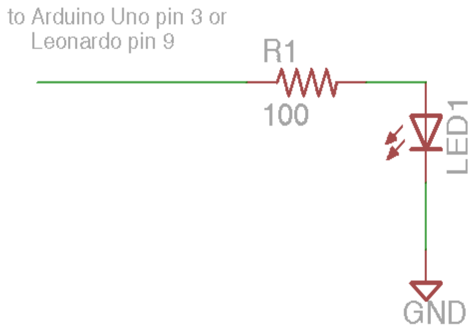
Note that if you use bit-bang sending and IRrecvLoop for receiving you will not need any hardware timers or hardware interrupts. This may be especially useful for unsupported platforms.

Our implementation of bit-bang output is dependent upon the “delayMicroseconds()” function. If you are porting this code to a different platform, bit-bang will only be as accurate as your limitation of this function. By the way unlike the “micros()” function which is only accurate to 4 µs on 16 MHz processors or 8 µs on 8 MHz processors, delayMicroseconds() attempts to be accurate to 1 µs resolution. That still introduces some granularity into our results.

We want to reiterate that most users will not need to make any changes to IRLibTimer.h as long as they connect their output hardware to a supported pin and they do not have conflicts with other libraries.

1.4.3. IR LED Driver Components and Schematics

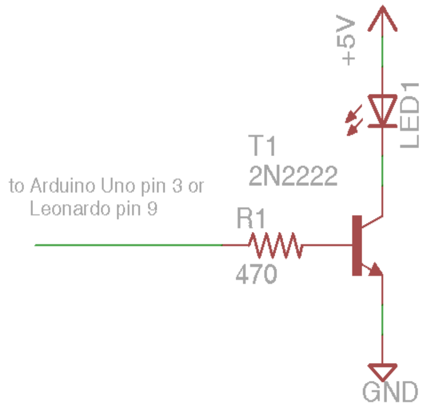
The simplest output circuit is simply connect the IR LED directly to the output pin of the Arduino and then connect it to +5 volts with a current limiting resistor of 100 ohms like this.



Make sure you get the polarity of the LED correct. The shorter of the two leads should connect to ground. The longer lead connects to the resistor which in turn connects to the Arduino.

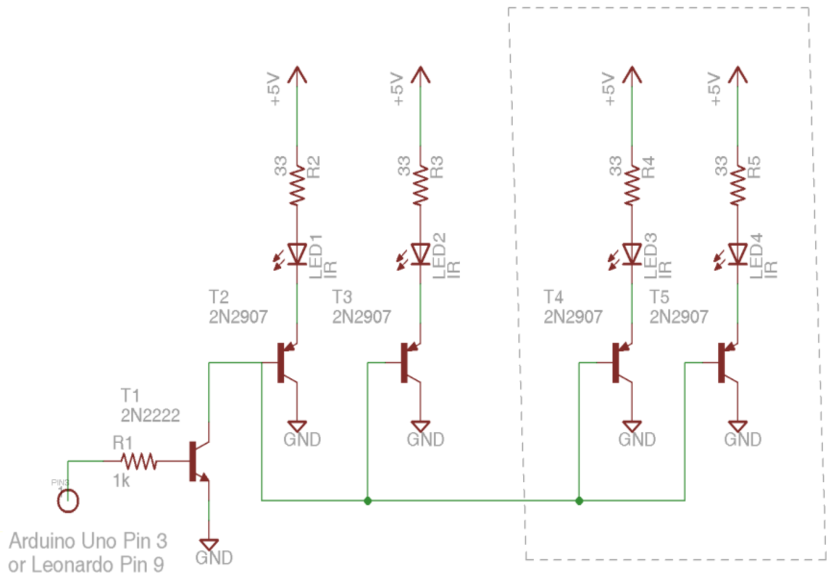
Note that all of our examples presume +5 volt supplies however some Arduino systems run at 3.3 volts. These schematics should also work at 3.3 volts however you might want to slightly lower some of the resistor values.

The output pins of an Arduino cannot supply much current. This is especially true of some newer Arduinos based on 16 and 32 bit ARM processors rather than the traditional 8-bit AVR processors. A better solution is to use a driver transistor. The schematic below shows a 2N2222 transistor but any similar NPN transistor should work. The base of the transistor is connected to the output of the Arduino using a 470 ohm resistor. The emitter is connected to ground that the LED is connected between the +5V and the collector.



