# User's Manual for IRLib2.

## A Library for Receiving, Decoding and Sending Infrared Signals Using Arduino.

(September 2016)

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 an earlier library called IRremote which was created by Ken Shirriff. IRLib version 1.x was a major rewrite of that system. IRLib2 is a significant rewrite from IRLib 1.x. IRLib2 it is not fully backwards compatible with IRLib1 however it should take minimal effort to update any programs to use the new library. This is especially true if you are only using the interface of the library and not relying on any internals. See Appendix A for a discussion of the differences between IRLib2 and IRLib1 and an explanation of the reason for the choices I made in the rewrite. In this document any mention of IRLib should be construed to mean the current IRLib2 unless otherwise specifically specified. Also throughout the document we will make note of a few of the specific differences between IRLib2 and IRLib1.

One of the main differences we will note here is that IRLib1 was a single library with a single header file that included everything. You would make use of it by inserting the line…

```
#include <IRLib.h>
```

However IRLib2 is more modular in design. It is actually organized into 6 different folders and many multiples of header files. To make the most efficient use of the libraries you should include only those header files which you actually need. However to assist in quickly converting older sketches to the new system, you can do…

```
#include <IRLibAll.h>
```

This will include the entire package into your sketch including all receivers and all protocols. While the linking loader will eliminate some portions of the library that you do not actually use, it will not eliminate everything that is unused. Additionally there is another file…

```
#include <IRLib2.h>
```

This will include only the original seven protocols supported by IRLib1. The use of these all-inclusive include files is not recommended if you are concerned about the size of your code. We will use it occasionally in sample sketches just to simplify things. We will later describe the procedure for including only the specific pieces of the code that you actually need to use.

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.

There are also three appendices to this document. Appendix A as mentioned we explains the differences between IRLib2 and IRLib1 and gives an explanation of why we made the changes. Appendix B describes a special notation called IRP notation that is used in some online documentation to describe a particular IR protocol. Appendix C provide some guidance about our coding standards so that if you contribute to the library you can help maintain the same look and feel.

Note also that the code is well commented 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 the following sections:



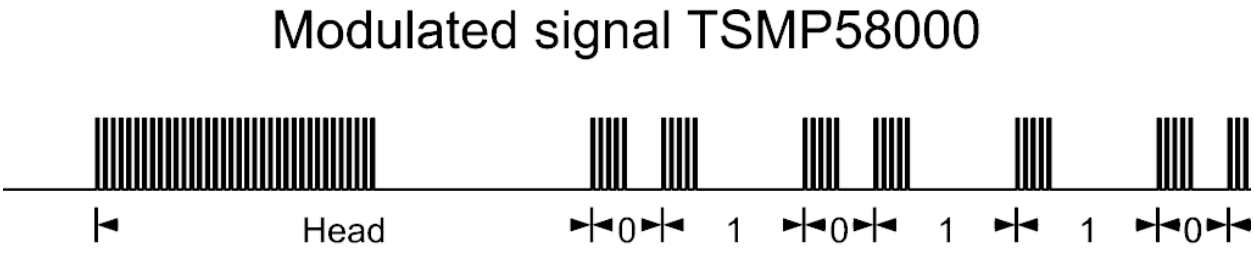
## 1.1 Receiver Classes

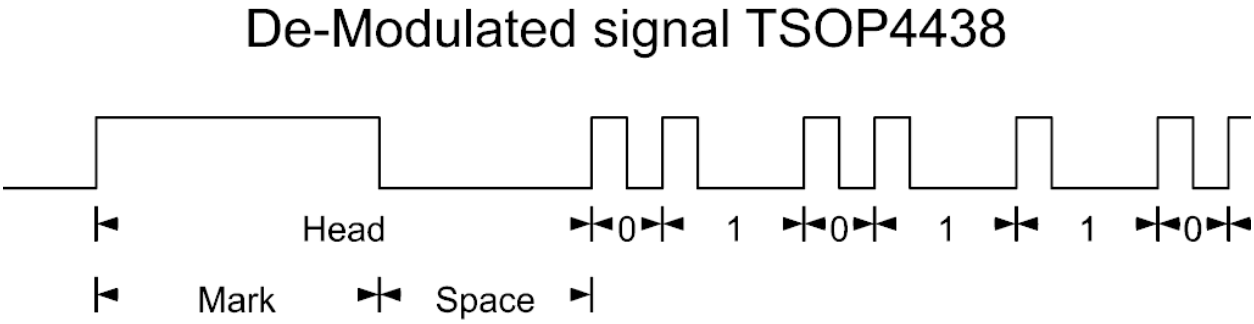
IR data is received on an input pin of the Arduino as a square wave of pulses of varying lengths. The length of the on and off pulses encodes the data. It is the job of the receiver class to record the timing of the pulses and the spaces between them. The data is stored in an array and passed to the 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 records basic pulse widths, there is an 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 TSOP4438 or TSOP58438 or similar. See the section 1.5 Hardware Considerations 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. If you want to measure the frequency of a modulated signal, you will need a device such as a TSMP58000 which does not filter the signal.

The output of the receiver is connected to one of the digital input pins of your Arduino. It is the job of the receiver class to monitor this signal and to determine the length of the pulses known as “marks” and the length of the intervening idle periods known as “spaces”. The hardware receiver devices typically are normally high signal and then go low when a signal is received. However the code compensates for this so it 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). Microseconds are often abbreviated as "µs". Note however that the software makes use of the built-in Arduino function “micros(void)” which returns results in four microsecond granularity. This 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 and three derived classes. Additionally there is a frequency measuring class is a library of its own. 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
- IRLibFreq Measures the modulation frequency using TSMP58000

The code for the base class and the IRrecvLoop can be found in the IRLib2 folder and the code for the other two derived receiver classes and frequency class each have their own folder. To use one of the receivers, you will have to include its header file in your sketch. You do not need to include the base class header file. You pass the pin number to which you receiver

device is connected in the constructor of the object. For example to use the IRrecv receiver you would do the following…

```
#include <IRLibRecv.h>
IRrecv myReceiver(2);//create instance of receiver using pin 2
```

Similarly for the PCI version you would do…

```
#include <IRLibRecvPCI.h>
IRrecvPCI myReceiver(2);//create instance of receiver using pin 2
```

And finally the "loop" version would be…

```
#include <IRLibRecvLoop.h>
IRrecvLoop myReceiver(2);//create instance of receiver using pin 2
```

You can name the receiver object anything you want but to simplify our explanations we will presume that you have created a receiver object named "myReceiver". 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. This is because all of the classes share one set of global data and methods in common. The data must be declared globally in the library because it is accessed by interrupt service routines (called ISR routines). The structure of an interrupt service routine does not allow for the passing of any parameters. Therefore any data that the ISR accesses must be global. It is possible to create both a receiver object and the frequency detection object but you should have no more than one of each.

The data is stored in an array of uint16_t 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. This means that 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. It does nothing until you call getResults(). It then sits in a tight loop and does not release control back to you until the complete signal has been received.

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 <IRLibRecv.h>
```

```
IRrecv MyReceiver(2); //Create the receiver. Use pin 2

#include <IRLibDecodeBase.h>
#include <IRLib_P01_NEC.h>
#include <IRLibCombo.h>
IRdecode MyDecoder;

void setup() {
  Serial.begin(9600);
  MyReceiver.enableIRIn();//start receiving
}

void loop() {
  if (MyReceiver.getResults()) {//wait till it returns true
    MyDecoder.decode();
    MyDecoder.dumpResultso();
    MyReceiver.enableIRIn();    //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 code is implemented in IRLib2/IRLibRecvBase.cpp, and the prototype in IRLib2/IRLibRecvBase.h is…

```
class IRrecvBase {
public:
  IRrecvBase(void) {};
  IRrecvBase(uint8_t recvPin);
  bool getResults()(const uint16_t timePerTicks=1);
  virtual void enableIRIn(void);
  virtual void disableIRIn(void);
  void enableAutoResume(uint16_t *P);
  void setFrameTimeout(uint16_t frameTimeout);
  void blink13(bool enableBlinkLED);
  uint16_t markExcess;
protected:
  void init(void);
};
```

#### 1.1.2.1 Constructor IRrecvBase

The prototype for the constructor is…

```
  IRrecvBase(uint8_t recvPin);
```

The value passed is the pin number to which the receiver is connected. Note: previous versions of the constructor for IRrecvPCI were passed the interrupt number instead of the pin number but now all receiver classes consistently have the pin number as its only parameter.

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

#### 1.1.2.2 Method enableIRIn

The receiver does not begin recording data until you call this method. Its prototype is…

```
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().

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 getResults().

After you have finished processing the received data, you need to call enableIRIn() again to tell the receiver that you want more data. Even if you have enabled the auto-resume feature, you still need to call this method when you are ready for more data so that the receiver knows that it is safe to reuse the buffer. We considered renaming this method to "needMoreData" but in the long run its current name is more accurate. A complete discussion of the auto resume feature can be found in the section 1.1.2.4 Method useAutoResume below.

> NOTE: In IRLib1 there were 2 methods for starting the receiver. The method "myReceiver.enableIRIn()" was used to initialize the receiver and the method "myReceiver.resume()" was used to restart the receiver after you had finished processing the data. However if you also use the sending class it would disable input because it uses the hardware timers for different purposes. Therefore you would have to use "enableIRIn()" to restart after sending. It got too confusing to remember whether you needed to use "resume()" or "enableIRIn()" so we just eliminated the "resume()" completely. That method no longer exists in IRLib2. Now you just use "enableIRIn()" whether you are initializing for the first time or restarting. See Appendix A for more details.

#### 1.1.2.3 Method getResults

After you have initialized receiving with enableIRIn you need to repeatedly call getResults() to see if a complete sequence has been received. The prototype for this method is…

```
bool getResults(const uint16_t timePerTicks=1);
```

This method will return “true” when a complete sequence has been received and will return “false” otherwise. The optional parameter 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 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 and do not need converting. This parameter is only used by the base class and is not necessary or present in any of the actual derived classes you will use.

When a complete sequence has been received by the class it does not continue recording signals. It also does not normally 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 again call "enableIRIn()".

#### 1.1.2.4 Method enableAutoResume

All three receiver classes capture their data in a globally declared buffer we will describe as the "receiver buffer". The decoders access that data through a pointer which points to what we will describe as the "decoder buffer". Normally the decoder buffer pointer points to the receiver buffer so they essentially same buffer.

If the receiver would continue recording the next frame of data before you had finished processing the first one, it would get overwritten. In some instances however you want it to resume receiving as quickly as possible so you do not miss the next frame while you are processing the first one. You can however declare an external buffer for the decoder to use and tell the receiver that it should automatically copy the data from the receiver buffer to your external decoder buffer and automatically resume. This is especially useful to use this external buffer and auto resume whenever you are using the IRrecvPCI receiver. That receiver has difficulty determining that the first frame has completed unless a secondary frame has begun. We highly recommend the use of this auto resume feature whenever you are using the IRrecvPCI receiver. Of course this feature requires 200 bytes of RAM memory which is very scarce on most Arduino platforms so you may not be able to use this feature if you cannot afford to declare the extra buffer space.

Note that this method is not available when using the IRrecvLoop receiver class because that class is not interrupt driven. Therefore you cannot multitask by receiving signals while you are decoding a previous sequence.

The prototype for this method is…

```
  void enableAutoResume useExtnBuf(uint16_t *P);
```

Below is an excerpt of the autoResume sample sketch that shows how to use this method. You declare the buffer and then invoke the method in your setup function prior to starting the receiver with enableIRIn(). When a sequence is complete and getResults() returns true, the data will be copied to "myBuffer" and it will automatically start recording the next available sequence even though you have not yet invoked enableIRIn().

Whether using auto resume or not, in your main loop do not re-invoke enableIRIn() until after you have processed all of the data. If by chance you have not finished processing the first frame when the second one is completed, the receiver will not auto resume again until you have invoked enableIRIn(). Essentially invoking that method tells it that it is safe to write into the decoder buffer.

```
uint16_t myBuffer[RECV_BUF_LENGTH];

void setup() {
  //Enable auto resume and pass it the address of your extra buffer
  myReceiver.enableAutoResume(myBuffer);
```

```
  myReceiver.enableIRIn(); // Start the receiver
//… Other initialization code here…
}
void loop() {
  //Continue looping until you get a complete signal received
  if (myReceiver.getResults()) {
    myDecoder.decode();           //Decode it
    myDecoder.dumpResults(true);  //Now print results
    myReceiver.enableIRIn();      //Tell receiver we are ready
  }
}
```

You should only invoke enableAutoResume one time in your setup function and it must be before enableIRIn().

> NOTE: In IRLib1 there was a decoder method "useExtnBuf(void *P)" that would allow you to manually resume receiving before you were done processing the data. It did not however solve the problem with IRrecvPCI possibly missing the secondary frame. When we first implemented auto resume, we implemented was an option in the useExtnBuf method however was extremely confusing when you should call enableIRIn. If you are using external buffer but using auto resume then you would reenable before decoding. But if you are using auto resume you would reenable after decoding. We eventually concluded that auto resume was so useful that there was no reason to use an external buffer without it. It also made more logical sense to make this a method of the receiver class than the decoder class. It isn't so much about adding an external buffer to the decoder but it is more about changing the behavior of the receiver. See Appendix A for more details.

Many thanks and acknowledgment to contributor Gabriel Staples who contributed auto resume code that was inspiration for our implementation.

#### 1.1.2.5 Method setFrameTimeout

The receiver measures the length of marks and spaces and when it finds an especially long-duration space it assumes that is the end of the frame of data. This time period is called the "frame timeout" value. Any space which is found to exceed that value is presumed to be the end of frame and a new frame begins with the next mark. The default value is specified in IRLib2/IRLibRecvBase.h as follows…

```
#define DEFAULT_FRAME_TIMEOUT 7800
```

If you wish to change the value for a particular application sketch you can do so with the following method…

```
void setFrameTimeout(uint16_t frameTimeout);
```

We chose the default value to be 1.25 times the largest space any valid IR protocol might have. Our research shows that in Ken Sherriff's IRremote library ir_Dish.cpp they use DISH_RPT_SPACE 6200 while another reference says about 6000. If we take 6200*1.25= 7750 rounded up we will use 7800. Previously IRLib1.x the default value was 10000 which was

probably too large. Thanks to Gabriel Staples for this note. We had to implement this is a method because the IRrecv class needs to convert this value into a number of ticks.

#### 1.1.2.6 Method blink13

For debugging purposes you may want to know if your Arduino is receiving a signal. If you call this method then pin 13 will be blinked on every time a mark is received and blinked off with a spaces received. The default is off. The prototype is…

```
void blink13(bool enableBlinkLED);
```

#### 1.1.2.7 Method disableIRIn

Even when not receiving data or waiting to receive data, the ISR may remain active but remains in a do-nothing state. If you want to truly shut down the ISR you can call this method. The derived method should disable the ISR and then call this base method to then turn everything off. The prototype is…

```
  void disableIRIn(void);
```

#### 1.1.2.8 Variable markExcess

There is one value in the class that you can change directly. It is…

```
  uint8_t markExcess;
```

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. IRLib1.x used that value however my own experience is that 50µs is a better value so that is the default for IRLib2. You can change that value based on your own experiences by changing the value for example…

```
myReceiver.markExcess= 75;
```

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

#### 1.1.2.9 Protected Method init()

There is one protected method of the base class which is used for internal use.

```
void init(void);
```

Because it is protected you cannot call it in your sketch and you would have no need to do so.

### 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 file IRLibProtocols/IRLibHardware.h. Each type of Arduino platform such as Arduino Uno, Leonardo, Mega etc. has a choice of different timers. For example the Uno defaults to TIMER2 while the Leonardo defaults to TIMER1 because it does not have a 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 IRLibHardware.h to use a different timer.

The prototype of the class is in IRLibRecv/IRLibRecv.h and the code is implemented in IRLibRecv/IRLibRecv.cpp. This makes this class essentially a library of its own. If it had been included in the IRLib2 folder, it's globally declared ISR might conflict with other libraries even if you did not create an instance of this class. The prototype is…

```
class IRrecv: public IRrecvBase {
public:
  IRrecv(uint8_t recvPin):IRrecvBase(recvPin){};
  void enableIRIn(void);
  bool getResults(void);
  void disableIRIn(void);
  void setFrameTimeout(uint16_t frameTimeout);
};
```

Of course this class also inherits other methods from the base class that are not listed in this prototype. 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.

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 enableIRIn() method again. Do not call enableIRIn() until you are finished with the data from the previous frame even if you have enabled auto resume.

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 as well as the IRfrequency code described in the next section is loosely based upon and inspired by work by the developers of the AnalysIR program. AnalysIR is a Windows-based application which allows you to graphically analyze IR input signals through an 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 and therefore jumbled results.

This problem can be alleviated by using the auto resume feature however that requires an external buffer which can use up a considerable amount of valuable RAM memory. Auto resume will ensure that 2 frames that arrive close together will be properly received and decoded so it is highly recommended when using this type of receiver.

As with the IRrecv class, the interrupt service routine is put into an idle state once a completed sequence has been detected and is not re-enabled until you call enableIRIn().

The prototype of the class is in IRLibRecvPCI/IRLibRecvPCI.h and the code is implemented in IRLibRecvPCI/IRLibRecvPCI.cpp. This makes this class essentially a library of its own. If it had been included in the IRLib2 folder, it's globally declared ISR might conflict with other libraries even if you did not create an instance of this class. The prototype is…

```
class IRrecvPCI: public IRrecvBase {
public:
  IRrecvPCI(uint8_t pin);
  void enableIRIn(void);
  bool getResults(void);
  void disableIRIn(void);
private:
  uint8_t intrNum;
};
```

Of course this class also inherits other methods from the base class that are not listed in this prototype. 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.

> NOTE: In IRLib1 the constructor for this class was passed the interrupt number and not the pin number. This behavior has changed to be pin number. It then converts that pin number into the interrupt number using the built-in Arduino function "digitalPinToInterrupt(pin)" as is recommended at https://www.arduino.cc/en/Reference/AttachInterrupt See that page for more details.

Only certain particular pins support the pin change interrupt depending on which type of Arduino you are using. You can verify that the pin is supported by putting the following code in your setup function.

```
#define MY_PIN 2   //pin number 2
if(digitalPinToInterrupt(MY_PIN)==NOT_AN_INTERRUPT) {
  Serial.println(F("Invalid frequency pin number."));
  while (1) {};//infinite loop because of fatal error
}
```

The table below is a partial list of the pin numbers that are available on various platforms. The table was adapted from http://arduino.cc/en/Reference/AttachInterrupt and you should reference that page for more information about the Arduino built in function attachInterrupt() that is used by this class.

| Board | Digital Pins Usable For Interrupts |
|---|---|
| Uno, Nano, Mini, other 328-based | 2, 3 |
| Mega, Mega2560, MegaADK | 2, 3, 18, 19, 20, 21 |
| Micro, Leonardo, other 32u4-based | 0, 1, 2, 3, 7 |
| Pinoccio | 4, 5, SCL(15), SDA(16), RX1(13), TX1(14), 7 |
| Zero* | all digital pins, except 4 |
| MKR1000 Rev.1* | 0, 1, 4, 5, 6, 7, 8, 9, A1, A2 |
| Due* | all digital pins |
| 101* | all digital pins |

*These platforms have not been tested with IRLib but should theoretically work with this receiver. They are not however supported for sending IR signals or using the IRrecv receiver 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 this receiver is interrupt driven and you should be able to do just about anything inside your main loop, we recommend that you call getResults() as frequently as possible because this receiver has difficulty detecting when a frame is complete. If however you are using the auto resume feature, this restriction is not necessary. You can take considerable time doing other things in your main loop when using auto resume. We recommend that if you can afford to allocate an external buffer and use auto resume that you should do so with this receiver for best results.

A reminder that when a complete stream has been received, no additional measurements are taken until you call the enableIRIn() method again. Do not call enableIRIn()

until you are finished with the data from the previous frame even if you have enabled auto resume.

### 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 to 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. This class cannot make use of the auto resume feature because it is not interrupt driven and therefore cannot multitask.

You will still need to call enableIRIn() when you have finished processing your data to get the receiver ready to receive again. However the actual data collection will not begin again until you call getResults().

The prototype of the class is in IRLib2/IRLibRecvLoop.h and the code is implemented in IRLib2/IRLibRecvLoop.cpp. Unlike the IRLibRecv and IRLibRecvPCI, this receiver should never conflict with any other libraries. For that reason it does not need to be a library unto itself. The linking loader will eliminate it completely if you do not create an instance. Therefore the code files are located in the IRLib2 folder. The prototype is…

```
class IRrecvLoop: public IRrecvBase {
public:
  IRrecvLoop(uint8_t recvPin):IRrecvBase(recvPin){};
  bool getResults(void);
};
```

Although the prototype only lists the constructor and getResults(), all of the methods of the base class are technically available. However as we mentioned earlier, auto resume is not available so enableAutoResume() does nothing. Similarly because it is not interrupt driven the method disableIRIn() does nothing. Like the other receivers, the value passed to the constructor is the number of the receiver pin. Any available digital input pin can be used.

Because this class uses no hardware timers and no hardware interrupts it is completely hardware independent and should work on any Arduino-like platform even those which are not technically supported by this library.

### 1.1.6. Global Receiver Structures and Functions

There are 3 additional global functions in IRLib2/IRLibRecvBase.h and implemented in IRLib2/IRLibRecvBase.cpp. These functions are not methods of any class. Two of these are

global because they are referenced internally by the ISR and you cannot pass variables or objects to an ISR. The third of these global functions is somewhat related to both sending and receiving but doesn't really belong in either class. In this section we will also describe a structure containing other global variables that were again made global so that they can be referenced inside the ISR.

#### 1.1.6.1 Function IRLib_noOutput

Some users create custom boards for input and output of IR signals and those boards are connected to their Arduino even in the case when the sketch 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 this function. We are unsure if this function is actually necessary but we provide it out of an abundance of caution. The prototype is…

```
void IRLib_noOutput(void);
```

NOTE: This previously was a method of the base receiver class but it really doesn't have anything to do with receiving so we renamed it and made it a standalone function.

#### 1.1.6.2 Function IRLib_doBlink

This function is for internal use only. It is called by the ISR or in the case of the loop receiver by IRLibRecvLoop::getResults(). It does actual blinking of the pin 13 indicator LED. The prototype is…

```
void IRLib_doBlink(void);
```

#### 1.1.6.3 IRLib_IRrecvComplete

This function is for internal use only. It is called by the ISR and implements much of the logic of the auto resume feature. It is not used by IRLibRecvLoop::getResults() because that receiver does not support auto resume. The prototype is…

```
void IRLib_IRrecvComplete(uint8_t Reason);
```

This function is called by both the 50us and PCI ISR in one of two circumstances:

1) The SPACE was long enough that we are sure the frame is over and ready to process.
2) The buffer overflowed we have to quit.

The parameter is for debugging purposes only.

#### 1.1.6.4 Global data recvGlobal

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 you create custom decoder classes need access to this

data, you need not deal with the structure. It is primarily for internal use. The type "recvGlobal_t" defined in IRLib2/IRLibGlobal.h as shown below. A single global instance of the structure is declared in IRLib2/IRLibRecvBase.cpp as follows

```
  recvGlobal_t recvGlobal;
```

Additionally the ISR routines keep track of what they're doing by implementing a state machine. The state machine is only used by the IRrecv and IRrecvPCI classes. The current state of the state machine is stored in a field of the global structure of type "currentState_t".

Below is the definition of the states of the state machine, and the 2 structure types that are used by the receivers. The actual code in IRLib/IRLibglobal.h Is more extensively commented than what is shown here. We will briefly describe each field below.

```
// Receiver states. This previously was enum but changed
// it to uint8_t to guarantee it was a single atomic 8-bit value.
#define  STATE_UNKNOWN 0
#define  STATE_READY_TO_BEGIN 1
#define  STATE_TIMING_MARK 2
#define  STATE_TIMING_SPACE 3
#define  STATE_FINISHED 4
#define  STATE_RUNNING 5
typedef uint8_t  currentState_t;

typedef struct {
  uint8_t recvPin;         // pin number for receiver
  bool enableAutoResume;   // flag tells if we should auto resume
  uint16_t frameTimeout;   // set by "setFrameTimeout()"
  uint16_t frameTimeoutTicks;// = frameTimeout/USEC_PER_TICKS

  bool enableBlinkLED;

  volatile bool decoderWantsData; //previous decode is finished.
  volatile bool newDataAvailable; //tells getResults
  volatile bool didAutoResume;    //tells getResults

  volatile uint16_t recvBuffer[RECV_BUF_LENGTH];
  volatile bufIndex_t recvLength;

  volatile uint16_t* decodeBuffer;
  volatile bufIndex_t decodeLength;

  volatile uint32_t timer;     // state timer, counts 50uS ticks
  volatile currentState_t currentState;   // state machine
}
recvGlobal_t;
extern  recvGlobal_t recvGlobal; //declared in IRLibRecvBase.cpp
```

The first four fields of the recvGlobal structure are initialized by the receiver class and once initialize will probably not change. The first three are pretty much self-explanatory. The

field uint16_t frameTimeoutTicks is only used by the 50 µs IR recv class it is always equal to frameTimeout/USEC_PER_TICKS (which is almost always 50).

The next field enableBlinkLED tells whether or not to blink indicator LED.

The remaining fields are declared volatile because they may be accessed both inside and outside the ISR. The next three fields are flags that help facilitate auto resume. The flag decoderWantsData is the way the decoder tells the ISR that it is safe to write new data into the decode buffer. The flag newDataAvailable is the way the ISR tells getResults() that a complete sequence has been received. The flag didAutoResume is set by the ISR and tells IRLibRecvBase::getResults() that because we did auto resume, the ISR has already copied the data from recvGlobal.recvBuffer to recvGlobal.decodeBuffer. However getResults() still needs to perform some math on the data to convert from ticks to microseconds if necessary and to make the "markExcess" adjustments.

The field recvBuffer[RECV_BUF_LENGTH] is the actual receive buffer. The receivers always initially accumulate the timing of the marks and spaces in this buffer. The field recvLength tells the number of entries in that buffer.

The field decodeBuffer is a pointer that the decoders use to access the timing data. If there is no external buffer for auto resume then this pointer will point to recvGlobal.recvBuffer and essentially the decode buffer is the receiving buffer. However if an external buffer and auto resume have been enabled, then this pointer points to that external buffer. In that case the external buffer is the decode buffer. The field decodeLength is the number of entries in the decode buffer.

The field uint32_t timer is a temporary variable used to count the number of ticks or microseconds of the current interval.

The field currentState_t currentState is the current state of the state machine. Its legal values are defined above.

Near the top of IRLib2/IRLibGlobal.h are the following definitions…

```
#define RECV_BUF_LENGTH 100
#if (RECV_BUF_LENGTH > 255)
   typedef uint16_t bufIndex_t;
#else
   typedef uint8_t bufIndex_t;
#endif
```

The #define is the length of the receiver and decoder buffers. The default value of 100 will work for most electronic protocols. However some protocols used by air conditioners and fans have extremely lengthy frame sizes. Because each entry in the buffer is uint16_t it will use two bytes of RAM. You'll want to use as small value as possible. If you need more than 255 entries, the conditional compile directives will make sure that your index into the buffers is sufficient to handle that many entries.

### 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 IRLibFreq/dumpFreq 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 IRLibFreq/freq 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 class is not derived from the IRLibRecv class. It is a library unto itself with the code implemented in IRLibFreq/IRLibFreq.cpp. The prototype in IRLibFreq/IRLibFreq.h is…

```
class IRfrequency {
  public:
    IRfrequency(uint8_t pin);
    void enableFreqDetect(void);
    bool haveData(void);      //detect if data is received
    void disableFreqDetect(void);
    void computeFreq(void);      //computes but does not print
results
    void dumpResults(bool detail);     //computes and prints result
    double results; //results in kHz
    uint8_t samples; //number of samples used in computation
  private:
    volatile FREQUENCY_BUFFER_TYPE Time_Stamp[256];
    uint8_t intrNum;
    uint16_t i;
    uint32_t sum;
};
```

#### 1.1.7.1 Constructor IRfrequency(uint8_t pin)

The parameter to the constructor is the pin number to which you will connect your TSMP58000 device. Note previous versions of this library you would pass the interrupt number rather than the pin number. Only particular pins which can use the pin change interrupt can be used. See section on the IRrecvPCI class for a table showing which pin numbers are available

on a particular platform. There is also sample code that shows how to see if you have chosen a valid pin number.

As with other receiver classes, you should instantiate only one copy of this class. However you can simultaneously have a frequency receiver and one of the other normal receivers at the same time.

#### 1.1.7.2 Method enableFreqDetect

The class does not begin collecting frequency data until you call the 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 16 bits 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.

The system uses only 16 bits of timing data which is generally sufficient. If you can use more or less RAM memory, you can change the value…

```
#define FREQUENCY_BUFFER_TYPE uint16_t
```

This value can be found in IRLibFreq/IRLibFreq.h and can be changed to either uint8_t or uint32_t however 16 bit default is generally best.

If you are simultaneously receiving a signal using a TSOPxxxx device and one of the three available receiver classes as in the dumpFreq 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 freq 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.

#### 1.1.7.3 Method disableFreqDetect

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() to again. In the normal receiver classes it generally does not hurt anything to leave interrupts running however this class you really should disable it once you have collected sufficient samples. Otherwise the receiver continues collect data overwriting the data collected.

#### 1.1.7.4 Methods computeFreq and dumpResults

Once you have collected the data, 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 8 to 16 bits 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. The computeFreq method stores the results in a float double variable myReceiver.results. It also tells you the number of actual samples used in the computation in the myReceiver.samples. If the samples value is extremely small, you might wish to discard that data and remeasure.

## 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 a number of additional derived classes with one class for each protocol supported. Additionally there is a class will turn your received data into a 32-bit hash code which can be used for creating a unique data value 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: In IRLib1 there was a universal class which incorporated seven of the supported protocols. Additional protocols were supported only through sample sketches which illustrated how to extend the base class to incorporate more protocols. If you wanted to use only one or two protocols or wanted to incorporate any protocols beyond the original seven, it required a great deal of programming on the part of the developer to use this library. The new modular design used by IRLib2 has you include the base decoding class, followed by header files for only the protocols you wish to use, followed by a file which will assemble into a decoder using all and only the protocols you want. That process is described in section 1.2.3 The Combo Decoder Class.

### 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. These signals 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 converts 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. The library has a base decoder class each protocol has a derived class based on base class. There is also a special hash decoder class and another base class which defines common methods used by both RC5 and RC6 protocols.

Below is a version of our example sketch IRLib2/examples/dump/ which you can use as a guide throughout this discussion. Note that this example uses IRLibAll.h to simplify the example but it is not recommended because it includes more code than is necessary and can make your application larger than needed. We recommend you use the combo system which we will describe in section 1.2.3 The Combo Decoder Class.

```
#include "IRLibAll.h"
IRrecv myReceiver(2); //create receiver and pass pin number
IRdecode myDecoder;   //create decoder

void setup() {
  Serial.begin(9600);
  delay(2000); while (!Serial); //delay for Leonardo
  myReceiver.enableIRIn(); // Start the receiver
  Serial.println(F("Ready to receive IR signals"));
}

void loop() {
  //Continue looping until you get a complete signal received
  if (myReceiver.getResults()) {
    myDecoder.decode();           //Decode it
    myDecoder.dumpResults(false);  //the results of a Now print
results.
    myReceiver.enableIRIn();      //Restart receiver
  }
}
```

The IRLibAll.h among other things defines a new derived class called IRdecode which combines all of the supported protocols into a single class. You should then create an instance of that class. That file also makes available all three receiver classes so we will create a receiver as well for this example.

In the setup function it initializes the serial monitor, initializes the receiver, and prints a prompt message to the serial monitor. In the main loop we continually call myReceiver.getResults() and when it returns true we call myDecoder.decode(). That method will return "true" if it discovered one of the protocols that you are using. In this instance we want to see the results even if it did not correctly decode the data so we are ignoring the return value. We then use the myDecoder.dumpResults(true) method to print the results to the serial monitor. We then tell the receiver that we finished and we are ready for another frame of data by calling myReceiver.enableIRIn().

### 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 code is implemented in IRLib2/IRLibDecodeBase.cpp in the prototype in IRLib2/IRLibDecodeBase.h is as follows…

```
// Base class for decoding raw results
class IRdecodeBase {
public:
  IRdecodeBase(void);
  uint8_t protocolNum;      // NEC, SONY, RC5, UNKNOWN etc.
  uint32_t value;           // Decoded value
  uint16_t address;         // More data for protocols > 32 bits
  uint8_t bits;             // Number of bits in decoded value
  bool ignoreHeader; // Relaxed header test allows AGC to settle
  bool decodeGeneric(uint8_t expectedLength, uint16_t headMark,
                     uint16_t headSpace, uint16_t markData,
                     uint16_t spaceOne, uint16_t spaceZero);
  void dumpResults(bool verbose = true);//print data to serial
protected:
  virtual void resetDecoder(void);   // Initializes the decoder
  bufIndex_t offset; // Index into decodeBuffer used various places
};
```

The constructors for the base class and for any of its derived classes take no input. Like the receiver classes you should not create multiple instances of the decoder class because they share common data structures. You instead create a universal decoder class that incorporates multiple decoders into a single class. Note: IRLib1 decoder stored all of their data internally in the object so there was not a restriction on the number of simultaneous decoder objects. This new restriction with IRLib2 should cause no problems.

Although not part of the base class, each protocol class and the combined IRdecode class have a method

```
  bool decode(void);
```

This method will return true if it found one of the included protocols and false otherwise.

The results can be found in 4 data fields of the class. The type of protocol detected is stored in myDecode.protocolNum. The primary data is in the 32-bit variable myDecode.value. The number of significant bits is stored in myDecode.bits. Some protocols have more than 32 bits while others need some sort of additional information such as a repeat flag or other uses. These values are returned in the 16-bit value myDecode.address.

The possible return values for protocolNum can be found in the file IRLibProtocols/IRLibProtocols.h as follows…

```
#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 SAMSUNG36 8
#define GICABLE 9
#define DIRECTV 10
#define RCMM 11
//#define ADDITIONAL_12 12 //add additional protocols here
//#define ADDITIONAL_13 13
#define LAST_PROTOCOL 11 //Be sure to update this
```

In the sample sketch after we called the decode() method we then call dumpResults(true) to look at all of the received data. Here is an example of some typical output from dumpResults…

```
Decoded Sony(2): Value:58BCA Adrs:0 (20 bits)
Raw samples(42): Gap:3200
  Head: m2400  s600
0:m650 s550    1:m1250 s600       2:m650 s550      3:m1250 s550
4:m1300 s550   5:m650 s600        6:m600 s600      7:m600 s600
8:m1250 s550   9:m650 s600        10:m1250 s600    11:m1200 s600
12:m1200 s600  13:m1200 s650      14:m600 s600     15:m600 s600
16:m1250 s600  17:m500 s700       18:m1250 s550    19:m650
Extent=32800
Mark  min:500   max:1300
Space min:550   max:700
```

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 which is the number stored in recvGlobal.decodeLength. It then dumps out all of the values from the decode buffer which is recvGlobal.decodeBuffer[]. 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 IRLib2/examples/analyze example sketch.

The sample output above is the results you get from dumpResults(true) which is the verbose version of the output. If you call dumpResults(false) you get only the top line of the output as follows…

```
Decoded Sony(2): Value:58BCA Adrs:0 (20 bits)
```

There is an additional field that you may want to set before calling your decoder…

```
  bool ignoreHeader;
```

Most receiver devices include a special circuit called Automatic Gain Control or AGC that adaptively boosts the signal to a standard level so it can be properly decoded. Most protocols begin with an exceptionally long “mark” pulse which allows the AGC amplifier to have time to adjust itself. If you are getting a weak signal, this initial heading mark can come in shorter than what was actually transmitted. 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. If the duration of their header mark is not up to protocol, they can still operate if all of the other timing values are correct. 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 myDecoder.ignoreHeader=true; Setting this flag can improve the ability to decode weak signals by allowing the AGC to adjust.

Warning however if you’re using multiple protocols, it can give you inaccurate 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. Fortunately 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.

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.

Earlier we mentioned that the protocol numbers were available in IRLibProtocols/IRLibProtocols.h. That header also defines a function as follows…

```
const __FlashStringHelper *Pnames(uint8_t Type);
```

You pass it a protocol number it 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.

### 1.2.3. The Combo Decoder Class

In IRLib1 there was an omnibus decoder class called IRdecode which combined the 7 supported protocols into a single decoder. However if you only used a small number of protocols it was wasting a lot of code space in your application. As the number of protocols we will support keeps increasing, this wasted space could be significant. We instead created a modular system in which you include the base decoder class followed by header files for only the protocols that you want to use. This can save valuable space in your program. We highly recommend that you use this system rather than including IRLib2.h or IRLibAll.h unless you really need to include absolutely everything.

First we will describe how to use the system for a single protocol. Suppose you're only receiving Sony codes. At the top of your file you would include…

```
#include "IRLibDecodeBase.h"  //Include the base class
```

```
#include "IRLib_P02_Sony.h"   //Include the protocol you want
IRdecodeSony myDecoder;       //Create an instance of the object
```

Note that the name of the class we are creating is IRdecodeSony. You would of course substitute the name of some other protocol if that's what you're using such as IRdecodeNEC, IRdecodeRC5 etc. If your application also would send only Sony codes, you would include the IRLibSendBase.h prior to the protocol as well.

If you are using multiple protocols your code might look like this excerpt from our example sketch IRLib2/examples/comboDump/.

```
#include <IRLibDecodeBase.h> // First include the decode base
#include <IRLib_P01_NEC.h>   // Now include only the protocols you
#include <IRLib_P02_Sony.h>  // wish to actually use. The lowest
#include <IRLib_P07_NECx.h>  // number must be first.
#include <IRLib_P09_GICable.h>
#include <IRLib_P11_RCMM.h>
#include <IRLibCombo.h>     // After all protocols, include this
IRdecode myDecoder;         //Create an instance
```

We begin by including header the IRLibDecodeBase.h followed by the header files for each of the protocols you wish to use. **NOTE: The lowest numbered protocol you're using must be the first included.** Subsequent protocol includes need not be in numerical order but we recommend that you keep them in order anyway. If you would later decide to remove the top protocol then everything else would still be all right. When we say "lowest numbered" we do not mean that you have to use IRLib_P01_NEC.h in every sketch. We mean that among the protocols you are using, the lowest numbered one in your list must be first.

Note: If your application both sends and receives then you would also include the sending base class prior to including all of the protocol classes.

After you have included the base file and all of your protocol files you then include the file IRLibCombo.h. It defines a new derived class called IRdecode which combines all of your selected protocols into a single class. You should then create an instance of that class.

The IRdecode class inherits all of the data fields of the base class and has a single method decode() which returns true if it successfully identified the protocol. The prototype is…

```
bool IRdecode::decode(void);
```

There are at least 11 protocols supported by the library. Each is implemented in his own header file which can be found in the IRLibProtocols folder. The file names are of the form IRLib_Pnn_Name.h where "nn" is the protocol number and "name" is the name of the protocol for example the Sony protocol is in IRLib_P02_Sony.h. There is no associated ".cpp" file for the protocols. All of the code is in the header file. This means that the code will be recompiled every time you compile your sketch. While this will slow down compilation slightly, the flexibility it gives us is well worth it. These files contain both the decoding and sending classes but a series of conditional compile flags insurers that if you do not include the base decode class that the individual protocol decoding classes will not compile. Similarly if you do not include the sending base class, the individual protocol sending classes will not compile. If your application both sends and receives you must include both base classes before including the protocol classes.

Complete details on the individual protocols can be found in section 1.4 Protocol Details. There are more details on how we implemented this combo system in Appendix A and in section 3 Implementing New cut protocols.

## 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. We implement this as an abstract base class and a derived class for each of the supported protocols. Additionally there is a sendRaw class which can be passed an array of raw timing values if you do not know the actual protocol.

> NOTE: In IRLib1 there was a universal class which incorporated seven of the supported protocols. Additional protocols were supported only through sample sketches which illustrated how to extend the base class to incorporate more protocols. If you wanted to use only one or two protocols or wanted to incorporate any protocols beyond the original seven, it required a great deal of programming on the part of the developer to use this library. The new modular design used by IRLib2 has you include the base decoding class, followed by header files for only the protocols you wish to use, followed by a file which will assemble into a decoder using all only the protocols you want. That process is described in section 1.3.3 The Combo Sending Class.

### 1.3.1. Sending Overview

Given the information produced by the decoder classes, you can use that data to send a signal that produces the same results as the one you received. Note however that are sending classes are designed to be used with the decoder data from this library. 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.

Warning: if your application both sends and receives data, the sending of data uses the same hardware timer features as some of the receiving classes. This effectively shuts down your receiver class. Therefore if you are also receiving, after sending data you should always call myReceiver.enableIRIn() to restart your receiver. Although this theoretically should only affect the use of IRrecv we recommend you maintain this practice for IRrecvPCI and IRrecvLoop as well so that you have the opportunity to switch the type of receiver you are using without any unforeseen consequences.

### 1.3.2. IRsendBase Class

The code for the base sending class is implemented in IRLibSendBase.cpp and the prototype is in IRLibSendBase.h both of which are found in the IRLibProtocols folder. They are not in the IRLib2 folder because doing so would have automatically included global functions and structures that are not necessary for sketches that only do sending and not

receiving/decoding. By making this separate we can eliminate overhead and infrastructure not necessary for send-only applications. The prototype is…

```
class IRsendBase {
public:
  IRsendBase();
  void sendGeneric(uint32_t data,  uint8_t numBits,
        uint16_t headMark, uint16_t headSpace, uint16_t markOne,
        uint16_t markZero, uint16_t spaceOne, uint16_t spaceZero,
        uint8_t kHz, bool stopBits, uint32_t maxExtent=0);
protected:
  void enableIROut(uint8_t khz);
  void mark(uint16_t usec);
  void space(uint16_t usec);
  uint32_t extent;
  uint8_t onTime,offTime,iLength;//used by bit-bang output.
};
```

The base class is completely abstract. The constructor takes no parameters. The methods and data fields are all for internal use or for the creation of derived classes. Unlike receiving and decoding classes, there is no restriction on creating multiple sending objects however it is more efficient to use our combo system to create a master sending object.

Although not part of the base class, each send protocol class and the combo class created by IRLibCombo.h contain a single method send(…) which will send the appropriate signal based on the data and other parameters that you specify. That combo file is described in the next section.

### 1.3.3. The Combo Sending Class

In IRLib1 there was an omnibus sending class called IRsend which combined the 7 supported protocols into a single sender. However if you only used a small number of protocols it was wasting a lot of code space in your application. As the number of protocols we will support keeps increasing, this wasted space could be significant. We instead created a modular system in which you include the base sender class followed by header files for only the protocols that you want to use. This can save valuable space in your program. We highly recommend that you use this system rather than including IRLib2.h or IRLibAll.h unless you really need to include absolutely everything.

First we will describe how to use the system for a single protocol. Suppose you're only sending Sony codes. At the top of your file you would include…

```
#include "IRLibSendBase.h"    //Include the base class
#include "IRLib_P02_Sony.h"   //Include the protocol you want
IRsendSony mySender;          //Create an instance of the object
```

Note that the name of the class is IRsendSony. If you are using some of the protocol it would have that protocol name such as IRsendNEC, IRsendRC5 etc. If your application also would decode only Sony codes then you would include the IRLibDecodeBase.h prior to the protocol as well. Then to send a code you would do…

```
mySender.send(0xa8bca, 20);//Sony DVD power A8BCA, 20 bits
```

This passes the proper data and the fact that is a 20 bit version of the Sony protocol.

If you are using multiple protocols your code might look like this…

```
#include <IRLibSendBase.h>   // First include the sendbase
#include <IRLib_P01_NEC.h>   // Now only the protocols you wish
#include <IRLib_P02_Sony.h>  // to actually use. The lowest numbered
#include <IRLib_P07_NECx.h>  // must be first.
#include <IRLib_P09_GICable.h>
#include <IRLib_P11_RCMM.h>
#include <IRLibCombo.h>     // Combine them into "IRsend"

IRsend mySender;    // Create an instance
```

We begin by including header IRLibSendBase.h followed by the header files for each of the protocols you wish to use. **NOTE: The lowest numbered protocol you're using must be the first included.** Subsequent protocol includes need not be in numerical order but we recommend that you keep them in order anyway. If you would later decide to remove the top protocol then everything else would still be all right. When we say "lowest numbered" we do not mean that you have to use IRLib_P01_NEC.h in every sketch. We mean that among the protocols you are using, the lowest numbered one in your list must be first.

Note: If your application both sends and receives then you would also include the decoding base class prior to including all of the protocol classes.

After you have included the base file and all of your protocol files you then include the file IRLibCombo.h. It defines a new derived class called IRsend which combines all of your selected protocols into a single class. You should then create an instance of that class.

The IRsend has only one method that is implemented in IRLibCombo.h with the prototype as follows…

```
void IRsend::send(uint8_t type, uint32_t data, uint16_t data2,
          uint8_t khz=38);
```

You will always invoke this method with at least the first three parameters. The first is always the protocol type. The legal values are found in the file IRLibProtocols/IRLibProtocols.h as follows…

```
#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 SAMSUNG36 8
#define GICABLE 9
#define DIRECTV 10
#define RCMM 11
//#define ADDITIONAL_12 12 //add additional protocols here
```

```
//#define ADDITIONAL_13 13
#define LAST_PROTOCOL 11 //Be sure to update this
```

The second parameter is the 32-bit data value that you will transmit. The third parameter "data2" can have a variety of uses. Most of the time it denotes the number of bits. For example the following line will send a power on signal for a Sony VCR/DVD player that uses a 20 bit Sony code.

```
mySender.send(SONY,0xa8bca, 20);//Sony DVD power A8BCA, 20 bits
```

Note that in this example, unlike our single protocol example, we had to specify which protocol we were using in the first parameter.

Many protocols have a fixed bit length so you can simply pass zero because whatever you pass is ignored. This third parameter can also be used as a repeat flag for some protocols.

The optional fourth parameter is the frequency modulation. If this parameter is omitted or if it is present but equal to zero it will default to 38 kHz which is the most common frequency.

Protocol UNKNOWN 0 is used for the IRsendRaw class. 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.

See the example programs IRLib2/examples/rawRecv and IRLib2/examples/rawSend for example how to use this class. The prototype is…

```
void IRsendRaw::send(uint16_t *buf, uint8_t len, uint8_t 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.

Complete details on how each protocol uses the combined sending object parameters can be found in section 1.4 Protocol Details.

## 1.4 Protocol Details

In this section will provide more details about each of the protocols supported by the library. Of special interest will be information regarding how the IRsend::send() method makes use of its various parameters differently in some protocols than others. The IRdecode::decode() is used consistently among all the protocols however there are some specific issues related to decoding that we will discuss for each protocol below.