Note that the current passing through the LED will in all likelihood exceed the maximum continuous current rating of the LED. However in our particular application we have a modulated signal sending a sequence of pulses that only last a few milliseconds total. As long as you're not sending a continuous signal, the circuit will work fine. Occasionally you will make up a special hardware board which includes both output and input portions but for a particular application you would be only using the input portion. Theoretically you could have an output pin accidentally left on continuously and it would burn out your LED. If you have an application that does input only but you have connected both input and output circuits you should use the IRrecv::No_Output() method in the setup portion of your sketch.

I have had good success with single transistor and single LED circuits over moderate distances but if you really want power you can use multiple LEDs with multiple driving transistors. The schematic below is loosely based on the output portion of the famous TV-B-Gone device with its four output LEDs. Note that the transistors and LEDs on the right half of the schematic can be eliminated if you want a double transistor and double LED circuit instead of quadruple.

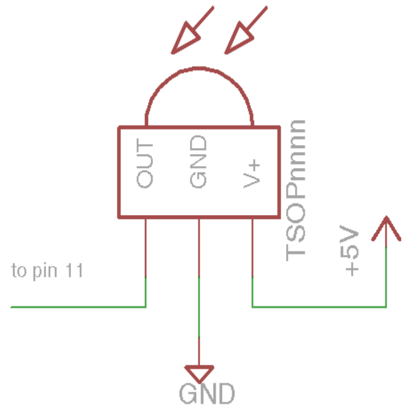


The circuit starts with an NPN transistor connected to the Arduino output pin via a 1K resistor. That NPN then drives up to four PNP transistors that drive the LEDs. Note that we have added a 33 ohm resistor to each LED to limit the current. This resistor was added because I designed the circuit to be used with an IR remote control toy helicopter that would continuously send signals to the helicopter. If your application only has intermittent signals you can eliminate those 33 ohm resistors. Again any general purpose PNP switching transistor similar to the one in the schematic should work okay.

IR LEDs come in different varieties. Some are narrow angle and some are wide-angle. We happen to like the IR-333-A Everlight which is a 20° narrow angle LED available here from Moser Electronics. For a wide-angle LED consider the IR333C/H0/L10 Everlight which has a 40° viewing angle also available here from Moser Electronics. Similar devices are available from Adafruit and RadioShack. The one from Adafruit was 20° but the RadioShack model did not specify the angle. I like to use a combination of narrow angle and wide-angle LEDs. Either use one each in a two LED application or two each in a four LED application.

1.4.4 IR Receiver Components and Schematics

The schematic for a receiver connection is much simpler than the driver circuit. You simply connect power and ground to the device and connect the output pin of the device to an input on your Arduino.



Note that all of our examples presume +5 volt supplies however some Arduino systems run at 3.3 volts. These schematics should also work at 3.3 volts. It is critical that you do not use a 5 volt supply on your receiver if you are connecting to a 3.3 volt system.

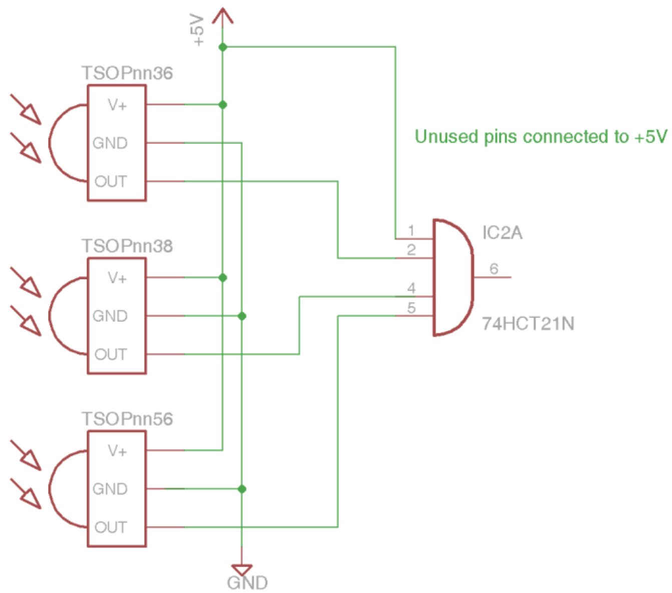
For ordinary receiving, any unused digital input pin can be used. Traditionally we use pin 11 but any pin should work. For determining frequency, you are limited to pins which have been change interrupts available. See the section on the IRfrequency class for details.