The code for each protocol is in a header file which can be found in the IRLibProtocols folder. The file names are of the form IRLib_Pnn_Name.h where "nn" is the protocol number and "name" is the name of the protocol for example the Sony protocol is in IRLib_P02_Sony.h. There is no associated ".cpp" file for the protocols. All of the code is in the header file. This means that the code will be recompiled every time you compile your sketch. While this will slow down compilation slightly, the flexibility it gives us is well worth it. These files contain both the

decoding and sending classes but a series of conditional compile flags insurers that if you do not include the base decoding class that the individual protocol decoding classes will not compile. Similarly if you do not include the sending base class, the individual protocol sending classes will not compile. If your application both sends and receives you must include both base classes before including the protocol classes.

The data sent in an IR signal is logically divided into subfields sometimes known as device numbers, sub device, function, subfunction, OEM codes, checksum fields and so on. IRLib ignores these distinctions and simply lumps all the data into one string of binary bits. Some reference material you will find lists as separate protocols any system that interprets that binary data differently. For example NEC protocol is a 32-bit sequence with a particular set of timing. That exact same timing and 32 bits of data is also used by protocols known as Apple and TiVo however they interpret those 32 bits differently than NEC. Because we do not care about the interpretation of the data, we do not treat these as separate protocols. We consider them all to be NEC protocols because for purposes of decoding and sending they are identical.

Similarly some protocols make the use of a checksum or repeated sequences for error correction. For example Panasonic_Old is a 22 bit protocol in the first 11 bits contain data and the following 11 bits are the same data in binary ones compliment. We do not ensure that this rule is maintained in either encoding or decoding. Also note that in the samples in the following sections when we demonstrate how to send data, those data sequences are just random patterns we made up for illustration purposes. They are not necessarily valid sequences for those protocols and will not follow the internal rules for those protocols.

Additionally some protocols have different variations that use different numbers of bits and some have protocols which are identical except for the modulation frequency. We tried to create single decode and send routines that will handle as many of these variations at once. That way we do not have duplicate routines that waste a lot of valuable code space. So while some references may consider these variations different protocols, we try to lump together as much as we can to make the code as space efficient as possible.

After including the base class files followed by the protocol files then you include IRLibCombo.h which will combine them together to create the IRdecode and IRsend classes using all and only the protocols which you have selected. The following sections are organized in protocol number order.

Among the things we will discuss are the handling of repeat codes. Some protocols give you information to let you know whether or not a particular button was pressed individually or rather it was held down causing it to auto repeat. Some protocols use a special sequence called a "ditto" that does not contain the data that is being repeated. It presumes you have remembered what was recently sent. It sends you a special signal that should be interpreted as "We held down the button and you should repeat what we just sent you". IRLib does not do this for you. It will send you a value equal to REPEAT_CODE which is defined in IRLibProtocols.h as 0xffffffff. Similarly if you send REPEAT_CODE it will generate that ditto sequence for you.

Some other protocols make use of toggle bits. This is a particular bit that will toggle off and on if you repeatedly press and release the same button but will remain constant if you hold down the button and the signal repeats. Unlike a ditto, the data that is being repeated is available in the signal therefore you do not need to remember it from a previous transmission. IRLib does not manage toggle bits for you. It simply decodes what it receives and it's up to you

to decide what to do if anything if that bit switches between successive receptions. Similarly it is up to you to decide whether or not to change the toggle bit when you are sending successive receptions. If you do not care whether a signal was the result of a series of individual keypresses or was the result of a single held keypress then you may want to mask out the toggle bits for these protocols and ignore them.

At the end of each of these protocol sections we will give you a more detailed description of the timing of that protocol in what is known as IRP notation. This is a type of shorthand for describing infrared signals. See Appendix B for an explanation of this notation and a link to some reference material which gives the IRP notation for these protocols and many others that we have not yet supported.

### 1.4.1 NEC, Apple, TiVo and Pioneer Protocols

The NEC protocol is one of the most common and is used by a variety of manufacturers beyond NEC itself. There are 2 varieties known as NEC1 and NEC2. The only difference is the way in which they handle repeat codes. NEC1 makes use of a special ditto sequence which we will decode as the value REPEAT_CODE as described in the introduction to this section. The NEC2 protocol does not use dittos and does not make any distinction between repeated signals or individual keypresses.

Note that the timing of the ditto sequence for NEC1 is nearly identical to the ditto used by the GICable protocol. IRLib generally cannot distinguish between the two. The header timing for GICable consisting of a mark and a space is 8820,1960 and for NEC1 is 9024,2256. If you are using both protocols and you receive an NEC ditto immediately after receiving a GICable then you should presume it is a GICable and vice versa.

This protocol always is 32 bits long. Additional protocols known as Apple and TiVo also use the exact same timing pattern and 32 bits of data as NEC2 however they interpret those 32 bits differently. We however treat all of these as NEC. Additionally Pioneer protocol is identical to NEC2 except that it uses 40 kHz rather than 38 kHz modulation. Unless you are also using a TSMP58000 receiver along with your TSOPxxxx receiver and measuring the frequency as well as the timing pattern, you will not be able to distinguish between NEC2 and Pioneer. Note also that Pioneer sometimes requires a double sequence of data consisting of two different values in order to perform one function.

The code for this protocol is in IRLibProtocols/IRLib_P01_NEC.h. The prototype for the send class for this protocol is…

```
void IRsendNEC::send(uint32_t data, uint8_t kHz=38);
```

The optional second parameter allows you to support Pioneer by changing the frequency to 40 instead of the default 38. Here are some examples if you create an instance of the NEC sending class by itself.

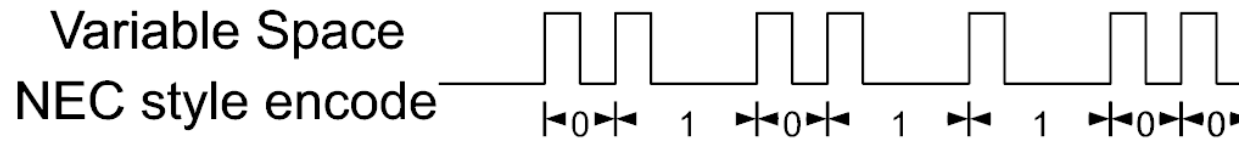
```
IRsendNEC mySender;
…
  mySender.send(0x1234abcd); //send NEC protocol
  mySender.send(REPEAT_CODE); //send NEC ditto sequence
  mySender.send(0xab12c3ff, 40); //send Pioneer protocol
```

When using the IRsend multi-protocol class there are always three parameters. The third parameter called "data2" is the frequency number. You should send it to either zero or 38 for NEC set it to 40 for pioneer. Here are some examples.

```
IRsend mySender;
…
  mySender.send(NEC,0x1234abcd,0); //send NEC protocol
  mySender.send(NEC,REPEAT_CODE,0); //send NEC ditto sequence
  mySender.send(NEC,0xab12c3ff, 40); //send Pioneer protocol
```

This protocol begins with a header consisting of a very long mark and space followed by 32 data bits and a single stop bit which is not encoded by the library. Zeros and ones are distinguished by varying the duration of the space while keeping a constant length for the marks as is typical with most protocols. Here is the IRP notation for all of the protocols supported in this module.

NEC1: {38k,564}<1,-1|1,-3>(16,-8,D:8,S:8,F:8,~F:8,1,^108,(16,-4,1,^108)*)
NEC2: {38k,564}<1,-1|1,-3>(16,-8,D:8,S:8,F:8,~F:8,1,^108)+
Apple: {38.0k,564}<1,-1|1,-3>(16,-8,D:8,S:8,C:1,F:7,I:8,1,^108m,(16,-4,1,^108m)*)
{C=~(#F+#PairID)&1)} C=1 if the number of 1 bits in the fields F and I is even. I is the remote ID. Apple uses the same framing as NEC1, with D=238 in normal use, 224 while pairing. S=135
TiVo: {38.4k,564}<1,-1|1,-3>(16,-8,133:8,48:8,F:8,U:4,~F:4:4,1,-78,(16,-4,1,-173)*)
Pioneer: {40k,564}<1,-1|1,-3>(16,-8,D:8,S:8,F:8,~F:8,1,^108m)+




### 1.4.2 Sony Protocol

The Sony protocol is used almost exclusively by Sony and is unlikely to be used by other manufacturers. The different varieties of the protocol are distinguished only by number of bits. Legal varieties are 8, 12, 15 and 20 bits. The protocol uses 40 kHz modulation however most TSOP receivers tuned to 38 kHz can still receive the signal reasonably well.

The code for this protocol is in IRLibProtocols/IRLib_P01_Sony.h. The prototype for the send class is…

```
void IRsendSony::send(uint32_t data, uint8_t nbits);
```

When using the IRsend multi-protocol class there are always three parameters. The third parameter called "data2" is the number of bits. Here are some examples…

```
IRsendSony mySender;
…
  mySender.send(0x12abc,20); //send 20 bits Sony protocol
  mySender.send(0x12ab, 15); //send 50 bits Sony protocol
```

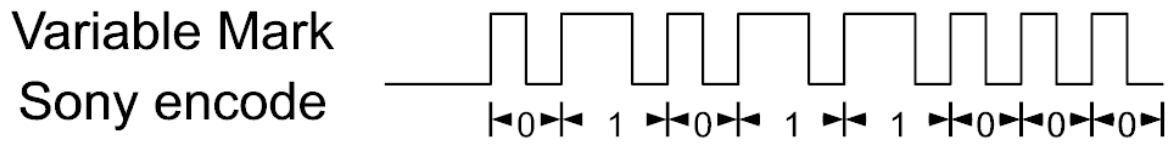
or alternatively…

```
IRsend mySender;
…
  mySender.send(SONY,0x12abc,20); //send 20 bits Sony protocol
  mySender.send(SONY,0x12ab, 15); //send 15 bits Sony protocol
```

The Sony specification requires that every time you press a button, the same signal is sent three times. So when you are receiving Sony signals you can expect each button press to produce three identical signals in a row. Our sending routine automatically repeats the sequence three times so that you only need to call send one time and it will handle the repeats. The specification does not make any distinction between successive keypresses versus a held keypress.

The protocol begins with a header consisting of a long mark and space followed by the data bits. Unlike most protocols which varies the length of the space and keep the length of the mark constant, Sony keeps the space constant and varies the length of the mark. We were able to implement this using our genericSend method however the logic for the decode method did not lend itself to the genericDecode so we had to implement a special decoder. No other known protocols use this system. The IRP notation for these protocols are…

Sony 8:{40k,600}<1,-1|2,-1>(4,-1,F:8,^22200u)
Sony 12: {40k,600}<1,-1|2,-1>(4,-1,F:7,D:5,^45m)+
Sony 15: {40k,600}<1,-1|2,-1>(4,-1,F:7,D:8,^45m)+
Sony 20: {40k,600}<1,-1|2,-1>(4,-1,F:7,D:5,S:8,^45m)+




### 1.4.3 Philips RC5, RC5-7F, and RC5-7F-57

The RC5 protocol was invented by Phillips but is very common and used by a variety of manufacturers. We support three of the four varieties of this protocol. The most common variety is designated simply as RC5 is a 13 bit protocol. There is also a 14 bit version known as RC5-7F. Both of these use a modulation frequency of 36 kHz however they can typically be received by standard 38 kHz TSOP receivers. There is another variation of RC5-7F known as RC5-7F-57 which is distinguished from RC5-7F only by the modulation frequency which is 57 kHz. It may be less likely that a 36 kHz receiver can accurately detect 57 kHz but it may be possible.

There is another variety known as RC5x which we do not currently support. It is so significantly different the other varieties that it will probably require a completely separate module.

The code for this protocol is in IRLibProtocols/IRLib_P03_RC5.h. The prototype for the send class for this protocol is…

```
void IRsendRC5::send(uint32_t data, uint8_t nBits=13,
          uint8_t kHz=36);
```

The second and third parameters are optional and will default to 13 bits and 36 kHz which is the generic RC5 variety. Here are some examples if you create an instance of the RC5 sending class by itself.

```
IRsendRC5 mySender;
…
  mySender.send(0x12ab); //send RC5 protocol default 13 bits 36 kHz
  mySender.send(0x3abc,14); //send RC5-7F 14 bits default 36 kHz
  mySender.send(0x3abc,14,57); //send RC5-7F-57 specify both
                               //bits and the kHz
```

When using the IRsend multi-protocol class there are always at least three parameters. The third parameter called "data2" is the number of bits and must always be specified as 13 or 14. The optional fourth parameter is the frequency which should be 36 or 57. If frequency is omitted, it defaults to 36. Here are some examples.

```
IRsend mySender;
…
  mySender.send(RC5,0x12ab,13); //send RC5 protocol. Must be 13 bits
  mySender.send(RC5,0x3abc,14); //send RC5-7F. Must specify 14 bits
  mySender.send(RC5,0x3abc,14,57); //send RC5-7F-57.  Specify
                                   //both bits & kHz
```

This protocol does not have a traditional long header instead it begins with a single "1" bit followed by either 13 or 14 data bits. We do not encode this initial start bit. Note that the IRremote library by Ken Shirriff upon which IRLib is based treats this as a 12 bit protocol that always begins with 2 consecutive start bits which are not encoded. However that second bit is not necessarily always "1". IRLib has always treated this as a 13 bit protocol with a single start bit.

All three varieties make use of a toggle bit to denote the difference between the repeated pressing of a single key or a held key which is repeating. The toggle bit changes if the button was pressed and released but it remains the same if the button was held. The toggle bit for RC5 is the 12th most significant bit or 0x800. The toggle bit for both varieties of RC5-7F is the 13th most significant bit or 0x1000.

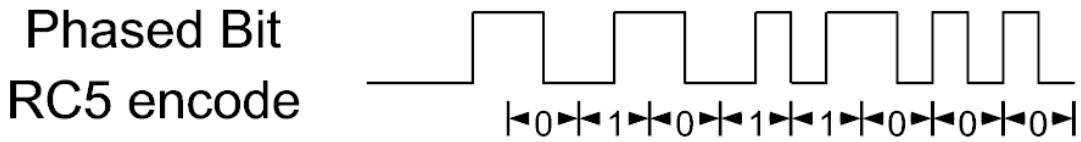
All varieties of RC5 use a special phase encoding of bits. A space/mark indicates a "1" and a mark/space indicates a "0". The RC6 group of protocols described in the next section also use phase encoding however the logic is reversed and that a space/mark indicates "0" and vice versa. Because these two groups of protocols share this common unusual feature they are derived from the IRdecodeRC and IRsendRC abstract classes which in turn are derived from the base classes.

The IRP notation for these protocols is…

RC5: {36k,msb,889}<1,-1|-1,1>(1,~F:1:6,T:1,D:5,F:6,^114m)+
RC5-7F: {36k,msb,889}<1,-1|-1,1>(1, ~D:1:5,T:1,D:5,F:7,^114m)+
RC5-7F-57: {57k,msb,889}<1,-1|-1,1>(1, ~D:1:5,T:1,D:5,F:7,^114m)+

The currently unsupported RC5x IRP notation is…

RC5x: {36k,msb,889}<1,-1|-1,1>(1,~S:1:6,T:1,D:5,-4,S:6,F:6,^114m)+




### 1.4.4 Phillips RC6, RC6-6-20, Replay and MCE Protocols

The RC6 protocol was invented by Phillips but is used by wide variety of manufacturers. We support four known varieties of this protocol. The most common variety is designated simply as RC6 is a 16 bit protocol. Technically it could be designated RC6-0-16. Some Sky and Sky+ remotes use a 20 bit version known as RC6-6-20. A protocol sometimes known as "Replay" is actually RC6-6-24 and finally "MCE" is a 32 bit protocol which is really RC6-6-32.

These all use a modulation frequency of 36 kHz however they can typically be received by standard 38 kHz TSOP receivers. The sequence begins with a header consisting of a long mark/space followed by a single start bit which we do not encode. This is followed by a three bit OEM code that is either "0" or "6". This is followed by a special sequence known as a trailer bit which is of double length from normal bits. In all but the 32-bit version the trailer bit also serves as a toggle bit. This trailer/toggle bit is then followed by the actual data bits. The 32-bit version always uses a zero as the trailer bit but it is not a toggle bit. The 32-bit version uses the highest order of the 32 bits as its toggle bit. The toggle bit changes if the button was pressed and released but it remains the same if the button was held.

According to our research the 16-bit version always uses OEM code 0 and the other versions always use OEM code 6. We could have chosen to hardcode these values but we wanted to make the code as flexible as possible allowing for OEM codes other than 0 or 6. We also wanted to encode the toggle bits in all cases. Because we encode three bits of OEM data plus the toggle bit, the bit lengths used by IRLib are actually 4 more than a bit lengths in the specification in the first three versions.

Specifically the protocols which are 16, 20, and 24 bits in length are handled by IRLib as being 20, 24, and 26 bits long respectively. Because it is difficult to encode and handle data greater than 32 bits, we chose not to encode the three bit OEM code which we believe will always be 6. Furthermore the trailer bit is always 0 so we are not encoding it. IRLib encodes only the 32 data bits. Also the highest order of the 32 bits is the toggle bit. If we ever encounter a 32-bit version with an OEM other than 6 it will require a special modified encoder and decoder. Here is a summary of information about the supported varieties

| Name | Description | Spec Bits | IRLib Bits | Toggle |
|---|---|---|---|---|
| RC6-0-16 | Original Phillips | 16 | 20 | 0x00010000 |
| RC6-6-20 | Used by some Sky and Sky+ | 20 | 24 | 0x00100000 |
| RC6-6-24 | Replay protocol | 24 | 28 | 0x01000000 |
| RC6-6-32 | MCE protocol | 32 | 32 | 0x80000000 |

The code for this protocol is in IRLibProtocols/IRLib_P04_RC6.h. The prototype for the send class for this protocol is…

```
void IRsendRC6::send(uint32_t data, uint8_t nBits=16);
```

The second parameter defaults to 16 bits. It will also default to 16 bits if you pass zero. Here are some examples if you create an instance of the RC6 sending class by itself.

```
IRsendRC6 mySender;
…
  mySender.send(0x12ab);       //send RC6 protocol
  mySender.send(0x12ab,20);    //send RC6 same as previous
  mySender.send(0xc12abc,24);  //send RC6-6-20
  mySender.send(0xc123abc,28); //send RC6-6-20 Replay protocol
  mySender.send(0x1234abcd,32); //send RC6-6-32 MCE protocol
```

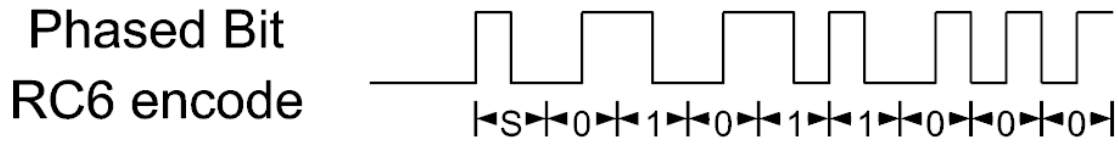
When using the IRsend multi-protocol class there are always three parameters. The third parameter called "data2" is the number of bits and must always be specified. However if you pass zero as the third parameter it will default back to original RC6-0-16. For example…

```
IRsend mySender;
…
  mySender.send(RC6,0x12ab,0);     //send RC6 protocol
  mySender.send(RC6,0x12ab,20);    //send RC6 same as previous
  mySender.send(RC6,0xc12abc,24);  //send RC6-6-20
  mySender.send(RC6,0xc123abc,28); //send RC6-6-20 Replay protocol
  mySender.send(RC6,0x1234abcd,32); //send RC6-6-32 MCE protocol
```

All varieties of RC6 use a special phase encoding of bits. A space/mark indicates a "0" and a mark/space indicates a "1". The RC5 group of protocols described in the previous section also use phase encoding however the logic is reversed and that a space/mark indicates "1" and vice versa. Because these two groups of protocols share this common unusual feature they are derived from the IRdecodeRC and IRsendRC abstract classes which in turn are derived from the base classes.

The IRP notation for these protocols is…

RC6: {36k,444,msb}<-1,1|1,-1>(6,-2,1:1,0:3,<-2,2|2,-2>(T:1),D:8,F:8,^107m)+
RC6-6-20: {36k,444,msb}<-1,1|1,-1>(6,-2,1:1,6:3,<-2,2|2,-2>(T:1),D:8,S:4,F:8,-???)+
Replay: {36k,444,msb}<-1,1|1,-1>(6,-2,1:1,6:3,<-2,2|2,-2>(T:1),D:8,S:8,F:8,-100m)+
MCE: {36k,444,msb}<-1,1|1,-1>(6,-2,1:1,6:3,-2,2,OEM1:8,OEM2:8,T:1,D:7,F:8,^107m)+




### 1.4.5 Panasonic_Old, Scientific Atlantic, Cisco, Time Warner, Bright House

The Panasonic_Old protocol is used by some cable boxes and DVR's manufactured by Scientific Atlantic and Cisco. They are often used by Time Warner and Bright House Cable systems. There is only a single 22 bit variety so you do not need to specify any bit length or frequency parameters. The protocol uses 57 kHz modulation which can be received by some 38 kHz receivers but may not be received by all of them. In our experience some will work and some will not.

The code for this protocol is in IRLibProtocols/IRLib_P05_Panasonic_Old.h. The prototype for the send class is…

```
void IRsendPanasonic_Old::send(uint32_t data);
```

When using the IRsend multi-protocol class there are always three parameters but because we have no need of it, whenever you pass as the third parameter is ignored. Here are some examples…

```
IRsendPanasonic_Old mySender;
…
  mySender.send(0x321abc); //send Panasonic_Old protocol
```

or alternatively

```
IRsend mySender;
…
  mySender.send(PANASONIC_OLD,0x321abc,0); //send Panasonic_Old
```

The protocol has no special repeat codes or toggle bits. It uses traditional fixed length marks and variable length spaces so it is implemented with the genericDecode and genericSend methods of the base class. For an illustration of how bits are encoded see the diagram at the end of the NEC section which also uses variable length spaces. The IRP notation for the protocol is…

Panasonic_Old: {57.6k,833}<1,-1|1,-3>(4,-4,D:5,F:6,~D:5,~F:6,1,-44m)+

### 1.4.6 JVC Protocol

JVC protocol is used by equipment made by that manufacturer. It is a 16-bit protocol that uses traditional variable length spaces and fixed length marks. It begins with a long mark/space header however that header is omitted on repeat frames. So an initial frame will have a header but subsequent frames of the same signal omit the header portion. Our only experience with this protocol was with an old JVC VCR/DVD player. We could only get the device to operate if we would send an initial frame with the header followed by a repeat frame without the header. The device would not work if only the initial frame was sent. Although this specification doesn't specifically state this, we decided to make our send routine automatically send a repeat frame after an initial frame. We also give you the opportunity to send additional repeat frames.

The code for this protocol is in IRLibProtocols/IRLib_P06_JVC.h and the prototype is…

```
void IRsendJVC::send(uint32_t data, bool first=true);
```

The second parameter should be true if this is an initial frame with a header and should be false if it is a repeat frame. When sending an initial frame it always sends a repeat frame. When sending a repeat frame, it is only sent once. Here are some examples if you create an instance of the JVC sending class by itself.

```
IRsendJVC mySender;
…
  mySender.send(0x12ab);       //send initial JVC plus auto repeat
  mySender.send(0x12ab,true);  //send same as previous
```

```
  mySender.send(0x12ab,false); //send one repeat frame
```

When using the IRsend multi-protocol class there are always three parameters. The third parameter called "data2" is the header bits and must always be specified. For example…

```
IRsend mySender;
…
  mySender.send(JVC,0x12ab,true);//send initial plus repeat
  mySender.send(JVC,0x12ab,false); //send one repeat frame
```

Because a repeat signal does include the full 16 bits of data we did not implement this using the REPEAT_CODE. Because there is not technically a toggle bit we needed a way to indicate whether or not you received an initial frame with a header or a repeat frame without one. For that reason IRdecodeJVC::decode() uses the "address" field as a flag. As always the data is in myDecode.value but additionally myDecode.address will be true if the header was present meaning this is an initial frame and it will be false if there was no header designating a repeat frame.

For an illustration of how bits are encoded see the diagram at the end of the NEC section which also uses variable length spaces. The IRP notation for this protocol is…

JVC: {38k,525}<1,-1|1,-3>(16,-8,(D:8,F:8,1,-45)+)

### 1.4.7 NECx Protocol

The NECx protocol is similar to the original NEC protocol. It is used by NEC and a variety of other manufacturers especially Samsung televisions. It differs in timing from NEC only by the length of the header. If you use the myDecoder.ignoreHeader=true; option then IRLib cannot distinguish between NEC and NECx.

Similar to NEC there are 2 varieties known as NECx1 and NECx2. The only difference is the way in which they handle repeat codes. NECx1 makes use of a special ditto sequence which we will decode as the value REPEAT_CODE as described in the introduction to this section. The NECx2 protocol does not use dittos and does not make any distinction between repeated signals or individual keypresses.

This protocol always is 32 bits long. It always uses 38 kHz modulation. Therefore there is no need for additional parameters when sending. The code for this protocol is in IRLibProtocols/IRLib_P07_NECx.h. The prototype for the send class for this protocol is…

```
void IRsendNEC::send(uint32_t data);
```

When using the IRsend multi-protocol class there are always three parameters but because we have no need of it, whatever you pass as the third parameter is ignored. Here are some examples…

```
IRsendNECx mySender;
…
  mySender.send(0x1234abcd);  //send NECx protocol
  mySender.send(REPEAT_CODE); //send NECx ditto sequence
```

or alternatively

```
IRsend mySender;
…
  mySender.send(NECX,0x1234abcd,0); //send NEC protocol
  mySender.send(NECX,REPEAT_CODE,0);//send NEC ditto sequence
```

This protocol begins with a header consisting of a very long mark and space followed by 32 data bits and a single stop bit which is not encoded by the library. Zeros and ones are distinguished by varying the duration of the space while keeping a constant length for the marks as is typical with most protocols. For an illustration of how bits are encoded see the diagram at the end of the NEC section which also uses variable length spaces. The IRP notation for these protocols are:

NECx1: {38k,564}<1,-1|1,-3>(8,-8,D:8,S:8,F:8,~F:8,1,^108,(8,-8,D:1,1,^108m)*)
NECx2: {38k,564}<1,-1|1,-3>(8,-8,D:8,S:8,F:8,~F:8,1,^108)+

### 1.4.8 Samsung36 Protocol

The Samsung36 protocol is a somewhat strange 36 bit protocol used by some Samsung devices especially Blu-ray players. Note not all Samsung devices use this protocol for example most Samsung TVs use NECx. Although Arduino based programs can use 64 bit variables, the overhead to do so is excessive. Therefore we have implemented this as a 16-bit address and a 20 bit data value. The data is actually transmitted in three segments of 16, 12, and 18 bits each but we have combined the 12 and 18 bits segments into the single 20 bit value.

When this protocol is decoded the first 16 bits are stored in myDecoder.address and the remaining 20 bits are in myDecoder.value. It has been our experience that the address is always the same for a given device so it may be sufficient for you to just use the value when determining which button has been pressed. You should test this with your own equipment to see if it is true.

There are no alternate variations of this protocol known. It always uses 36 bits and 38 kHz modulation.

The code for this protocol is in IRLibProtocols/IRLib_P08_Samsung36.h and the prototype is…

```
void IRsendSamsung36::send(uint32_t data, uint32_t address);
```

Although the second parameter will only ever be passed a 16 bit value, the internal logic of the code is simpler to implement if both the address and data have been extended to 32 bits. You must specify both the data and the address even if the address is invariant in your application. In that case you simply pass the same value each time. Here are some examples if you create an instance of the Samsung36 sending class by itself.

```
IRsendSamsung36 mySender;
…
  mySender.send(0x28d7,0x0400); //send Blu-ray play
  mySender.send(0xc837,0x0400); //send Blu-ray stop
```

or alternatively for the multiprotocol class…

```
IRsend mySender;
```

```
…
  mySender.send(SAMSUNG36,0x28d7,0x0400); //send Blu-ray play
  mySender.send(SAMSUNG36,0xc837,0x0400); //send Blu-ray stop
```

Although this protocol uses the standard fixed length marks and variable length spaces, it does not lend itself to using the decodeGeneric or sendGeneric base class methods. Therefore it implements its own internal functions called getBits and putBits to decode or encode portions of the stream. For an illustration of how bits are encoded see the diagram at the end of the NEC section which also uses variable length spaces. The IRP notation for this protocol is…

Samsung36: {38k,500}<1,-1|1,-3>(9,-9,D:8,S:8,1,-9,E:4,F:8,-68u,~F:8,1,-118)+

### 1.4.9 GICable, General Instruments, Motorola

The G.I. Cable protocol is used by some Motorola cable boxes and DVR's manufactured by General Instruments. They are used by a variety of cable systems. We have seen them on Time Warner/Bright House systems as an alternative to Panasonic_Old protocol. It is a 16 bit protocol very similar to NEC1 in that it uses a special ditto sequence as a repeat code. Like NEC1 we will decode as the value REPEAT_CODE as described in the introduction to this section.

Note that the timing of the ditto sequence for GICable is nearly identical to the ditto used by the NEC1 protocol. IRLib generally cannot distinguish between the two. The header timing for GICable consisting of a monarch and a space is 8820,1960 and for NEC1 is 9024,2256. If you are using both protocols and you receive an NEC ditto immediately after receiving a GICable then you should presume it is a GICable and vice versa.

This protocol always is 16 bits long and uses 38 kHz modulation. There are no known alternative variations to the protocol.

The code for this protocol is in IRLibProtocols/IRLib_P09_GICable.h and the prototype for the send class for this protocol is…

```
void IRsendGICable::send(uint32_t data);
```

Here are some examples if you create an instance of the GICable sending class by itself.

```
IRsendGICable mySender;
…
  mySender.send(0x12ab);      //send GICable protocol
  mySender.send(REPEAT_CODE); //send GICable ditto sequence
```

When using the IRsend multi-protocol class there are always three parameters. The third parameter is unnecessary and is ignored. Here are some examples.

```
IRsend mySender;
…
  mySender.send(GICABLE,0x12ab,0);     //send GICable protocol
  mySender.send(GICABLE,REPEAT_CODE,0);//send GICable ditto sequence
```

This protocol begins with a header consisting of a very long mark and space followed by 16 data bits and a single stop bit which is not encoded by the library. Zeros and ones are distinguished by varying the duration of the space while keeping a constant length for the marks as is typical with most protocols. For an illustration of how bits are encoded see the diagram at the end of the NEC section which also uses variable length spaces. Here is the IRP notation…

GICable: {38.7k,490}<1,-4.5|1,-9>(18,-9,F:8,D:4,C:4,1,-84,(18,-4.5,1,-178)*)
where {C = -(D + F:4 + F:4:4)}

### 1.4.10 DirecTV Protocol

The DirecTV protocol is used by that manufacture's set top boxes. It comes in six different varieties. It uses three different frequencies of 38, 40, or 57 kHz. It also uses two different varieties of lead out times of either 9000 µs or 30,000 µs. A "lead out time" is the amount of blank space at the end of a signal before a subsequent signal can begin. By using combinations of three frequencies and two lead out times that gives six varieties. The default is 38 kHz and 30,000 µs.

The sequence begins with a long mark/space header followed by 16 bits of data. It uses an unusual encoding system which encodes data 2 bits at a time. We will describe this in more detail later. The initial header mark is 6000 µs but on repeated signals is only 3000 µs. In the decoding routines we set myDecoder.address=true; if it is an initial frame and it is false for repeat frames. The default is true.

> NOTE: DirecTV protocol was not officially supported in IRLib1. There was however an example sketch illustrating the protocol. The behavior of this repeat flag has been reversed from that old sample sketch. Previously the flag=true meant it was a repeat frame. Now true designates that it is a first frame. This makes it compatible with previous implementations of other protocols such as JVC.

Most 38 kHz TSOPxxxx devices will be able to receive 40 kHz signals however they may or may not be able to receive 57 kHz signals.

Our multiprotocol send class has been created to use no more than four parameters. We are already using all of them for the protocol number, date of value, first frame flag, and frequency. Expanding it to an additional parameter would make it more difficult for other protocols. Therefore we need an additional variable to tell whether to use the long or short lead out. You can set mySender.longLeadOut= false; to use a shorter lead out time of 9000 µs rather than the default 30,000 µs. Note that the lead out time is not really a critical issue especially since the default is the longer time. But we give you the opportunity to change this feature if you choose to do so.

The code for this protocol is in IRLibProtocols/IRLib_P10_DirecTV.h and the prototype for the send class for t's for his protocol is…

```
void IRsendDirecTV::send(uint32_t data, bool first= true,
                                uint8_t khz=38);
```

Here are some examples if you create an instance of the DirecTV sending class by itself.

```
IRsendDirecTV mySender;
…
```

```
  mySender.send(0x12ab);      //send DirecTV 1st frame, 38 kHz
  mySender.send(0x12ab,false);//send repeat frame, 38 kHz
  mySender.send(0x12ab,true,57);//must specify frame type when using
                                //alternate frequency
  mySender.send(0x12ab,57);//WRONG!! Interprets 57 as true 1st frame
                           //and default 38 kHz frequency
```

Note that you must specify the initial frame flag as true or false whenever using an alternative modulation frequency. Otherwise the modulation frequency is interpreted as the frame flag and the frequency will default to its usual 38.

When using the IRsend multi-protocol class there are always at least three and possibly 4 parameters. The third parameter is required and is the first frame flag. The optional fourth parameter is the frequency. Here are some examples.

```
IRsend mySender;
…
  mySender.send(DIRECTV,0x12ab,true);// DirecTV 1st frame, 38 kHz
  mySender.send(DIRECTV,0x12ab,false);//send repeat frame, 38 kHz
  mySender.send(DIRECTV,0x12ab,true,57);//1st frame 57 kHz
```

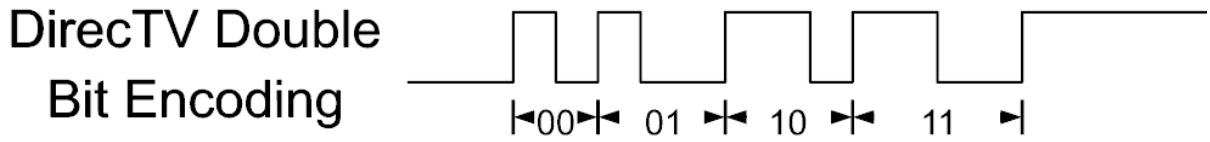
As mentioned earlier, this protocol uses an unusual encoding system that varies both the length of a mark and the length of a space in order to encode 2 bits for each mark/space pair. Marks and spaces are either 600 µs or 1200 µs long. The encoding goes as follows…

| Mark | Space | Value |
|---|---|---|
| 300 | 300 | 00 |
| 300 | 600 | 01 |
| 600 | 300 | 10 |
| 600 | 600 | 11 |

This means that after the initial header mark and space there are only eight mark/space pairs necessary to encode 16 bits of data. Dedicated decoder and sending routines are necessary because this is not the usual variable length space used by sendGeneric and decodeGeneric methods. We have found no other protocol that uses this unique system. If another such firm always found we will implement sendDoubleBit and decodeDoubleBit functions that they can share.

The IRP notation for this protocol is…

DirecTV: {38k,600,msb}<1,-1|1,-2|2,-1|2,-2>(5,(5,-2,D:4,F:8,C:4,1,-50)+)
where {C=7*(F:2:6)+5*(F:2:4)+3*(F:2:2)+(F:2)}




### 1.4.11 Phillips RCMM, Nokia or U-Verse Protocol

The Phillips rcmm protocol is sometimes known as the Nokia protocol but is most famous because it is used by set-top boxes for AT&T U-Verse cable systems. It comes in 12,

24, and 32 bit varieties all using 36 kHz modulation frequency. Most 38 kHz receiver devices can still receive that frequency. It uses an unusual double bit encoding system that is different even from the DirecTV double bit system described earlier. We will discuss this at the end of the section. The 32-bit version uses a toggle bit of 0x00008000 and as usual it is up to the end-user to implement it outside the library routines.

The code for this protocol is in IRLibProtocols/IRLib_P11_RCMM.h. The prototype for the send class is…

```
void IRsendRCMM::send(uint32_t data, uint8_t nBits=12);
```

When using the IRsend multi-protocol class there are always three parameters. The third parameter called "data2" is the number of bits. Here are some examples…

```
IRsendRCMM mySender;
…
  mySender.send(0x12a,12); //send 12 bits rcmm protocol
  mySender.send(0x123abc,24); //send 24 bits rcmm protocol
  mySender.send(0x1234abcd,32); //send 32 bits rcmm protocol
```

or alternatively…

```
IRsend mySender;
…
  mySender.send(RCMM,0x12a,12); //send 12 bits rcmm protocol
  mySender.send(RCMM,0x123abc,24); //send 24 bits rcmm protocol
  mySender.send(RCMM,0x1234abcd,32); //send 32 bits rcmm protocol
```

This protocol uses a fixed length mark and a variable length space however rather than using a long space to denote a logical "1" and a short space to denote logical "0", it uses 4 different lengths of spaces to denote bit patterns of 00, 01, 10, and 11 as follows…

| Mark | Space | Value |
|---|---|---|
| 167 | 277 | 00 |
| 167 | 444 | 01 |
| 167 | 611 | 10 |
| 157 | 778 | 11 |

This presents a problem because normally IRLib decodes with a tolerance of 25% however if you take 25% of the 778 value you get 194.5 which is excessive. So instead of comparing values based on a percentage of the total, we implemented an alternative system that is a tolerance plus or minus some absolute value. That value can be found in IRLibProtocols/IRLib_P11_RCMM.h.

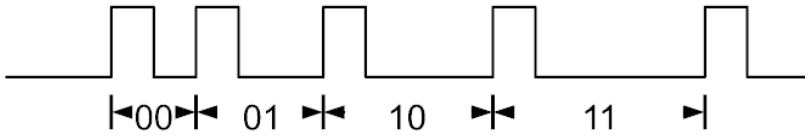
```
#define RCMM_TOLERANCE 80
```

This means that if comparing for example to 611 anything from 691 down to 531 would be considered. Even this value gives some overlap but it does generally work. The code looks for the shortest values first. We highly recommend that you use the IRrecvPCI or IRrecvLoop receivers when using this protocol because the other IRrecv receiver can introduce errors plus or minus 50 µs. It may also be necessary to change the myDecoder.markExcess value to properly decode this protocol.

The IRP notation for this protocol is…

Nokia 12 bit: {36k,msb}<167,-277|167,-444|167,-611|167,-778>(412,-277,D:4,F:8,167,-???)+
Nokia 24-bit: {36k,msb}<167,-277|167,-444|167,-611|167,-778>(412,-277,D:8,S:8,F:8,167,-???)+
Nokia 32 bit: {36k,msb}<167,-277|167,-444|167,-611|167,-778>(412,-277,D:8,S:8,T:1X:7,F:8,167,^100m)+

RCMM Double Bit Encoding




## 1.5 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 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 AVR chips 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.

### 1.5.1. 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 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 port registers. We have decided not to implement such a capability.

The IRrecvLoop receiver class uses no interrupts or timers so it allows you to get away from such hardware dependency. Using 16 MHz 8-bit AVR processors 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. Note that if you have an 8 MHz processor even the PCI method used by IRfrequency is not fast enough to measure modulation frequencies above 40 kHz. If you measure something in the range of 46 kHz or so it is probably actually a 57 or 58 kHz signal which are the standard high frequencies.

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 and 66% off 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 IRLibHardware.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 & | 11 | 9* | 5 | 6 | | 46 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | ATmega2560 | | | | | | |
| 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.

### 1.5.2 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 1.5.3 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 to 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 you would choose to do so 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 instead 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 IRLibHardware.h. The section which allows you to change timers begins at approximately line 35. 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 IRLibHardware.h at approximately line 110 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 an 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 IRLibHardware.h in a section at approximately line 90. 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.

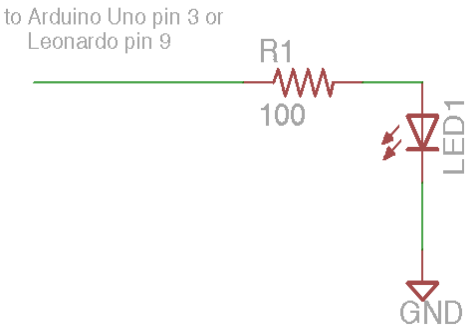
Note that if you use bit-bang sending and IRrecvLoop for receiving you will not need any hardware timers or hardware interrupts. This may be especially useful for unsupported platforms.

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 IRLibHardware.h as long as they connect their output hardware to a supported pin and they do not have conflicts with other libraries.

### 1.5.3. IR LED Driver Components and Schematics

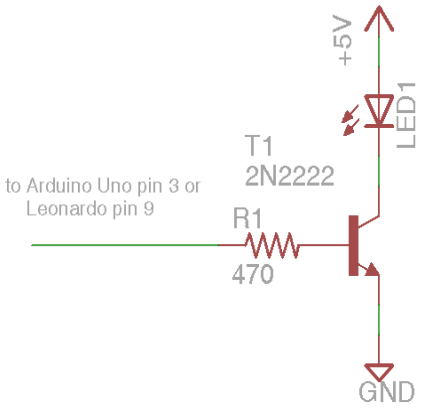
The simplest output circuit is simply connect the IR LED directly to the output pin of the Arduino and then connect it to +5 volts with a current limiting resistor of 100 ohms like this.




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.

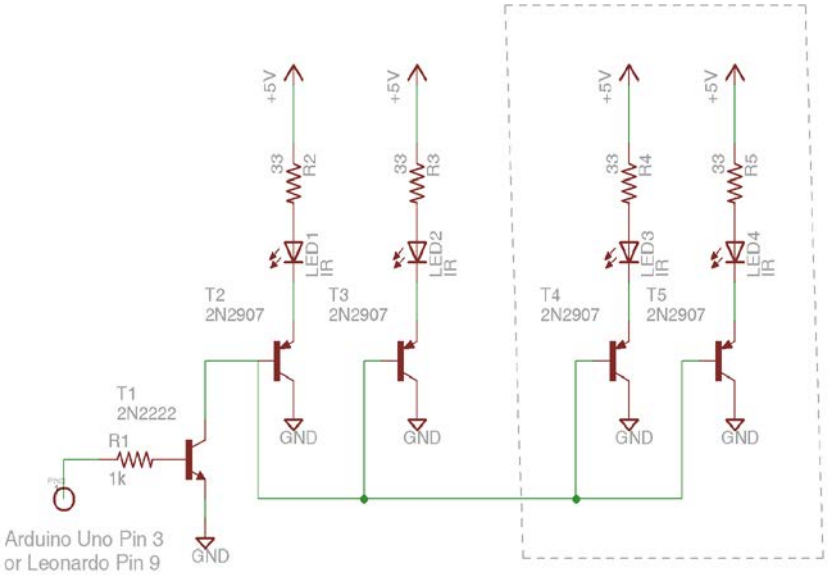
Note that all of our examples presume +5 volt supplies however some Arduino systems run at 3.3 volts. These schematics should also work at 3.3 volts however you might want to slightly lower some of the resistor values.

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 and 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 IRLib_NoOutput(); 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.

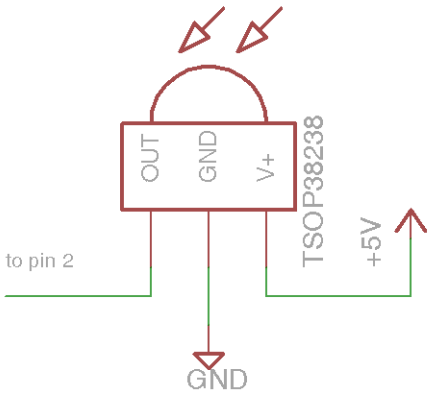


The circuit starts with an NPN transistor connected to the Arduino output pin via a 1K resistor. That NPN then drives up to four PNP transistors that drive the LEDs. Note that we have added a 33 ohm resistor to each LED to limit the current. This resistor was added because I designed the circuit to be 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 of each in a two LED application or two of each in a four LED application.

### 1.5.4 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. If using the IRrecvPCI or IRfrequency classes you are limited to particular pins. Even though there are no restrictions when using IRrecv and IRrecvLoop, we have started using pin 2 for all receivers because it makes it easy to switch back and forth between IRrecvPCI and the other two receivers.

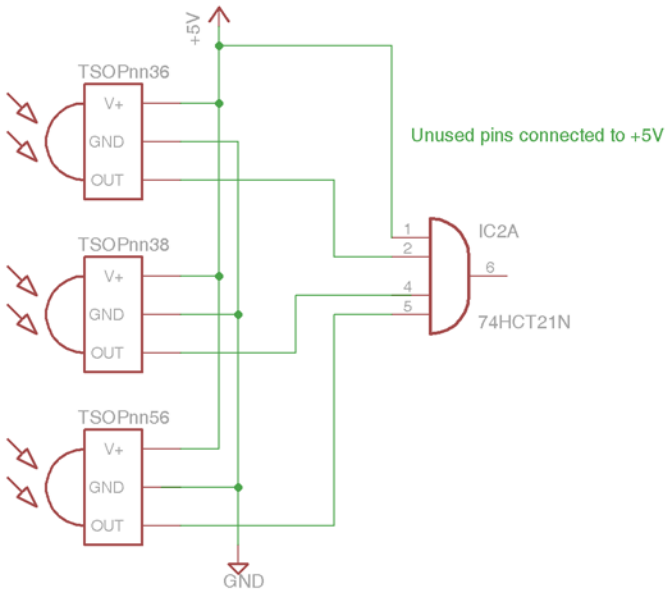
While the schematic is trivial, the challenge is finding the proper device for receiving. Although we deal with IR signals as though they are square waves with pulses varying from a couple of hundred microseconds up to a few thousand microseconds, in fact those pulses are actually modulated at a frequency from somewhere between 30 kHz and 58 kHz. Different protocols use different frequencies. The TSOP series of devices from Vishay are generally used. They demodulate 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 different frequencies, the bandpass filters in these 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 https://www.radioshack.com/products/38khz-infrared-receiver-module?variant=5717577093

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 2760640. 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.

There may be instances in which you need to use multiple receivers connected to a single pin. 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. Alternatively 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 could 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

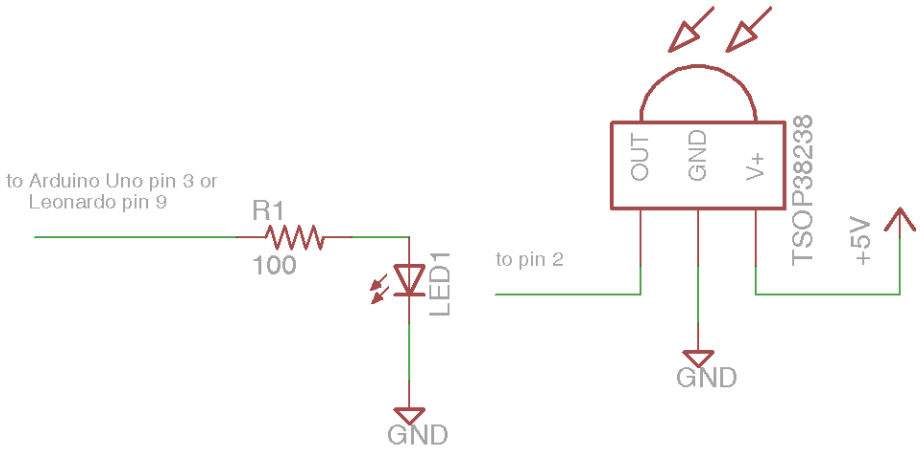
While part one of this document is intended to be a complete documentation of all the features of this library with lots of detail, this section is more of a step-by-step beginner's tutorial that will try not to overwhelm you with all the details at once. There is a general setup section followed by a section for each of the sample sketches which you will find in the folder IRLib2/examples. It presumes you have an Arduino Uno, or Leonardo or Mega development board or something equivalent. It presumes you have some sort of infrared remote such as a TV, cable box, DVD player might use. We will show you how to capture the signals from that remote, decode them, and re-create them using your own IR LED and a driver circuit.