While the schematic is trivial, the challenge is finding the proper device for receiving. Although we deal with IR signals and being square waves with pulses varying from a couple of hundred milliseconds up to a few thousand milliseconds, in fact those pulses are actually modulated at a frequency from somewhere between 30 kHz and 58 kHz. Different protocols use different frequencies. The TSOP series of devices from Vishay are generally used. They deem modulate this signal using a bandpass filter and automatic gain control. There is a 4 or 5 digit number after the TSOP designation. The final two digits designate the frequency. The next most significant two digits describe the type of package. A most significant fifth digit may describe the type of AGC. Here is a link to a selector guide from Vishay in PDF format. http://www.vishay.com/doc?49845

Typically you would use through hole packaging such as the TSOP44nn, TSOP84nn, or TSOP14nn. The frequency in the last two digits depends on the protocol you wish to use. Most hobbyists use 38 kHz such as TSOP4438. Frequencies outside the range 36-40 kHz are rare so a 38 kHz device is a good average among the most common protocols. Although they sell devices specifically to two different frequencies, the bandpass filters devices are centered around the specific frequency but are generally wide enough to allow a reasonable range of frequencies. One notable exception is the Panasonic_Old used by Scientific Atlantic and Cisco cable boxes. They are used mostly by Time Warner and Bright House cable systems. These systems use 57 kHz and on occasion 38 kHz receivers have difficulty detecting those signals. We have successfully used a part from RadioShack as seen here http://www.radioshack.com/product/index.jsp?productId=2049727

That particular part when received from them does not look like the photo on the website. It does not include the connector bracket that is depicted. It is described as a 38 kHz device and the packaging looks like the TSOP4438 but RadioShack does not provide a manufacturers part number. They only provide their catalog number 276-640. We have successfully used the RadioShack device at frequencies from 36 kHz to 57 kHz which is the entire range needed. A similar part from Adafruit Industries http://www.adafruit.com/products/157 designated as a TSOP38238 did not work at 57 kHz but if you do not need that higher frequency, it works quite well.

If you are having difficulty receiving 57 kHz signals with a 38 kHz device, you could purchase multiple receivers each tuned to a different frequency. Similarly you might have a device which is receiving signals from different directions. Perhaps you have a robot and you want to put three receivers equally spaced around the outside of the robot that could receive IR signals from any angle. For whatever reason that you might want multiple receivers, a schematic such as the one shown below to be used to connect them.



You would connect the output pins of each receiver to a logical AND gate. Here we are using a 74HCT21N quad input AND gate. You could use a 74HCT08N dual input that come 4 to a package. The 74HCTxx series is a CMOS chip that will operate at 5V TTL compatible levels suitable for use with Arduino at 5V. If you have any unused input pins on the gate you should tie them to +5.

You may wonder why we are using an AND gate when we want to receive a signal anytime any of the input devices are receiving a signal. You might think we want a logical OR gate. However the devices are normally high output and go low when a signal is received. Similarly our library is expecting active low signals. The logical equation says that NOT(NOT(A) OR NOT(B))=A AND B. Thus we use an AND gate.

There is one additional type of receiver we might want to use. The IRfrequency class which is used to detect modulation frequencies requires a different type of receiving device. The TSMP58000 device (note that is TSMP not TSOP) device receives raw signals. It does not have a bandpass filter tuned to a particular frequency and does not have automatic gain control. This type of devices generally described as an “IR learner” instead of “IR receiver”. It can be used to detect the modulation frequency of an incoming signal. This device must be connected to a hardware interrupt pin such as those used with the IRrecvPCI receiver class. See the section on the IRfrequency class.

2. Tutorials and Examples

When this section is fully implemented it will provide tutorials on how to use this library and explanation of all of the example programs in the examples directory. For now we will simply include links to some tutorials we have already posted in our blog on this website. Additional tutorials and examples will appear here eventually.

- IRLib Tutorial part 1: Hardware set up – 03/14/2013
- IRLib Tutorial part 2: Controlling a Servo Using an IR Remote – 03/14/2013
- IRLib Tutorial part 3a: Sending IR Codes – 04/22/2013
- IRLib Tutorial part 3b: Creating a Virtual Remote Using Python – 05/04/2013
- IRLib Tutorial part 3c: Python, PySerial and PyGame Installation on Windows – 05/30/2013
- IRLib Tutorial part 4: IR Remote Mouse – 06/10/2013

3. Implementing New Protocols

This section not yet implemented.