## 2.1 Setting up Your Hardware

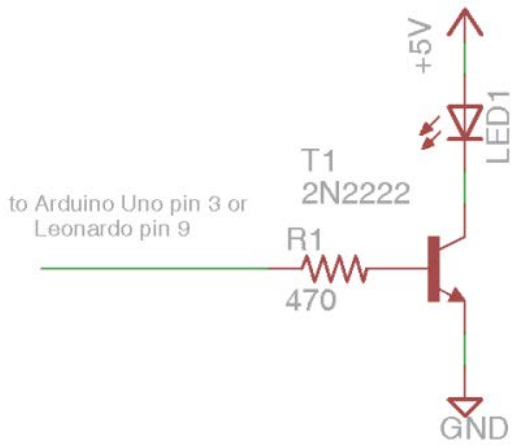
To receive IR signals you will need a receiver device such as a TSOP4438 or something similar. In section 1.5.4 IR Receiver Components and Schematics you can find complete details on these type of devices and where to obtain them. For purposes of getting started quickly we suggest you try RadioShack if there still is one near you or perhaps Adafruit. These devices have three pins. The first pin on the left is the signal pin, the center is ground, and the right pin is power which is usually +5v although it will work on +3.3v systems if that's what you're using.

For most applications the signal pin can be connected to any digital input pin on your Arduino however in all of our examples we will use pin 2. If you're using pin change interrupts you have to use particular pins and 2 is one of them. By using pin 2 all of the time we can switch back and forth between different receiving methods.

To transmit IR signals you need an IR LED. The schematic below shows the simplest possible circuit that you can use to transmit signals. All you need is the LED and a 100 ohm current limiting resistor.




Be sure to get the polarity correct on the LED. The shorter of the two leads connects to ground and the longer one connects to the resistor.




Because Arduino pins cannot produce much current we recommend you use an NPN transistor as a driver circuit if at all possible. On the left is a schematic of what that looks like.

Whether you are using a transistor driver or connecting directly with a limiting resistor you should use pin 3 for Arduino Uno or Mega and use pin 9 on Leonardo platforms. Note the polarity of the LED again. The long leads goes to +5v and the short lead connects to the transistor.

Note that this particular circuit will overdrive the LED beyond its rating. However because we are using a modulated signal that is only on part of the time, it can briefly handle that amount of current. If your application is going to send continuous pulses you might consider an alternative schematic which you can find in the hardware section of this document.

## 2.2 Receiving IR signals

In the section we will give you several examples of how to receive and decode IR signals from your TV or cable or DVD remote. For these examples you will only need the receiver portion of the circuit.

### 2.2.1 The Dump Example

Here is a slightly edited version of the IRLib2/examples/dump.ino sample program.

```
#include "IRLibAll.h"

IRrecv myReceiver(2); //create receiver and pass pin number
IRdecode myDecoder;   //create decoder

void setup() {
  Serial.begin(9600);
  delay(2000); while (!Serial); //delay for Leonardo
  myReceiver.enableIRIn(); // Start the receiver
  Serial.println(F("Ready to receive IR signals"));
}

void loop() {
  //Continue looping until you get a complete signal received
  if (myReceiver.getResults()) {
    myDecoder.decode();           //Decode it
    myDecoder.dumpResults(true);  //Now print results.
                                  // Use false for less detail
    myReceiver.enableIRIn();      //Restart receiver
  }
}
```

This starts with including the header file IRLibAll.h which will include everything in the IRLib library. In general we don't recommend that you do this because it will put lots of things in your program that you don't necessarily need. They will use up valuable memory and may conflict with other libraries you might be using. But for a simple sample program such as this one where we don't care about the most efficient use, it will do just fine.

We need to create a receiver object to receive the IR signals and put the data in a buffer. We also need a decoder to interpret those signals and turn them into a single binary number that we can use to identify the signal and if necessary later re-create it. We will use the IRrecv receiver class which is one of three different varieties we can use. We will talk about the others later. The parameter 2 tells that that we are using pin 2 for our input. We will use a decoder class called IRdecode which contains all of the protocols which this library supports.

The setup function initializes the serial port and prints a ready message on the serial monitor. After you upload the sketch you will need to open the serial monitor on your Arduino IDE and you should see the ready message. The statement myReceiver.enableIRIn() initializes the receiver object. Your device will now begin looking for signals to be received and recording the length of each pulse and space between pulses measured in microseconds.

In our loop function we continuously call our receiver's getResults() method. When that method returns true, it means that the receiver has received a complete signal which we call a frame of IR data. Once the complete frame has been received and the data has been stored in a buffer, the receiver stops receiving until we tell it to begin again with another call to enableIRIn().

Once we have the data then we need to decode it. The decoder looks at the timing of the pulses and determines if it recognizes the particular protocol. The call to myDecoder.decode() will return true if it recognizes the protocol and false if it does not. However in this example we are ignoring the return value from that method. We want to see what was received whether it was a protocol that we recognize or not.

Next we call myDecoder.dumpResults(true); to print out our results on the serial monitor. The true parameter gives us a verbose dump of the results. If you pass false it will give you just one line of information about the received code. Here is an example of what I saw when I pushed the play button on the remote for my Sony DVD player.

```
Ready to receive IR signals
Decoded Sony(2): Value:58BCA Adrs:0 (20 bits)
Raw samples(42): Gap:65300
  Head: m2400  s600
0:m600 s650    1:m1150 s650       2:m550 s650      3:m1200 s650
4:m1150 s700   5:m500 s700        6:m550 s650      7:m550 s650
8:m1150 s700   9:m550 s650        10:m1150 s650    11:m1200 s650
12:m1150 s650  13:m1200 s650      14:m550 s650     15:m600 s600

16:m1200 s650  17:m550 s650       18:m1200 s650    19:m550
Extent=32750
Mark  min:500   max:1200
Space min:600   max:700
```

This tells us that it decoded Sony protocol which is protocol 2. A binary representation of the code in hexadecimal is 58BCA. Some protocols also make use of another piece of data which we call "Address" but in this case Sony does not use that field. It also tells us that this is a 20 bit version of the Sony protocol. Some protocols have varieties of different numbers of bits while others always use a fixed number of bits.

When a signal is on we call it a "mark". The empty interval between signals we call a "space". After telling us that it received 42 samples of marks and spaces it says there was a gap of 65300 µs. This is the amount of time from the previous signal to the start of this signal. Most of the time we simply ignore that gap although there can be rare instances where it's useful information. Sometimes you want to know the amount of space between frames of data.

Most protocols begin with a header which consists of an abnormally long mark and space. In this case we had a mark of 2400 and a space of 600. Next comes the actual data bits each consisting of a mark and a space. The timing values for these marks and spaces are given. We note that bit 0 has 600, 650 followed by bit 1 which has a mark of 1150 and a space of 650. If you look at each bit some have marks and spaces that are nearly equal to one another while others have marks which are approximately twice the normal value but spaces that are still in the range of 600 or so. We interpret each of those long marks as being a binary "1" and each of them with nearly equal length marks and spaces as a binary "0". If you do the math and convert them to hexadecimal you will get the value 58BCA which is what we reported.

This means that that hexadecimal value is a compact or encoded way of representing all of that timing data. We can then use that hex value to later re-create the same signal because we know how the Sony protocol works. While Sony encodes its bits using a long marks and equal length spaces, most protocols are the opposite and use constant marks and variable length spaces. And again other protocols use even more complicated systems of representing zeros and ones that are not quite as easy to interpret by just looking at the numbers.

After all of the data bits have been printed, it also prints some other information that can be useful in analyzing unknown protocols. We won't go into that right now.

One of the features of the Sony protocol is that it always transmits the en's tire sequence three times in a row. When I created this particular example it printed out that entire dump of information only two times instead of three. The reason I did not receive all three copies is that we missed receiving one. While we were busy printing out the first sequence, we missed the second one but managed to capture the third one. Later we will show you how to use an extra buffer and a feature called auto resume which allows you to continue to receive another buffer full of data while you are working on the first one.

If you edit the parameter to dumpResults to be false you only get the top line of information like this.

```
Ready to receive IR signals
Decoded Sony(2): Value:58BCA Adrs:0 (20 bits)
Decoded Sony(2): Value:58BCA Adrs:0 (20 bits)
Decoded Sony(2): Value:58BCA Adrs:0 (20 bits)
```

Because we did not print out as much information, the receiver was able to capture all three signals. Most protocols do not repeatedly send the same signal multiple times unless you

hold down the button for a long period of time or pump it repeatedly. Sony is unique in that respect that it always sends everything three times.

### 2.2.2 Using Different Receivers

In our previous example we use the receiver class IRrecv which we said was one of three different possible receiver classes. You may have noticed that all of the timing values we received were multiples of 50. That is because this particular receiver uses a timer driven interrupt every 50 µs. When 50 µs have elapsed, the interrupt is triggered and it measures whether the input pin is showing a mark or a space. So it is only looking at the data every 50 µs. That is why all of the timing data is measured in multiples of that chunk of time.

There are two other available receiver classes that we will briefly discuss here. Complete details on these classes are available in the reference section. Start with the dump example that we used in the previous section but edit the line where we created the receiver to read…

```
IRrecvPCI myReceiver(2);
```

This uses the PCI version of the receiver code. PCI stands for Pin Change Interrupt. It is a hardware interrupt that is triggered every time a pin switches from low to high or from high to low. It will give us more accurate results than the 50 µs version of the receiver. Here is some sample output using the same Sony code. Depending on what kind of Arduino you are using, there are only certain pins available for using PCI interrupts. It happens that pin 2 is one of them on most platforms so that's why we use it as our usual example. The IRrecv receiver class does not have this restriction and it can use any digital input pin.

```
Ready to receive IR signals
Decoded Sony(2): Value:58BCA Adrs:0 (20 bits)
Raw samples(42): Gap:27874
  Head: m2406  s634
0:m590 s634    1:m1190 s626       2:m622 s602      3:m1190 s634
4:m1190 s658   5:m566 s634        6:m582 s634      7:m590 s634
8:m1190 s634   9:m590 s634        10:m1214 s634    11:m1166 s658
12:m1158 s658  13:m1166 s642      14:m582 s650     15:m598 s634

16:m1134 s714  17:m558 s642       18:m1158 s658    19:m566
Extent=32854
Mark  min:558   max:1214
Space min:602   max:714
```

We still recognize this as a Sony protocol with a data value of 58BCA and 20 bits of data however you can see that the timing values are no longer even multiples of 50. We are getting much more accurate timing value than with the IRrecv receiver. For everyday use the original receiver is accurate enough to decode the data but if you're analyzing unknown protocols the PCI version is better.

One of the limitations of the PCI receiver is that sometimes it does not pick up a second or third frame of data beyond the first one unless there is a long delay between frames. While the IRrecv was able to accurately decode two out of the three of the Sony signals, the PCI version will only get the first one. If I change the dump parameter to false making the print out

go much faster, PCI would capture all three signals. Later we will show you how to use the auto resume feature which makes the PCI receiver much more likely to capture all signals.

Finally there is the IRrecvLoop receiver class. You can try editing it as follows

```
IRrecvLoop myReceiver(2);
```

It can use any available digital input pin. It does not use any timer or pin change interrupts or any special platform specific features at all. However it has severe limitations. Using the other two receivers we would call enableIRIn() and it would begin receiving immediately. Our program could go off and do other things while it was waiting for the frame to complete and we only needed to intermittently call getResults() to see if it was ready. If getResults() returned false, we could go about our business doing other things. Using the loop version of the receiver, it does not begin receiving immediately. It does not poll the input pin until your first call to getResults() and when you do call it, it sits in a very tight loop monitoring the input pin as quickly as possible. It does not release control of your program until the complete sequence has been received. You cannot do anything else while looking for a sequence to come in. And of course because it is not interrupt driven you cannot use the multitasking auto resume feature.

This type of receiver is most useful because it can use any input pin and you will not have any conflicts with other libraries that make use of hardware timers or interrupts.

### 2.2.3 Efficient Decoding with the ComboDump Example.

We mentioned that the use of the IRLib2/IRLibAll.h header file included absolutely everything that our library supports but that can use up valuable memory especially if you are only using one or two of the supported protocols. We suggested you not use that include file for everyday use but only use it when you really want to be able to decode everything you could.

In this section we will describe how to more efficiently use only one receiver class and create a special decoder class containing only the protocols you want to use.

Here is a version of the example file in IRLib2/examples/comboDump.h sample program.

```
#include <IRLibDecodeBase.h> // First include the decode base
#include <IRLib_P01_NEC.h>   // Include only the protocols you wish
#include <IRLib_P02_Sony.h>  // to actually use. The lowest numbered
#include <IRLib_P07_NECx.h>  // must be first. #include
<IRLib_P09_GICable.h>
#include <IRLib_P11_RCMM.h>
#include <IRLibCombo.h>     // After all protocols, include this
// All of the above automatically creates a universal decoder
// class called "IRdecode" containing only the protocols you want.
// Now declare an instance of that decoder.
IRdecode myDecoder;

// Include a receiver either this or IRLibRecvPCI or IRLibRecvLoop
#include <IRLibRecv.h>
IRrecv myReceiver(2);  //pin number for the receiver

void setup() {
```

```
  Serial.begin(9600);
  delay(2000); while (!Serial); //delay for Leonardo
  myReceiver.enableIRIn(); // Start the receiver
  Serial.println(F("Ready to receive IR signals"));
}
void loop() {
  //Continue looping until you get a complete signal received
  if (myReceiver.getResults()) {
    myDecoder.decode();           //Decode it
    myDecoder.dumpResults(true);  //Now print results.
    myReceiver.enableIRIn();      //Restart receiver
  }
}
```

This example starts with multiple include files. The first file is IRLibDecodeBase.h which contains code which is common to all of the protocol decoder functions and tells the system that you want to do some decoding. You then list include files for only the protocols that you want to use. See the reference section for a list of supported protocols and their file names. Or look in the IRLibProtocols folder for these files. In this example we have chosen NEC, Sony, NECx and GICable. There is a restriction that the lowest numbered protocol among those that you are using must be the first one included. While it is technically okay to put subsequent protocols in any order, we recommend that you always put them all in numerical order. For example if you put GICable before Sony it would be okay but if you later deleted NEC then they would be out of sequence. So it's best just to keep them all in order.

After including all of the protocols you want, you must include the file IRLibCombo.h which will then combine all of the individual decoders for each protocol into a master decoder class called IRdecode. We create an instance of that class called myDecoder as always.

Because we are not using IRLibAll.h we also need to include a file to tell the program which receiver class we wish to use. It's best not to include them all because as we have explained some can conflict with other programs and libraries. In our case we include IRLibRecv.h and create an instance of the IRrecv receiver class. If we want to use the other receivers we would substitute as follows

```
#include <IRLibRecvPCI.h>
IRrecvPCI myReceiver(2);
```

or attentively

```
#include <IRLibRecvLoop.h>
IRrecvLoop myReceiver(2);
```

The remainder of the example is the same as our original dump.ino example in the previous sections except that it will only decode the protocols we have selected. If I press the play button on my Sony remote I will get the same results as I got before. But if I press the play button on my Bright House Networks cable/DVR set-top box I get the following

```
Decoded Unknown(0): Value:0 Adrs:0 (0 bits)
Raw samples(48): Gap:45126
  Head: m3400  s3300
```

```
0:m850 s2500   1:m900 s2450       2:m900 s800      3:m850 s2500
4:m900 s2450   5:m900 s2450       6:m900 s2500     7:m900 s750
8:m900 s800    9:m900 s2450       10:m950 s2450    11:m850 s800
12:m850 s850   13:m900 s2450      14:m950 s750     15:m900 s800

16:m900 s800   17:m900 s750       18:m950 s2450    19:m850 s2500
20:m850 s800   21:m950 s750       22:m950
Extent=63150
Mark  min:850   max:950
Space min:750   max:2500
```

As you can see it does not give us a protocol name or number, value, address, nor number of bits. The protocol is not recognized. I happen to know however that my cable box uses protocol 5 named Panasonic_Old. If I add a line after the Sony include that reads

```
#include <IRLib_P05_Panasonic_Old.h>
```

And recompile and capture the signal again then it will recognize the code. Here is the short version of the decoded dump.

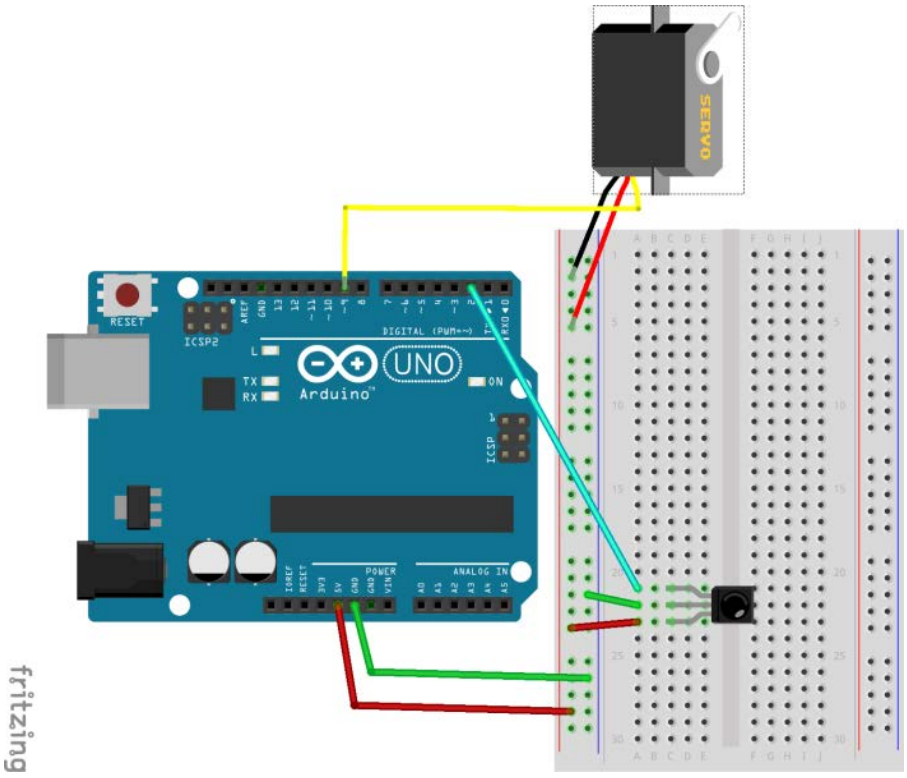
```
Ready to receive IR signals
Decoded Panasonic Old(5): Value:37990C Adrs:0 (22 bits)
```

In the beginning of this section we said that using the combo system would save lots of program memory. Because Arduino based platforms have an extremely limited amount of memory, having unused code in your program can have a big impact. For example the dump.ino example compiled on an Arduino Uno reports that it takes 11,608 bytes (35%) of program storage space. However our combo example using just five protocols uses 7,972 bytes (24%) of program storage space. If we strip the example down even further to use just one protocol it would use 4,686 bytes (14%) of program storage space.

While none of these examples come close to filling up all of your memory, receiving and decoding of IR signals is usually only a small part of what your project will do. For example I have a particular application that transmits five different protocols similar to our previous example. However it also has a 128x64 pixel OLED display, and communicates with my iPhone via Bluetooth low energy. It is very close to filling the entire program memory of a 32u4 processor such as the one used by Arduino Leonardo. I would like to add an additional protocol to that project but there just isn't enough memory. At some point I will have to switch to a more powerful platform in order to implement everything I want to do with the project. However I would not have gotten this far if I had not used this more efficient combo program in place of throwing everything in together.

### 2.2.4 Controlling a Servo using an IR Remote

So far we have learned how to receive and decode IR signals but now we want to try to do something useful with that information. We are going to use an IR remote to control the position of a servo. You will need an IR receiver and a servo motor the type of which are used in radio controlled models or robots. You can obtain them at any hobby store or our favorite parts supplier Adafruit.com. Here is a wiring diagram for the project.




Our IR receiver is wired into pin 2 as usual. A power and ground line should be connected to the servo. The control line from the servo should go to pin 9 on the Arduino. This sample program uses the Arduino library "Servo" which makes use of hardware timer 1. That is the same hardware timer used by IRLib on the Arduino Leonardo. If you are using a Leonardo or other ATmega32u4 based platforms you will have to change the default timer used by IRLib. See the reference manual section 1.5.1 and 1.5.2 for details.

Below is our sample program which can be found at IRLib2/examples/servo/.

```
/* servo.ino Example sketch for IRLib2
 *  Illustrate how to now to control a servo using IR signals.
 */
#include <IRLibDecodeBase.h> // First include the decode base
#include <IRLib_P02_Sony.h>  // Include only one protocol
IRdecodeSony myDecoder; // Now declare an instance of that decoder.

#include <Servo.h>

#include <IRLibRecv.h>
IRrecv myReceiver(2); //pin number for the receiver

Servo myServo;  // create servo object to control a servo
int16_t pos;         // variable to store the servo position
int16_t speed;       // Number of degrees to move each time
                     // a left/right button is pressed

void setup() {
```

```
  myServo.attach(9);       // attaches the servo on pin 9 to the
                           // servo object
  pos = 90;                // start at midpoint 90 degrees
  speed = 3;               // servo moves 3 degrees each time
                           // left/right is pushed
  myServo.write(pos);      // Set initial position
  myReceiver.enableIRIn(); // Start the receiver
}

// You will have to set these values depending on the protocol
// and remote codes that you are using. These are from my
// Sony DVD/VCR
#define MYPROTOCOL SONY
#define RIGHT_ARROW   0x86bca //Move several clockwise
#define LEFT_ARROW    0x46bca //Move servo counterclockwise
#define SELECT_BUTTON 0xd0bca //Center the servo
#define UP_ARROW      0x42bca //Increased degrees servo moves
#define DOWN_ARROW    0xc2bca //Decrease degrees servo moves
#define BUTTON_0 0x90bca  //Pushing buttons 0-9 moves to fixed
#define BUTTON_1 0x00bca  // positions each 20 degrees greater
#define BUTTON_2 0x80bca
#define BUTTON_3 0x40bca
#define BUTTON_4 0xc0bca
#define BUTTON_5 0x20bca
#define BUTTON_6 0xa0bca
#define BUTTON_7 0x60bca
#define BUTTON_8 0xe0bca
#define BUTTON_9 0x10bca

void loop() {
  if (myReceiver.getResults()) {
    if(myDecoder.decode()) {
      switch(myDecoder.value) {
        case LEFT_ARROW:    pos=min(180,pos+speed); break;
        case RIGHT_ARROW:   pos=max(0,pos-speed); break;
        case SELECT_BUTTON: pos=90; break;
        case UP_ARROW:      speed=min(10, speed+1); break;
        case DOWN_ARROW:    speed=max(1, speed-1); break;
        case BUTTON_0:      pos=0*20; break;
        case BUTTON_1:      pos=1*20; break;
        case BUTTON_2:      pos=2*20; break;
        case BUTTON_3:      pos=3*20; break;
        case BUTTON_4:      pos=4*20; break;
        case BUTTON_5:      pos=5*20; break;
        case BUTTON_6:      pos=6*20; break;
        case BUTTON_7:      pos=7*20; break;
        case BUTTON_8:      pos=8*20; break;
        case BUTTON_9:      pos=9*20; break;
      }
      myServo.write(pos); // tell servo to go to position in
```

```
                            // variable 'pos'
    }
    myReceiver.enableIRIn();
  }
}
```

Note that because we are only using the Sony protocol we do not need to include IRLibCombo.h to create a combo decoder. We can use the Sony decoder directly. Note however that that class name is IRdecodeSony and not just IRdecode.

Also note that after getResults() returns true, we check to see if myDecoder.decode() returns true. If IRLib receives a frame of IR data from something other than a Sony protocol, then getResults() would still return true. It would get the data but it's just not the data we wanted. In previous examples we were curious to see what was received whether it was the right data or not. But in this case we only want to do something if we received valid data.

If your application uses more than one protocol you would also want to put in a condition that said something like…

```
if(myReceiver.getResults()) {
  if(myDecoder.decode()) {
    if (myDecoder.protocolNum== SONY) {
         //now we know we've got good Sony data and nothing else
    }
  }
}
```

Unless you happen to have the exact same Sony DVD player that I have, this sketch will not work for you. You should first load the dump.ino sketch and find out the proper protocol and codes for whatever type of remote you are using. For the time being, you will have to have a remote that uses one of our supported protocols. Using the dump sketch, find out what codes are transmitted for the 10 numbered keys 0 through 9 as well as or four directional arrow keys and the select key. Edit those into the sample sketch as well as editing which protocol you will be using.

Now when you press the left or right arrow keys, the servo will move 3° clockwise or counterclockwise. If you push the up or down arrow keys, it will increase the increment that is used when you press the left or right arrow keys. If you push any of the numbered buttons it will move the servo to one of 10 fixed positions.

The sample program is just to spark some ideas. Using a similar program you can control all sorts of devices. If you have a robot arm with multiple servos you can have different buttons on the remote change the position of different servos. You could use IR signals to turn on LEDs or addressable pixels such as NeoPixels off and on or change colors. If you attach a relay and appropriate driver circuitry you can use it to turn on or off a lamp or other device.

### 2.2.5 Receiving and Decoding Unknown Protocols Using Hash Codes

So far we have assumed that you are using one of our 11 protocols that are supported by this library. While we hope to expand the number of supported protocols, creating a new decoder and sending routine for a new protocol is a very difficult process. We devote a major portion of this document in section 3 on how to implement new protocols. It requires intimate

knowledge of how IR signals work and a fair amount of programming skill that is beyond this tutorial. However all hope is not lost. There is another decoder called a "hash decoder".

The hash decoder analyzes the timing signals and creates a 32 bit binary value that represents the pattern which was received. It is highly likely that this pattern will be unique for a given IR sequence. However it is not 100% guaranteed. When using a known protocol, we can take the decoded binary value and later re-create the original signal from that bit pattern. Unfortunately using the hash decoder there is no way to reverse engineer the sequence and re-create the original signal based on the hash value. If you are using IRLib only to receive and decode signals such as our servo sample program in the previous section, then the hash decoder should be sufficient to come up with some unique value for each of the 15 functions which that program uses.

Here is the listing for the program IRLib2/examples/hashDecode.ino that we will use.

```
/* hashDecode.ino Example sketch for IRLib2
 * Illustrate the use of the hash decoder.
 */
//First will create a decoder that handles only NEC, Sony and
// the hash decoder. If it is not NEC or Sony that it will
// return a 32 bit hash.
#include <IRLibDecodeBase.h>
#include <IRLib_P01_NEC.h>
#include <IRLib_P02_Sony.h>
#include <IRLib_HashRaw.h>  //Must be last protocol
#include <IRLibCombo.h>     // After all protocols, include this
IRdecode myDecoder;

#include <IRLibRecv.h>
IRrecv myReceiver(2);    //create a receiver on pin number 2

void setup() {
  Serial.begin(9600);
  delay(2000); while (!Serial); //delay for Leonardo
  myReceiver.enableIRIn(); // Start the receiver
  Serial.println(F("Ready to receive IR signals"));
}

void loop() {
  if(myReceiver.getResults()) {
    myDecoder.decode();
    if(myDecoder.protocolNum==UNKNOWN) {
      Serial.print(F("Unknown protocol. Hash value is: 0x"));
      Serial.println(myDecoder.value,HEX);
    } else {
      myDecoder.dumpResults(false);
    };
    myReceiver.enableIRIn();
  }
}
```

For this example I will use the remote for my cable box/DVR. Although that particular box uses the Panasonic_Old protocol which is supported, to illustrate the hash decoder I am not including that protocol in the stack of protocols used in this example. The program will report it as unknown and provide a 32 bit hash code. I can plug those values into the servo sample and use this unknown protocol. Here are the values I obtained for the left arrow, right arrow, select, up arrow, and down arrow buttons.

```
Unknown protocol. Hash value is: 0x1BDD06FA
Unknown protocol. Hash value is: 0xD98004F2
Unknown protocol. Hash value is: 0xBB3030B6
Unknown protocol. Hash value is: 0x6E7FD7A
Unknown protocol. Hash value is: 0x4318951A
```

I can then plug those values into the servo sketch along with the values for button 0 through 9 and change that sketch to use the hash decoder instead of the Sony decoder. Note that these values are not really the proper bit sequence for that protocol. If I use the proper decoder Panasonic_Old then I would have obtained these values.

```
Decoded Panasonic Old(5): Value:364137 Adrs:0 (22 bits)
Decoded Panasonic Old(5): Value:37810F Adrs:0 (22 bits)
Decoded Panasonic Old(5): Value:366133 Adrs:0 (22 bits)
Decoded Panasonic Old(5): Value:36812F Adrs:0 (22 bits)
Decoded Panasonic Old(5): Value:37A10B Adrs:0 (22 bits)
```

As you can see our hash values bear no resemblance whatsoever to the actual data. But we don't care as long as it gives us some unique value for each button press. You should be able to adapt this method to your application as long as all you need to do is receive the signal and not retransmit it.

### 2.2.6 Using Auto Resume

The major new feature of IRLib2 other than the reorganization of the code is the auto resume feature. Normally when the receiver object detects that a complete frame of data has been received, it goes into an idle mode and waits for you to call getResults() and to process the data. It does not resume looking for data until you call enableIRIn() to notify the receiver you are ready for another bunch of data. However there are times when you might want the receiver to continue to record data from a second frame while you are still processing the first one. As we saw in an earlier example with the Sony protocol, the signal is repeated three times rapidly we often are not able to finish processing one frame before another one comes along.

The auto resume feature allows the receiver to copy the data to an external buffer called the decode buffer. This frees up the receive buffer so that the receiver can immediately begin recording another frame of data. Normally the decode buffer and receive buffer are the same buffer.

This feature is especially useful when using the IRrecvPCI receiver which has difficulty determining the end of frame unless there is an especially long gap between frames. This particular receiver becomes much more likely to detect a quick second frame when using auto resume.

The disadvantage to this feature is that it is costly in terms of RAM memory. You need to declare an extra buffer using 100 words of 16-bit integer values. Example code for how to use this feature can be found in IRLib2/examples/autoResume which is shown below.

```
#include "IRLibAll.h"

IRrecvPCI myReceiver(2); //create receiver and pass pin number
IRdecode myDecoder;   //create decoder

//Create a buffer that we will use for decoding one stream while
//the receiver is using the default buffer "recvGlobal.recvBuffer"
uint16_t myBuffer[RECV_BUF_LENGTH];

void setup() {
  Serial.begin(9600);
  delay(2000); while (!Serial); //delay for Leonardo
  //Enable auto resume and pass it the address of your extra buffer
  myReceiver.enableAutoResume(myBuffer);
  myReceiver.enableIRIn(); // Start the receiver
  Serial.println(F("Ready to receive IR signals"));
}

void loop() {
  //Continue looping until you get a complete signal received
  if (myReceiver.getResults()) {
    myDecoder.decode();           //Decode it
    myDecoder.dumpResults(true);  //Now print results.
    myReceiver.enableIRIn();      //Restart receiver
  }
}
```

Note that we have defined an extra buffer called myBuffer using the default receive buffer length. When in the setup function before you enableIRIn() you call myReceiver.enableAutoResume(myBuffer). As you can see you pass it the address of the external buffer.

Everything else is automatic. When the receiver records the first frame of data it will automatically copy it to the external buffer. Your decoder will use that external buffer for its processing and the receive buffer that is normally used by IRLib2 is free to start receiving another frame of data. Although it continues to record the second frame immediately, you still shouldn't call myReceiver.enableIRIn() until you have finished processing the data. When you call that function you are in effect telling the receiver that the decoder has completed processing everything and it is safe to copy a new frame to decode buffer. Using IRLib1 things worked differently. There was no auto resume but there was the opportunity to declare an extra buffer. Using IRLib1 you would reenable IR input using a "resume" function that we no longer have. Then you would do this resume immediately after getResults() returns true. We considered renaming enableIRIn() as "decoderWantsMoreData()" but decided that it's given name was more accurate.

### 2.2.7 Advanced Receiving Examples

There are three additional examples in the IRLib2/examples folder. One of them is called "analyze" and it is sort of a souped-up version of our usual "dump" routine however it is used to analyze unknown protocols. Its use is thoroughly explained in section 3 on implementing your own protocol code.

The other two examples are "freq" and "dumpFreq". The first one is to measure the modulation frequency of an IR signal and the second one combines that function with our normal dump routine so that you can see the pattern of marks and spaces as well as measure the frequency.

IR signals are not normal square waves that you would measure in hundreds of microseconds as it might seem. Each of the marks which we treat as a continuous "on" signal is actually a series of tight pulses at a frequency of somewhere between 36 kHz and 57 kHz. The typical TSOPxxxx receiver device uses a bandpass filter to only listen to signals in those frequency ranges and turns it into the clean square wave that we normally use where a mark is completely "on" at a space is completely "off". There is another class of devices called TSMP that does not include the bandpass filter. We recommend the TSMP58000 for measuring frequency.

Frequency is measured using the IRfrequency receiver class. It uses pin change interrupts just like the IRrecvPCI receiver class. That means it is limited to a particular number of special pins that can use PCI interrupts. In our examples we always use pin 2 for traditional IRrecvPCI reception and we use pin 3 to measure frequency. See the reference section on IRrecvPCI and IRfrequency for more details. Let's look at the IRLib2/examples/freq code in a few sections.

We create an instance of the IRfrequency class by passing it the pin number to which we have connected our TSMP58000 device. In the setup function we initialize serial output as always. There is a test to see the speed of the Arduino device we are using. Most Arduino platforms use 16 MHz processors however some Arduino compatible devices that run at 3.3 V operate at only 8 MHz. It has been our experience that measurement of modulation frequencies above 40 kHz are inaccurate using the slower processors.

We also have included code that verifies that the pin we are using is a valid PCI pin number. That section of code looks like this…

```
  int8_t intrNum=digitalPinToInterrupt(FREQUENCY_PIN);
  if (intrNum==NOT_AN_INTERRUPT) {
    Serial.println(F("Invalid frequency pin number."));
    while (1) {};//infinite loop because of fatal error
  }
```

In the dumpFreq sample program which is similar to this one, we performed that test not only for the frequency receiver but also for the standard IRrecvPCI receiver. If you are using either of these PCI based receivers and are not getting any results, it could be that you have chosen an invalid pin number. Use these examples to verify that the pin you are using will work.

The last item in our setup function is to call myFreq.enableFreqDetect(). This is similar to the enableIRIn() function that we are familiar with in our traditional receivers. It initializes the

frequency detection receiver and immediately begins recording pulses. It will continue to record data until its buffer overflows. In our main loop we test myFreq.haveData() and when it returns true we know that we have a pretty good sample of pulses. Unlike the usual receivers, the frequency detection receiver continues to record data overwriting the buffer constantly until you call myFreq.disableFreqDetect(). To see the results of our data recall myFreq.dumpResults() which gives us a report of what it found.

Because we do not have the necessary speed to measure the size of an individual wavelength, it measures up to 256 wavelengths and averages them. There is a parameter to the dumpResults() functions which will give you verbose or brief reports.

There is also another method called myFreq.computeResults() that does the computation. It is automatically called by dumpResults() however if you want to do the computation but not dump the results and use it some other way you can call computeResults() directly. The computeResults() will set 2 values. First is myFreq.results which is a float value containing the frequency in kilohertz. Also myFreq.samples is an integer number of samples that were used to compute that frequency.

The sample program dumpFreq is a combination of the frequency measurement and a traditional dump. It requires that you have both a TSMP frequency measurement device and a TSOP device connected to pins 3 and 2 respectively. We would like to be able to find a way to measure the frequency and to decode the bit patterns using only a TSMP frequency device but it would take a very large buffer to do so. We might eventually come up with such a method that would only work on devices such as an Arduino Zero or other ARM-based Arduino compatible devices rather than the traditional limited 8-bit AVR devices.

## 2.3 How to Send IR Signals

In the previous section we showed you how to receive and decode IR signals. We showed you how to control a servo or other device using low signals. However another major application of this library is to transmit IR signals. People have used the sending ability to create their own custom TV remotes or to control IR controlled toy robots and drones.

We already showed you how to create an IR transmitter circuit at the beginning of this tutorial. We highly recommend you use the version that has a transistor to boost the power of the signal. In our hardware section there are other schematics for even more powerful IR transmitter systems.

This section presumes that you already have been through the receiving tutorials in section 2.2 and have decoded some IR signals using the various example sketches we have provided.

### 2.3.1 Simple Sending of an IR Signal

We will start with the simplest possible example we can design. Even though it is not very useful it will demonstrate some principles. Below is a listing of IRLib2/examples/send.ino

```
/* send.ino Example sketch for IRLib2
 *  Illustrates how to send a code.
 */
#include <IRLibSendBase.h>    // First include the send base
#include <IRLib_P01_NEC.h>
```

```
#include <IRLib_P02_Sony.h>
#include <IRLibCombo.h>

IRsend mySender; //declare a sender object

void setup() {
  Serial.begin(9600);
  delay(2000); while (!Serial); //delay for Leonardo
  Serial.println(F("Every time you press a key is a serial monitor
we will send."));
}

void loop() {
  if (Serial.read() != -1) {
    //send a code every time a character is received from the
    // serial port. You could modify this sketch to send when you
    // push a button connected to an digital input pin.
    //Substitute values and protocols in the following statement
    // for device you have available.
    mySender.send(SONY,0xa8bca, 20);//Sony DVD power A8BCA, 20 bits
    //mySender.send(NEC,0x61a0f00f,0);//NEC TV power
button=0x61a0f00f
    Serial.println(F("Sent signal."));
  }
}
```

Although we could have done #include <IRLibAll.h> we wanted to show you that sending protocols using the same combo system as decoding used. Rather than start with the base decoder this time we include IRLibSendBase.h followed by the header files for the protocols we want to use. The same restriction on keeping them in numerical order applies as was discussed in the decoding section earlier. After all of the protocols have been included we also include IRLibCombo.h as we did before. It will create a combination sending class called IRsend and we will create a single instance of that object.

In this particular example we could have omitted the combo file and created an instance of the Sony sender class which would be called IRsendSony however we made a multiple protocol sending class just so we could illustrate how to do so.

Note we did not create a receiver or decoder object because we are not going to receive or decode in this particular example.

In the setup function we initialize the serial monitor and print a prompt message. There is no initialization necessary for the sending object. In the main loop, we check the serial input port for any keypress. Depending on how you have set up your serial monitor you may or may not need to press enter to send a character. Every time we see a character received, we will send a particular signal using the Sony protocol using the statement

```
mySender.send(SONY,0xa8bca, 20);//Sony DVD power A8BCA, 20 bits
```

This happens to be the power button on my Sony DVD player. I obtained that value by using the dump sketch and determined that it was 0xa8bca and that it was a 20 bit code. Some

protocols do not vary the number of bits so the third parameter is ignored. We've given you another example of how to send an NEC code using the same sender object. You can comment out the Sony line and un-comment the NEC line. NEC protocol is always 32 bits. We could have put 32 as the third parameter but as we mentioned that protocol ignores what you put there. For your purposes you will probably want to substitute a protocol and code value of your own. Be sure to also include the protocol include file in the list of files at the top and maintain protocol number order.

As we said this is the simplest sketch we could come up with but it isn't very useful. You could modify it to use pushbuttons connected to digital input pins of your Arduino. Depending on which button is pushed you would send a different signal. We like using example sketches that rely on the serial monitor for input because it does not require you to wire up additional hardware in order to play with the program.

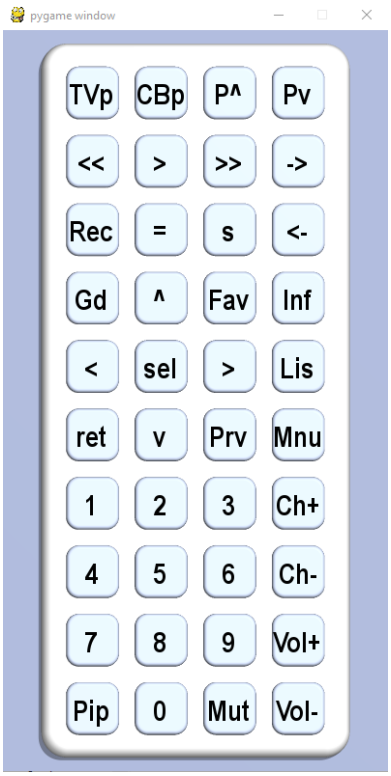
### 2.3.2 A More Interesting Sending Example

Although we will not re-create the entire listing here, we will be using the example sketch named IRLib2/examples/serialRemote in this section. Because we want to use all of the supported protocols we begin with including IRLibAll.h. This will include some things we don't need but this is just an example.

You should upload this sketch and open your serial monitor. You will be prompted to enter a protocol number, a data value in hexadecimal, and the number of bits. The protocol number and number bits should be in decimal. Do not include the 0x prefix in the hexadecimal number. After entering the data you should press enter and the program will send your data. If you omit the number of bits it will default to zero. Using the sample sketch you can send it a signal in any of our supported protocols by typing the data into the serial port.

Typing data into the Arduino IDE serial monitor is not a very convenient way to send signals but there are other ways we can send to the serial port. You could write an application program on your PC to send the data to the serial port. As example we have included a Python script in the file IRLib2/examples/serialRemote/serialRemote.py.

When you run the program it displays a virtual remote control on your screen that looks like the image on the right. When you click on one of the buttons, it will send serial data to your Arduino and the Arduino will send the IR signals.




It is written in Python 3 and will likely not work in Python 2. You must have also installed the pyserial and pygame packages which you can find here.
https://pypi.python.org/pypi/pyserial
http://www.pygame.org/download.shtml

The installation of Python 3 and the support packages is beyond the scope of this tutorial but there are plenty of tutorials online that can show you how to download and install those packages.

You will have to modify the serialRemote.py in several places to customize it to your needs. On approximately line 13 you'll have to modify this line

```
ser = serial.Serial('COM11', 9600)
```

to use the proper serial port and baud rate. Also beginning at approximately line 28 you will see a line like this

```
label_text=("TVp", "CBp", "P^", "Pv",\
        "<<", ">", ">>", "->",\
```

You may edit these 40 text labels to whatever symbols you want on your remote. Then at approximately line 52 you will see an array of IR codes stored as text strings like this

```
IR_Codes= ("3,180c,13","5,37c107","5,36d924","5,37d904",\
          "5,37291a","5,37990c","5,36293a","5,36b129",\
```

In our example the first button is the TV power button for a Magnavox TV which uses RC5 protocol. That is protocol 3 and the power code is 0x180c. It uses a 13 bit variety of the code. The next button is for my cable box which uses protocol 5 Panasonic_Old. Because that particular protocol always uses 22 bits, we do not need to specify the number of bits. Note that each sequence is enclosed in quotation marks and separated by commas. You will have to substitute your own values for these codes.

You could also implement your own application in any programming language that you wish as long as it is capable of sending a string of text to a serial port. Although we have laid this out like a TV remote, you could create an application that looks more like a game controller or perhaps virtual joysticks. As long as you can send serial data to the Arduino you can do anything.

One more thing… Because we believe in keeping everything as open source as possible, in the same folder we have also included the source code that we use to create the remote background image with its array of 40 blank buttons. The image was rendered in the Persistence of Vision Ray tracer or POV-Ray as it is often called. Although you do not need that program because we have already included the rendered image, you can find out more about this free program at http://povray.org/.

### 2.3.3 Sending Raw Timing for Unknown Protocols

In the receiving and decoding tutorials we said that if you had an unknown protocol that you absolutely needed to decode and did not want to try to implement a custom decoder, that you could use the hash decode system to turn the signal into a unique 32 bit binary number. Unfortunately there is no way to reverse engineer that 32 bit binary number back into its original signal for resending purposes. There is a way however that you can re-create an unknown signal using raw timing data. The problem is that it takes an array of 16-bit values with enough elements to store the heading information and any other marks and spaces in the signal.

We have provided you with 2 sample sketches that would facilitate this process. The first is IRLib2/examples/rawRecv which allows you to capture the raw timing data similar to the "dump" sketch. However it dumps the data in a form that you could easily cut and paste into another program which you will find in IRLib2/examples/rawSend.

You should load the rawRecv program and open the serial monitor. Note that this program makes use of the IRrecvPCI receiver. Although the other two receiver classes will work, the PCI receiver gives the most accurate results when trying to capture the exact timing of an unknown protocol. Note that we did not include a decoder class because we are going to access the data directly. There is a global array declared by IRLib named recvGlobal.recvBuffer which is an array of 16-bit timing values. The number of values in the array is given by recvGlobal.recvLength. For reference purposes both of these variables are declared in IRLib2/IRLibGlobals.h which you need not include in your sketch because it is already included when you include any of the receiver classes.

Aim your remote at the receiver and push the button just once. Here is some sample output from the sketch.

```
Ready to receive IR signals
Do a cut-and-paste of the following lines into the
designated location in rawSend.ino

#define RAW_DATA_LEN 42
uint16_t rawData[RAW_DATA_LEN]={
   2406, 626, 1206, 634, 582, 618, 1230, 610,
   614, 610, 1214, 602, 622, 602, 582, 642,
   542, 682, 1214, 602, 542, 682, 1222, 602,
   1214, 634, 1190, 634, 1190, 634, 566, 650,
   590, 634, 1190, 634, 590, 634, 1190, 634,
   566, 1000};
Do a cut-and-paste of the following lines into the
designated location in rawSend.ino
```

You should then drag your mouse across the #define line and the following array of data and then do a cut-and-paste into the designated area of the rawSend program. Note that the rawSend example begins with the following lines…

```
#include <IRLibSendBase.h>    //We need the base code
#include <IRLib_HashRaw.h>    //Only use raw sender

IRsendRaw mySender;
```

It starts with the send base code followed by the header containing the code for hash decoding and raw sending. Even though we cannot use raw send to send a hash code, we put both the hash and raw code in the same module because they both deal with unknown protocols.

Like the earlier send examples, you should upload the program and open the serial monitor. Every time you send a character it will transmit the raw data and re-create the original IR signal that you captured with rawRecv. Depending on how you have the Arduino IDE serial monitor configured you may or may not need to press the enter key to send the code.

Of course the disadvantage to this method is it takes an entire array of 16-bit values just to store a single code so it is not very practical for most purposes. You can achieve some savings by putting the array in program memory rather than RAM memory but that will complicate things. We will leave that as an exercise for the reader.

### 2.3.4 Testing Bit Patterns

We will describe one other sending example that is not of much practical use that might be interesting just the same. We use the program IRLib2/examples/pattern to verify that our sending and decoding routines really work. We use 2 Arduino devices with one of them configured for sending using this "pattern" example. We set up a second Arduino with a receiver using the dump program. Upload both programs and open both serial monitors. On the sending Arduino you should enter a number from 1 through 11 as a protocol number. It will then transmit one or more varieties of patterns of bits that should properly decode on the other Arduino.

Note that the bit pattern sent by this program may or may not be actual valid signals for that particular protocol. Some protocols have rules which divide the bit patterns up into different fields. Some of those fields might be a checksum or some other verification system to ensure that a proper pattern has been received. We are just sending random data to test out code. Sometimes it's something visually easy to spot like "1234abcd" or sometimes it's a bit pattern with all of the bits on.

For more details see the individual sections of the documentation for each of the supported protocols. There it explains the various variations within a particular protocol.

## 2.4 Sending and Receiving in the Same Program

There are some special concerns when you both send and receive IR data in the same program. To illustrate this we will use the program IRLib2/examples/record.

You should upload the program and open the serial monitor. You should send it an IR signal. If it recognizes it as one of the seven protocols it will record the protocol number and any other data necessary to repeat the signal. If it is not a recognized protocol, it will record the raw time values. Then if you press any key in the serial monitor it will retransmit the most recently saved received signal.

Some protocols such as NEC use a special repeat sequence. If you want to transmit the repeat sequence you should press the "r" key into the serial monitor. Note that this sketch does not test whether or not a given protocol actually supports the repeat code or not. We could have modified this to make sure that the repeat command only works for NEC or other protocols that use a special repeat ditto sequence. See the section on the NEC protocol in the reference section for more details.

Some protocols such as RC5 and RC6 also use a special toggle bit. We have implemented that feature in this program. See the reference section for those protocols for more details about toggle bits.

We will not describe all of the details of this program however there are some details we want to point out. Note at the beginning we have included both the sending and decoding base headers followed by seven of the protocols and the raw send header followed by IRLibCombo.h to create both sending and decoding combo classes.

The setup function initializes the serial monitor and prints some prompt messages as well as initializes some variables. Note that we call myReceiver.enableIRIn() as is typical anytime we are receiving data.

Let's take a closer look at the loop portion of the program which we reproduce here…

```
void loop() {
  if (Serial.available()) {
    uint8_t C= Serial.read();
    if(C=='r')codeValue=REPEAT_CODE;
    if(gotOne) {
      sendCode();
      myReceiver.enableIRIn(); // Re-enable receiver
    }
  }
  else if (myReceiver.getResults()) {
    myDecoder.decode();
    storeCode();
    myReceiver.enableIRIn(); // Re-enable receiver
  }
}
```

If there is serial data available it means that we want to transmit a previously received code. Before we look at how we do that, let's look at the "else" portion of that statement. We check to see if there has been a signal received and if getResults() returns true then we attempt to decode it. Note we do not care if myDecoder.decode() returns false because we're going to process unknown protocols anyway. After we have stored the data we call enableIRIn() to restart the receiver and wait for another code to come in.

Now let's look at the first part of the if statement where we have received a character from the serial monitor telling us to play back the recorded signal. If data is available we read a character to see if it is a letter "r". If it is an "r" we substitute the saved value with a repeat code. We check a flag to see if we have previously recorded any code. We don't want to try to send until we have received at least one code. Our sendCode() function will then send the data. After a brief delay we must reenable the receiver. That may seem strange but we have to call enableIRIn() after the send. That is because the sending code inside IRLib makes use of a hardware timer that is also used by the IRrecv class. Any use of the sending code automatically disables receiving whether we want it to or not. So whenever we have a program that both sends and receives you must always reenable the receiver after any sending.

Technically this should not be necessary for the IRrecvPCI or IRrecvLoop receiver classes. Theoretically it only has a conflict with the IRrecv receiver. However to make the logic

of some other parts of the library work well you should still do this whether you are using the IRrecv receiver or the other two.

# 3. Implementing New Protocols

In 2012 when I first started building my own IR remote using Ken Shirriff's IRremote library, one of the first things I had to do was add additional unsupported protocols for devices I wanted to control. Being unfamiliar with the code it was difficult for me to figure out what part of the code dealt with receiving and what part with decoding. I also noticed that while it was written in C++ it did not take advantage of the benefits of object-oriented coding techniques. For my own use I began rearranging and rewriting the code that became the basis for the original IRLib. My goal has always been to make it easier to add new protocols. The major reorganization that has come with IRLib2 finally has gotten the package to the point where it is as easy as it is ever going to be for other programmers to implement new protocols. I have been promising to write this section of the documentation since the very beginning in 2013 when I first released the code.

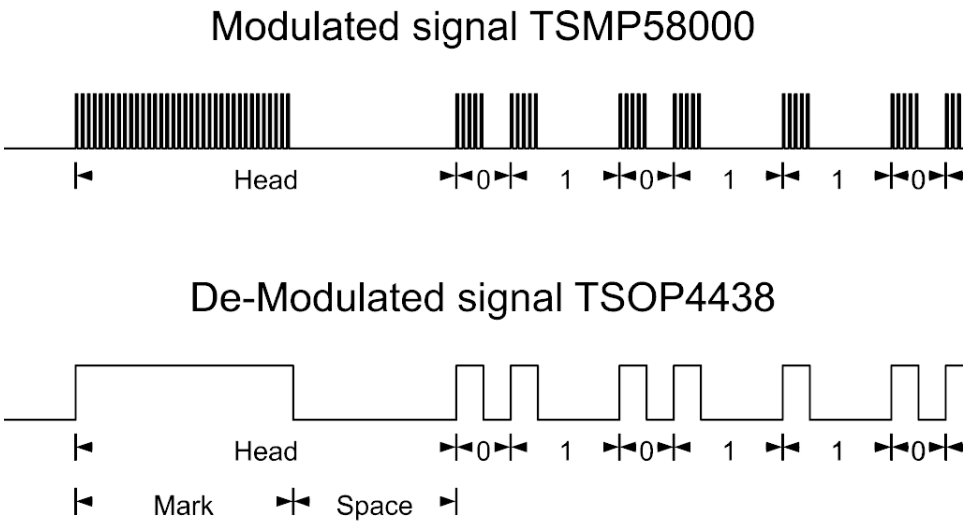
I've always felt that a well-written library isolates the application programmer from much of the internal processes. It should present you with a simple clear API and the rest of it should be "magic" that you don't care how it works. However writing a new protocol does require that we look beneath the hood to see the engine that drives everything. You also need to know a little more about how IR signals are encoded and decoded in a bit more detail than we have given you before. While many of our examples required little programming skill on the part of the application programmer and allowed you to sort of follow a recipe to create a program, the implementation of a new protocol is not for novice programmers. We assume you have a pretty good working knowledge of programming in C++. We also presume you know how to use the existing features of IRLib2. It will also be helpful if you read Appendix A. Its primary purpose is to document the reorganization of the code from IRLib1 into the current form in IRLib2 but it provides good insight on how protocols are integrated into the library.

We will begin with a detailed tutorial on how IR signals are encoded even though we have given you bits and pieces of that information throughout the documentation and previous tutorials. We will then show you how to analyze a dump of an IR signal that will provide clues to it's internal workings. We will refer you to Appendix C where you can learn about reference material describing various protocols in something called IRP notation. Finally we will show you how we used all of this knowledge to implement some of the protocols we already support.

## 3.1 Understanding IR Signals

IR signals are transmitted using pulses of infrared light that we call "marks" interrupted by periods of time we call "spaces". It is the timing and pattern of these marks and spaces that are used to encode the data. In turn the marks are themselves modulated by a frequency ranging from approximately 36 kHz through 58 kHz. Each protocol uses a particular modulation frequency. Some protocols have variations which are identical except for the modulation frequency. In order to analyze a particular IR signal, you need to determine its modulation frequency as well as the pattern of marks and spaces.

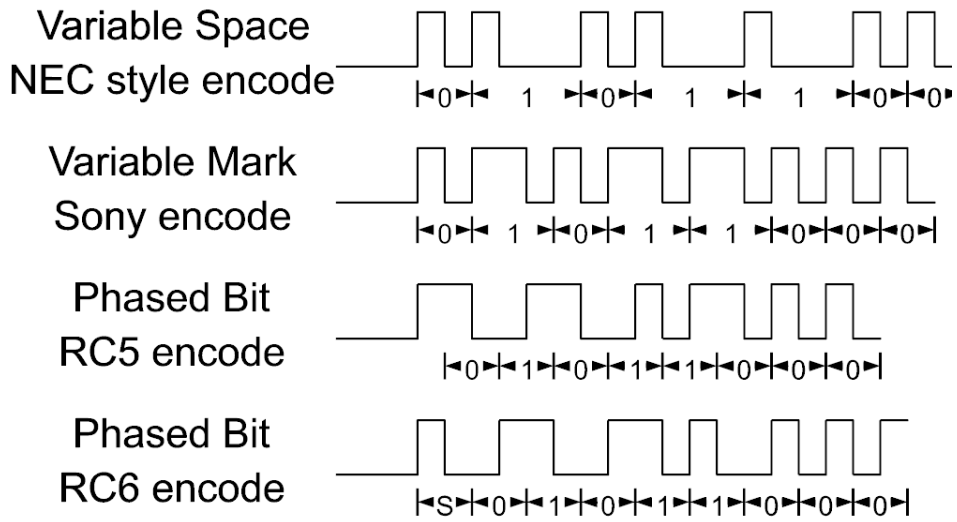
We will measure the frequency using a TSMP58000 receiver device that can measure the modulated signal wavelength. You also use a TSOPxxxx device to analyze the pattern of marks and spaces. The image below illustrates what those signals would look like this on an oscilloscope or digital logic analyzer.




As you can see the TSOP device filters out the modulation and gives a clean signal. Note that although we have illustrated the marks as being a high voltage and the spaces as low, both of these devices are actually normally high. Then when they receive an input signal left, the output is pulled to ground. We believe its easier to understand signals conceptually if we think of the marks as being "on" and spaces as "off". In fact that is the way the IR LED is transmitting it.

Most but not all protocols begin with and especially long mark/space pair which we call a header. This long initial mark not only helps identify the particular protocol but also allows for the automatic gain control to have plenty of time to adjust itself so that the remainder of the signal comes through clearly. When you are analyzing an unknown protocol you should transmit very close to your receiver to get a clean signal to begin with so that you can accurately measure the length of this initial mark.

In the above illustration, you can see that the data marks are all of the constant length and that the spaces between vary to indicate whether it is a zero or a one bit. The vast majority of protocols that you will encounter use this encoding system. The Sony protocol uses the opposite system. It uses constant length spaces but it varies the length of the marks as seen in this illustration.

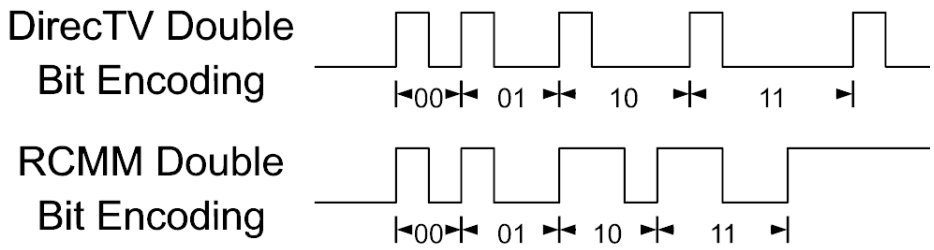



In this illustration the top part illustrates the variable space encoding used by NEC and most other protocols. The second row shows the Sony system which uses variable marks instead of variable spaces. It is relatively easy to look at a graph of those signals and see that they are either variable spaces or variable marks. Once you get good at it, you can look at timing values and figure out the bit patterns in your head.

Not all protocols are as easy to figure out visually. The RC5 and RC6 protocols and their several variants use something called phase encoding. With this type of encoding, a zero or one is represented by a transition from low to high or vice versa. The third row shows the RC5 protocol which encodes a zero as a transition from high to low and encodes a one by a transition from low to high. The overall timing of a particular bit remains constant but the way in which it behaves during that time period is how you determine zeros and ones. RC5 begins with a transition to high that lasts one time period. You can see that the initial zero bit actually begins in the middle of that initial mark. Had the first bit been a one it would have a transition to low after one time interval and then the low to high transition would indicate a one. Each bit takes two time periods.

The RC6 protocol also uses phase encoding but it reverses the logic. A transition from low to high designates a zero and a transition from high to low indicates a one. It always begins with a start bit that we have designated with the letter "S". Although we have not illustrated it here, the RC6 protocol also has a special bit in the middle of the sequence which is double the length of a normal bit.

Some protocols encode 2 bits at a time. The DirecTV protocol uses variable spaces but it doesn't just have a long and a short space to designate a zero or a one. It uses 4 different lengths of spaces to designate the bit sequences 00, 01, 10, and 11 as can be seen in the illustration below.




Another protocol that encodes 2 bits at a time is the RCMM protocol sometimes called Nokia protocol. It is used by U-Verse cable boxes. It encodes bits by varying the length of both the mark and the space. The mark gives you the value of the first of the two bits with a short mark meaning zero and a long mark meaning one. The space gives you the value of the second of the two bits with the short space indicating zero and a long space indicating one. It concludes with a very long mark followed by a very long space.

There is another protocol that we are aware of that we have not yet supported that uses a completely different system. The Async protocol sends data as if the IR LED is connected directly to a UART sending asynchronous serial data. A value of one as indicated by a mark in a value of zero is indicated by a space. Typically it begins with a start bit followed by 8 or 16 databits perhaps followed by a parity bit. So for example an 8-bit value of zero would be a mark followed by 8 time periods of space followed by a parity bit depending on whether you use odd or even parity. Similarly an eight bit value of 255 would be nine time periods of mark followed by a parity bit that might be a mark or a space. We may attempt to create a decoder for this protocol in a future version.

To our knowledge this represents all of the types of bit encoding that are in use by most consumer electronic devices. So as you are analyzing your unknown data stream you should be on the lookout for these kinds of patterns. As we mentioned earlier the use of constant length marks and variable length spaces is the most common way of encoding bits.

Although most protocols begin with a header of an exceptionally long mark/space, not all of them do. Some will use different headers to designate an initial frame versus a repeat frame. Others drop the header altogether for repeat frames. You also might encounter marks or spaces in the middle of the sequence that have nothing to do with the actual data bits being encoded. They just served to separate various bit fields within the protocol. The Samsung36 protocol is one such protocol. It is a 36 bit protocol which begins with a header consisting of a mark/space of 4500, 4500 microseconds. It then sends 16 bits of data using typical variable space encoding. Then there is a 500 µs mark followed by a 4500 µs space. It then sends 12 more bits of data followed by 68 µs of space. Because that last of those bits consists of a mark and a space pair in effect that last space is simply 68 µs longer than normal. This is followed by a normal encoding of the final eight bits followed by a stop bit. This is probably the strangest pattern we have encountered but we explain it here to show you that it's not always just a header followed by a bunch of data bits.

## 3.2 Analyzing Unknown Protocols

To illustrate how to analyze a05 unknown protocol, we are going to use some protocols that we already know how to decode but we are going to act as if they are unsupported. We will leave them out of the stack of protocols that we will include in our analysis program so that it will

fail to recognize that particular protocol. Our main example is going to be protocol 9 GICable. The steps we are outlining are very similar to the ones we took when we first implemented this protocol.

### 3.2.1 Using the Analyze Program

The first step in creating code to handle a new protocol is to record some signals from your remote and do some analysis. If you have access to a TSMP58000 to measure the frequency of the protocol you should do so using the example sketch IRLib2/examples/freq. You will connect your TSMP device to pin 3, upload the sketch, and send it some signals. The output will look something like this…

```
Interrupt=0 Pin=3
Avg. interval(us):25.49     Aprx. Frequency(kHz):39.23 (39)
```

It tells us that the frequency is approximately 39 kHz. We happen to know that this is the correct frequency but if this was an unknown protocol you would not know that. If you do not have access to a TSMP device to measure the frequency we will presume for the time being that it is 38 kHz because that is the most common frequency.

Next we suggest you use our typical dump example and look at some signals. We highly recommend that you use the PCI or Loop receiver rather than the 50 µs receiver because the PCI is the most accurate. Try pressing the same button on your remote repeatedly. Press and release it quickly so you get just one signal and do not hold down the button. Wait a second or two until you are sure the output has completed on your serial monitor and take another sample using the same button. Make sure that you are getting similar results between the two samples. Let's look at one of the samples.

```
Decoded Unknown(0): Value:0 Adrs:0 (0 bits)
Raw samples(36): Gap:47026
  Head: m8922  s4494
0:m482 s4498   1:m506 s4450       2:m526 s2214     3:m530 s4430
4:m522 s4474   5:m482 s2234       6:m534 s2214     7:m498 s2246
8:m506 s2214   9:m558 s2190       10:m526 s2218    11:m478 s2266
12:m506 s2210  13:m558 s2190      14:m446 s4534    15:m502 s2242

16:m502
Extent=68902
Mark  min:446   max:558
Space min:2190  max:4534
```

The signal starts out with a long mark and space with the mark being roughly twice the length of the space. That's a good sign. Many protocols have headers that look like that. We see that the marks run between 446 and 558 µs. We also note that all of the spaces are much larger than that and that they cover a wide range of values from 2192 up to 4534. If you look carefully the values are clustered high and low. Most of them are about 2200 or 4400 with nothing in between. That's also a very good sign that this is a well behaved protocol. It appears that it uses the most common form of encoding bits with fixed length marks and variable length spaces. Note that these minimum and maximum values do not include the mark/space pair in the header. It only includes the data bits. Just by eyeballing it we would say that the marks are approximately 500 µs and that the spaces are either 2200 or 4400. If we presume that the

shorter spaces indicate a zero and the longer spaces indicate a one then we see that the pattern of bits is 1101 1000 0000 0010. The final mark is typical of variable space encoding. You need that last mark so that you know the length of the final space.

We told you to press the same button repeatedly to see if you get the same results. The header should still be the approximate same length of mark/space. The data marks should be approximately the same as well as the high and low values for the spaces. But most important we went to see if successive presses of the same button give you a high and low space lengths in the same locations of the bitstream. Because it does, we now know that this protocol does not use toggle bits. If it did use toggle bits you would see the length of one of the space values swap back and forth between the low and the high values.

Next you should run our analysis sample program which can be found at IRLib2/examples/analyze. It is an enhanced version of our usual dump routine. However rather than analyzing a single transmitted signal, it is designed to take multiple samples of the same signal and average the results. In addition to averaging the timing value for each location of a mark and space in the stream, it also does other useful analysis. After uploading the sketch and opening your serial monitor, again you should briefly press and release the same button multiple times. Wait at least a second or two and press the same button again. Do this until you have obtained five or six samples. If you take many more than that, there is a chance that the accumulated time will overflow and you will get incorrect results. Below is the end result of five samples that we want to analyze.

```
Samples=5
Decoded Unknown(0): Value:0 Adrs:0 (0 bits)
Raw samples(36): Gap:244
  Head: m8955  s4457
0:m504 s4461   1:m512 s4463       2:m504 s2238     3:m517 s4449
4:m492 s4486   5:m494 s2239       6:m509 s2221     7:m530 s2214
8:m492 s2239   9:m508 s2238       10:m490 s2242    11:m512 s2224
12:m481 s2254  13:m517 s2234      14:m505 s4453    15:m513 s2225

16:m497
Extent=68869
Mark  min:481   max:530
Space min:2214  max:4486

          #    Low   High  Avg.
Mark     17    481   530   504
Space    16    2214  4486  2930

Marks           #    Low   High  Avg.
Above Avg      9     505   530   513
Belows Avg.    8     481   504   494

Spaces          #    Low   High  Avg.
Above Avg.     5     4449  4486  4462
Below Avg.     11    2214  2254  2233
```

```
Ratios          vs   Marks  vs   Spaces
                Below Above Below Above
Head Mark       17.97 17.31 4.02  2.00
Head Space      8.96  8.63  2.00  1.00

                      Low   High  Avg.
Space vs Avg Mark     4.38  8.79  5.76
Mark vs Avg Space     0.17  0.18  0.17

Mark Above/Below=     1.04
Space Above/Below=    2.01
```

The first portion of the output is created by our usual myDecoder.dumpResults() report method. Note however that these values are not the values from the fifth sample rather they are the average values of all five samples. The next section of our analysis repeats some of the information from the original dump but it has other info as well. It tells us that there are 17 marks and 16 spaces. This is likely a 16-bit protocol with the final mark. It tells us that the average length of a mark is 504 µs. That is useful. However the fact that the average space is 2930 is not very useful because there are no spaces in that range. What we really want to know is how many values are above average and how many are below average.

The next section reports how many marks are above average, how many are below average, and the averages of those. Because we've already determined that marks do not vary significantly throughout the stream, we are not very interested in this section. However if we were analyzing a Sony protocol that use variable length marks then this information would be very useful.

The next section reports the same statistics about spaces and this is what we really want to see. It tells us that five of the spaces are above average at 11 of them are below average. The lowest of the large values is 4449 and the highest is 4486. Similarly the lowest of the low values is 2214 and the highest is 2254 with an average of 2233.

The final section reports some ratios. These ratios are useful to determine if the values we are seeing are even multiples of some base value. For example the NEC protocol as a base timing value of 564. The header mark is 16 times that value. They header space is eight times that value. A bit of zero is represented by 564, 564 and a bit of one is represented by 564,3*564. So if such a relationship exists in our data, we want to know that. Some of these ratios will be useful and some will not.

The first set of ratios compares the header mark with the below average mark and the above average mark. We're getting values about 17.5 or 18. The header space is compared to the below average space and the above average space. We see that the header space is exactly the same as the higher space value and is double the lower space value.

One ratio that is noteworthy is that the low space is about 4.3 times the average mark in the high space is nearly 8.75 times the average mark.

From what we've seen we could probably now write a decoder that would use a header of 8800, 4400. We would encode a zero as 500, 500 and we would encode a one as 2200, 4400. It would be a 16 bit protocol with a trailing mark. As an experiment we did create such a

decoder and it was able to correctly decode the signal. If we would similarly create an encoder using these values, in all likelihood it would operate our equipment.

You might be able to simply stop here and declare success. You may have been able to decode the signals and re-create them sufficiently for your purposes. But if we really want to add this as a newly supported protocol for the library. We should extend the analysis further. For example we can't be sure that these timing values are 100% accurate. Recall that different receiver devices and circuitry as well as the conditions under which you are receiving the signal can cause minor variations. See the documentation about the ""myReceiver.markExcess" value. Our receiver circuitry typically under reports the actual length of a mark and over reports the actual length of a space. We have put in a fudge factor of 50 µs but we don't know that that's always 100% correct.

Furthermore at some point during your analysis you should try pressing a different button. Close your serial monitor and reopen it running the analyze sketch. This will clear out your previous values. Try recording a different button. We still should get marks that are approximately 500 µs and we should still get spaces that cluster around 2200 and 4400 µs although the long spaces will be in different locations giving you a different pattern of bits.

There is one more experiment that you should do. We have been telling you to press and release a button quickly to give a single frame data. But at some point you should try holding the button down so that it will repeat. Choose a button that is likely to want to auto repeat. For example normally you would not want the pause button to repeat but the arrow key should automatically repeat. See if you get different results as a repeat code. You might want to put your hand over the end of the remote to block the signal, press and hold the button, and then remove your hand and point it at the Arduino receiver so that you get only repeat signals and not the initial signal. In this case we got something really dramatically different. Here is a partial listing of what we recorded for a repeat signal.

```
Samples=5
Decoded Unknown(0): Value:0 Adrs:0 (0 bits)
Raw samples(4): Gap:10810
  Head: m8930  s2234
0:m518
Extent=11682
Mark  min:518   max:518
Space min:32767       max:0

    #    Low   High  Avg.
Mark     1     518   518   518
Space    0     65535 0     4294967295
```

It looks like we got a header about the same size mark as usual but a 2200 space rather than the usual 4400 space for the header. This is all followed by a single mark and there are absolutely no data spaces. This is obviously a ditto code similar to the one used by NEC protocols and their variants. With a ditto such as this, it's difficult to determine how much blank space is necessary following the initial signals. If we really want to implement this protocol it's time to do some research to see if we can identify what it is and find out if our timing measurements were accurate and if there are other features of the protocol that we have not identified. This research may not be possible if it is an especially obscure protocol. For example

we have used the above procedures to reverse engineer the protocol used by an IR control to a helicopter and an IR control for a toy robot dinosaur. We could find no online documentation to support our measurements or conclusions but we were able to write code sufficient to control these devices with an Arduino. That was all we really wanted. The research we are proposing in the next section is only if we really want to permanently add our newly discovered protocol to the library.

### 3.2.2 Researching and Identifying an Unknown Protocol

There are lots of places on the Internet where you can research IR protocols but in all my work the most extensive source is this particular webpage created by John S. Fine who has done extensive research on IR protocols. This webpage is my "bible" when trying to understand unknown protocols.

http://www.hifi-remote.com/johnsfine/DecodeIR.html

At this point you should stop reading here and read Appendix B which tells you how to interpret the information on that particular webpage or any other sources you find that describe an IR signal using what is known as IRP notation. This is a kind of shorthand for describing all of the properties of a particular protocol. As we proceed here, we presume you have read Appendix B and have that webpage called up on your browser so you can search through it.

Admittedly in some respects we are cheating here because we already know that this example is based on the GICable protocol. But I am outlining for you the actual steps that I went through in trying to determine that this was in fact that protocol. When I was presented with the signals, I didn't know what I was looking for.

In some respects the fact that we identified that is using a specialized sequence for a repeat code makes our job easier. There are only so many protocols that use these special ditto sequences. We know it isn't 48-NEC because it's not a 48 bit protocol. The Aiwa protocol uses dittos but it is 42 bits long. The Dgtec is 24 bits long. The G.I. Cable protocol is 16 bits and uses dittos so we should start to get excited that we have a good candidate. We should hold off celebrating yet because we have not exhausted all of the possibilities. As we continue to search that webpage we see that Kathrein uses dittos but the ditto includes part of the data and the number of bits is wrong. The Teac-k protocol is the wrong number bits as well.

Just because it's 16 bits and uses dittos we haven't completely verified that we've got the right protocol. It could be a protocol unknown to this particular reference material or it could be a variation that is not documented from this particular source. We need to look at the IRP notation more closely. The specification says…

{38.7k,490}<1,-4.5|1,-9>(18,-9,F:8,D:4,C:4,1,-84,(18,-4.5,1,-178)*) {C = -(D + F:4 + F:4:4)}

The first item is the modulation frequency. It specifies 38.7 and we measured 39. That's well within our tolerances. The base time unit is 490 µs. A zero is encoded by 490, 4.5*490 or in other words 490, 2205. A one is encoded by 490, 9*490 which is 490, 4410. Finally the header is 18*490, 9*490 which is 8820, 4410. There are 16 bits of data followed by stop bit followed by 84*490 of blank space before the next frame. This lead out time is not something we generally worry about too much. Finally we get to the definition of the ditto which is a header of 8820, 2205 followed by a single mark and a very long space.

Although we estimated the base time unit at about 500, a value of 490 is well within our tolerances. We should modify our decoder and encoder to use the values specified in the IRP specification rather than our actual measurements. Now that you know the name of the call you are using, you may want to do some online searches for other documentation on that protocol. Although we really like Mr. Fine's reference material, there was an instance where another source used slightly different timing values for the DirecTV protocol. Their explanation of the protocol seemed more detailed and logical so we went with their values.

Sometimes we have a good idea to begin with what protocol we are using. Although not all Samsung devices use a Samsung protocol (many use NECx for example), the name of the manufacturer might suggest you should try out a particular protocol first. You look up the IRP specification for your particular manufacturer and then sample some signals and see if you get results that fit with the specification. So in this process you start with the IRP definition and then verify it with the samples rather than starting with samples and searching for an IRP to match.

In the next section we will show you how to actually implement the decoding and sending classes for your particular protocol based on all of the information we have learned about it.

### 3.2.3 Implementation Using decodeGeneric and sendGeneric Functions

The process of actually implementing the protocol involves the following steps.

- Start with an empty template where we will fill out the C++ code to implement it.
- Write the code for the send routine
- Write the code for the decode routine
- Make changes in other files to add the protocol to those we support

Because this particular GICable protocol uses traditional fixed length marks and variable length spaces and has a traditional header and does not do very much else differently, it is a candidate for using generic decode and send functions. These functions were created so they could be shared by multiple protocols and would cut down on the code size by eliminating the need to have the same algorithm implemented multiple times using different timing values. We can simply pass our particular timing parameters to these functions and it will do all of the work for us. In a later section we will show you how to write specialized send and decode routines from scratch if you are not able to use the generic routines.

#### 3.2.3.1 Preparing the Template File

Because we've already implemented this protocol as protocol 9 let's presume we are implementing a brand-new unsupported protocol. It will be identical to 9 but when we do this for real of course it won't be. As of writing this documentation, we are currently up to 11 supported protocols so let's presume we are making protocol 12 and we will call it Tutorial.

As you have seen, each protocol has its own header file that contains not only prototypes for the routines but actually contains the C++ code itself. See Appendix A for a discussion about why we did this. We have provided a template that you can copy and edit to make this process simpler. Start by loading the file IRLibProtocols/IRLib_P99_Additional.h and create a copy of it under the name IRLib_P12_Tutorial.h. The Arduino IDE cannot open library files so you will have to use an external editor. We happen to like Notepad++ for editing libraries.

Open the new file that we have created and do a search and replace on the string "99" and replace all occurrences with "12". Similarly researcher replace on "Additional" and replace all occurrences with the name of our protocol which in this case is "Tutorial". We are now ready to write a sending routine.

#### 3.2.3.2 Implementing the send Function

In your newly created file, look for the section of the code for sending. It looks like this…

```
#ifdef IRLIBSENDBASE_H
class IRsendTutorial: public virtual IRsendBase {
  public:
    void IRsendTutorial::send(uint32_t data) {
    //void IRsendTutorial::send(uint32_t data, uint32_t data2)
      /*********
       *  Insert your code here.
       *********/
    };
};
#endif  //IRLIBSENDBASE_H
```

Because there are no variations of this protocol then we do not need to specify number of bits or any other parameters beyond the data to be sent, we can use the single parameter version of the send routine. You can delete the commented out alternative just below it and the comments which say "Insert Your Code Here." The code will consist of a single call to our sendGeneric(…) method. It is implemented in IRLibProtocols/IRLibSendBase.cpp and the prototype can be found in IRLibProtocols/IRLibSendBase.h as follows…

```
  void sendGeneric(uint32_t data,  uint8_t numBits,
                   uint16_t headMark, uint16_t headSpace,
                   uint16_t markOne, uint16_t markZero,
                   uint16_t spaceOne, uint16_t spaceZero,
                   uint8_t kHz, bool stopBits, uint32_t maxExtent=0);
```

The parameter names are fairly self-explanatory. The last two parameters deserve some attention. There is a Boolean value that tells you whether or not to end with a stop bit. It is needed because the sendGeneric function is also used by the Sony protocol which has variable length marks instead of variable length spaces. It does not use a stop bit. As long as you have a protocol that uses variable length spaces, the stop bit flag should always be true. The maxExtent parameter is optional. It is used for protocols that should have an overall frame length of a particular value. Our particular protocol has a fixed length lead out of 48*490 so we will not use this parameter.

There are 2 other methods we may need to use in implementing our sending class. They are the mark(uint16_t) and space(uint16_t). They are also in the IRLibSendBase.h header file where the base class is defined. Obviously mark() transmits a signal for the specified amount of time and a space() pauses for the specified amount of microseconds.

If we wanted to ignore the ability to send a ditto sequence our code would look like this…

```
  void send(uint32_t data) {
    sendGeneric(data,16, 8820, 4410, 490, 490,
```

```
                4410, 2205, 39, true);
    space(37*490);
    }
  }
```

The sendGeneric function automatically concludes with a space equal to the length of a space for a one bit. In our case that is 9*490. The IRP notation says that it should conclude with 48*490 of a space so we will pad that out by sending 37*490 µs of space.

Things get a little more complicated if we want to implement the repeat function using the ditto. Here is how we implemented the full protocol in IRLib_P09_GICable.h.

```
  void send(uint32_t data) {
    if(data==REPEAT_CODE) {
      enableIROut(39);
      mark (490*18); space(2205);
      mark (490);
      space(220);delay(87); //actually 490*178 or "space(87220);"
      } else {
      sendGeneric(data,16, 490*18, 490*9, 490, 490,
                  490*9, 2205/*(4.5*490)*/, 39, true);
      space(37*490);
    }
  }
```

Because we're not using the generic function to send the ditto, there is some other overhead we need to take care of. You need to begin by a call to enableIROut and pass it the frequency in kilohertz in this case 39. We then need to specifically send a mark of 8820 followed by a space of 2205 followed by a mark of 490 as specified in the IRP notation. The final space should be 490*178 however that comes out to 87220 which is larger than our largest 16-bit integer that we can pass. What we need to do instead is split up between the space() and some other form of delay. Note that the space() method does more than just actually delay. It turns off the transmitter and then it delays the specified amount of time. So we must call space() first with an amount of time that it can handle. To make things easy we took the 220 out of the 87220. Then we can call the Arduino built-in function delay() which delays in microseconds not milliseconds. By calling it with the delay(87) we are actually delaying 87000 µs.

If we are not sending a ditto then we could just call the sendGeneric function as we described earlier and we pad out the remaining space. Note that 37*490=18130 which is within the limits of that 16-bit parameter so we do not need to split it up like we did before.

We invite you to look through other protocols such as NEC, Sony, Panasonic_Old, JVC, and NECx to see how we made use of the sendGeneric function.

#### 3.2.3.3 Implementing the Decode Function

Now we move on to the decoding side. In the bottom half of the protocol file find the following function that we need to fill out.

```
  bool IRdecodeTutorial::decode(void) {
    IRLIB_ATTEMPT_MESSAGE(F("Tutorial"));
    /*********
```

```
     *  Insert your code here. Return false if it fails.
     *  Don't forget to include the following lines or
     *  equivalent somewhere in the code.
     *
     *  bits = 32;   //Substitute proper value here
     *  value = data;      //return data in "value"
     *  protocolNum = ADDITIONAL; //set the protocol number here.
     */
    return true;
  }
```

Again we can take advantage of a generic function for decoding like we did for sending. A simplified version of our decoder that does not recognize the ditto would be as follows…

```
  bool decode(void) {
    IRLIB_ATTEMPT_MESSAGE(F("Tutorial"));
    if(!decodeGeneric(36, 18*490, 9*490, 490, 9*490,
         2205/*(4.5*490)*/)) return false;
    protocolNum=TUTORIAL;
  }
```

The first line of the code is a macro call that will print debugging information if you have debugging enabled. We will talk more about that in a separate section. If debugging is disabled, this line of code does not get compiled. Next we call the decodeGeneric(…) method from the base decode class. It is implemented in IRLib2/IRLibDecodeBase.cpp and the prototype in IRLib2/IRLibDecodeBase.h is as follows…

```
  bool decodeGeneric(uint8_t expectedLength,
                     uint16_t headMark, uint16_t headSpace,
                     uint16_t markData, uint16_t spaceOne,
                     uint16_t spaceZero);
```

Unlike the sendGeneric(…) method which can be used for both variable length marks and variable length space protocols, the decodeGeneric(…) Method can only be used for protocols using variable length spaces. Therefore we do not need to separately specify the length of a mark for a data bit zero versus a data bit one. We should also talk about the first parameter expectedLength. This is the number of entries in the decode buffer that we are using. Usually this is (bits*2) +4. That is because it takes 2 slots (a mark and a space) to encode a single bit and add to that the original slot which is the gap between times, a slot for the header mark and one for the header space. And finally we need one more slot for the stop bit. In our case that brings us to 36 slots. This is the number you will see in a traditional dump where it says "Raw samples"…

```
Decoded Unknown(0): Value:0 Adrs:0 (0 bits)
Raw samples(36): Gap:23290
  Head: m8970  s4454
…
```

Note that this relationship of (bits*2)+4 is not always true because some protocols have other marks and spaces that interrupt the stream or perhaps have a different type of header.

The decodeGeneric(…) method will return true if the particular pattern we specified is found and will return false otherwise. So our "if statement" says that if it was not true, we should return false. If it is true, there are some other things we need to do. Specifically we need to set the "protocolNum" to the value TUTORIAL which in our case will be 12. We will show you in a minute where that value will get defined. In general your decoder routines should also set the number of bits and value of the binary number we decoded. Later the user will access these values using myDecoder.bits and myDecoder.value. We do not need to set these in this instance because the decodeGeneric(…) routine does that for us.

If we want to support the ditto feature the code becomes a little more complicated. It would look like this…

```
bool decode(void) {
  IRLIB_ATTEMPT_MESSAGE(F("G.I.cable"));
  // Check for repeat
  if (recvGlobal.decodeLength == 4 &&
      MATCH(recvGlobal.decodeBuffer[1],490*18) &&
      MATCH(recvGlobal.decodeBuffer[2],2205) &&
      MATCH(recvGlobal.decodeBuffer[3],490)) {
    bits = 0;
    value = REPEAT_CODE;
    protocolNum= TUTORIAL;
    return true;
  }
  if(!decodeGeneric(36, 18*490, 9*490, 490, 9*490,
                    2205/*(4.5*490)*/)) return false;
  protocolNum=TUTORIAL;
  return true;
}
```

The first thing we do is check the decodeLength to see if we have the right number of data slots. If we were decoding the entire normal sequence we would compare this to 36 however the ditto uses only four spots. We ignore decodeBuffer[0] because it contains the gap between frames. The header mark is in element [1] and the header space is in [2]. The macro MATCH(v,e) compares the values to see if they are close enough together to be within our tolerances. The parameter "v" is the value we are testing and the parameter "e" is the expected value. The default tolerance is 25%. That value as well as the definition of MATCH can be found and IRLib2/IRLibDecodeBase.h and we will talk more about them later.

After comparing the mark/space for the header, we also compare the single mark which is supposed to follow them. If all of these match then we know that we have an appropriate ditto code. Because the ditto does not actually repeat the data, technically it has returned zero bits so we specified that by setting bits=0; Also we need to set "value" to the special value REPEAT_CODE which is defined as 0xffffffff in the file IRLibProtocols/IRLibProtocols.h.

### 3.2.3.4 Other Changes to Implement a Protocol

There are several other changes you need to make the other files so that the rest of the code in the library knows that there is a new protocol available. We begin with the file named IRLibProtocols/IRLibProtocols.h. Near the top of the file beginning at approximately line 13 you will find the following defines…

```
#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 SAMSUNG36 8
#define GICABLE 9
#define DIRECTV 10
#define RCMM 11
//#define ADDITIONAL_12 12 //add additional protocols here
//#define ADDITIONAL_13 13
#define LAST_PROTOCOL 11 //Be sure to update this when adding
protocols
```

This is where we define the protocol numbers. You will add a new define statement that would read…

```
#define TUTORIAL 12
```

Also be sure to edit the definition of LAST_PROTOCOL to be 12 instead of 11. You should open the file IRLibProtocols/IRLibProtocols.cpp where you will find the following function…

```
const __FlashStringHelper *Pnames(uint8_t type) {
  if(type>LAST_PROTOCOL) type=UNKNOWN;
  // You can add additional strings before the entry for hash code.
  const __FlashStringHelper *Names[LAST_PROTOCOL+1]={
   F("Unknown"),F("NEC"),F("Sony"),F("RC5"),F("RC6"),
   F("Panasonic Old"),F("JVC"),("NECx"),F("Samsung36"),
   F("G.I.Cable"),F("DirecTV"),F("rcmm")
   //,F("Additional_12")//expand or edit these
  };
  return Names[type];
};
```

Remove the slashes from the line that says "Additional_12" and edit it to read "Tutorial". These strings are used by dumpResults() and perhaps other places to identify the protocol name based on the number. The "F()" macro is a built-in Arduino feature that ensures that the strings of text are stored in program memory rather than in RAM memory. As always when adding an item to a list make sure that you have a comma between each item and no comma at the end.

The final step is to edit IRLibProtocols/IRLibCombo.h to add the support for the new protocol. The first items you will find in the file are a series of conditional definitions. Make sure that you find a grouping that looks like this…

```
#ifndef IRLIB_PROTOCOL_12_H
   #define IR_SEND_PROTOCOL_12
   #define IR_DECODE_PROTOCOL_12
   #define PV_IR_DECODE_PROTOCOL_12
   #define PV_IR_SEND_PROTOCOL_12
#endif
```

We have already added support for protocol 12 but if you are adding additional protocols such as 13, 14 etc. you will need to cut and paste one of the sections and edit it to add the next number of protocol. This ensures that if you did not include the number 12 protocol in your stack of supported protocols, that all of the macros used by IRLibCombo.h are defined as empty.

There are four additional sections that we need to test. Below is a shortened version of the first two of those sections of code.

```
class IRdecode:
   PV_IR_DECODE_PROTOCOL_01
   PV_IR_DECODE_PROTOCOL_02
… Omitted some code here for clarity…
   PV_IR_DECODE_PROTOCOL_11
   PV_IR_DECODE_PROTOCOL_12
   PV_IR_DECODE_HASH //Must be last one.
                     //Add additional above this as needed.
{
public:
   bool decode(void) {
         IR_DECODE_PROTOCOL_01
         IR_DECODE_PROTOCOL_02
… Omitted some code here for clarity…
         IR_DECODE_PROTOCOL_11
         IR_DECODE_PROTOCOL_12
         IR_DECODE_HASH    //Must be last one.
                           //Add additional above this as needed.
         return false;
   }
};
```

The listing above shows the definition for our master decode class called IRdecode. Each protocol header file defines PV_IR_DECODE_PROTOCOL_xx and another macro IR_DECODE_PROTOCOL_xx where xx is a two digit protocol number. We have already added protocol 12 but if you are adding additional protocols 13, 14 etc. you must add another entry in each of these sections. Make sure that the _HASH is always the last one in the list. You must maintain numerical order as well.

Further down we have another similar section for the sending class IRsend.

```
class IRsend:
```

```
   PV_IR_SEND_PROTOCOL_01
   PV_IR_SEND_PROTOCOL_02
 … Omitted some code for clarity…
   PV_IR_SEND_PROTOCOL_11
   PV_IR_SEND_PROTOCOL_12
   PV_IR_SEND_RAW    //Must be last one.
                     //Add additional above this as needed.
{
public:
   void send(uint8_t protocolNum, uint32_t data,
               uint16_t data2=0, uint8_t khz=38) {
    if(khz==0)khz=38;
      switch(protocolNum) {
        IR_SEND_PROTOCOL_01
        IR_SEND_PROTOCOL_02
… Omitted code here for clarity…
        IR_SEND_PROTOCOL_11
        IR_SEND_PROTOCOL_12
        IR_SEND_RAW  //Must be last one.
                     //Add additional above this as needed.
      }
   }
};
```

As before we have already added the proper lines for protocol 12 but if you add additional protocols 13 and upwards you must edit in new lines. Be sure that the _RAW entry is always the last one.

These four macros were all defined in our protocol file. Look at the top of IRLib_P12_Tutorial.h and you will find the following section.

```
#define IRLIB_PROTOCOL_12_H
#define IR_SEND_PROTOCOL_12 case 12: IRsendTutorial::send(data); break;
#define IR_DECODE_PROTOCOL_12 if(IRdecodeTutorial::decode()) return true;
#ifdef IRLIB_HAVE_COMBO
      #define PV_IR_DECODE_PROTOCOL_12 ,public virtual IRdecodeTutorial
      #define PV_IR_SEND_PROTOCOL_12   ,public virtual IRsendTutorial
#else
      #define PV_IR_DECODE_PROTOCOL_12  public virtual IRdecodeTutorial
      #define PV_IR_SEND_PROTOCOL_12    public virtual IRsendTutorial
#endif
```

The first definition informs IRLibCombo.h that we have included the header file for protocol 12 and so it should be among those included in the master decode and send classes. If IRLibCombo.h does not see this, it will provide empty macros for all four of these.

For a complete explanation of how these macros are used by IRLibCombo.h to assemble our sending and decoding classes, please see Appendix A.3.2 Creating Protocol Combo Classes. We won't bother to reproduce that discussion here.

In the next section we will show you some examples of how to create sending and decoding routines from scratch when a protocol cannot use the generic sending and decoding routines to do most of the heavy lifting for us.

### 3.2.4 Implementing Protocols without the Generic Routines

The generic routines work well when a protocol is well behaved with a typical header followed by some bits that are encoded using variable length spaces. We were also able to implement the sendGeneric(…) method so that it could also use variable length marks. At one point the decodeGeneric(…) function would also handle variable length marks but it made the code very complicated and in our experience Sony is the only protocol to use variable length marks so we decided to simplify the generic routine and implement our own Sony routine.

The RC5 and RC6 protocols which use phase encoding have their own support functions that help decode and encode those protocols. To be honest if you came across another phase encoded protocol, I think it would be very difficult to reverse engineer from the timing data alone. Without some other documentation perhaps in IRP notation I'm not sure how you would go about figuring that out.

We invite you to go through the various protocol files and see how we have implemented other protocols. For example Samsung36 breaks the data up into various chunks with odd bits of marks and spaces thrown in the middle. Some protocols change the header depending on whether it is a start frame or repeat frame but we are generally still able to make some use of the generic routines.

The DirecTV and RCMM protocols use double bit encoding. They use some combination of a mark and a space to encode 2 bits at a time. While we are unaware of other protocols that use this type of encoding, looking at these protocols is a little more detail will give you a better feel for how to implement anything else you come across that cannot use the generic routines. You've already seen some of the techniques we will use when we implemented the ditto on our Tutorial protocol which was really GICable.

Below we will look at how the generic routines were created and also take a peek at the DirecTV and RCMM implementations.

#### 3.2.4.1 Implementing a Specialized Sending Protocol

First let's look at the generic routine. In IRLibProtocols/IRLibSendBase.cpp you will find the following code…

```
void IRsendBase::sendGeneric(uint32_t data, uint8_t numBits,
     uint16_t headMark, uint16_t headSpace, uint16_t markOne,
     uint16_t markZero, uint16_t spaceOne, uint16_t spaceZero,
     bool useStop, uint32_t maxExtent) {
  extent=0;
  data = data << (32 - numBits);
  enableIROut(kHz);
//Some protocols do not send a header when sending repeat codes. So
we pass a zero value to indicate skipping this.
  if(headMark) mark(headMark);
  if(headSpace) space(headSpace);
  for (uint8_t i = 0; i <numBits; i++) {
```

```
    if (data & TOPBIT) {
      mark(markOne);  space(spaceOne);
    }
    else {
      mark(markZero);  space(spaceZero);
    }
    data <<= 1;
  }
  if(useStop) mark(markOne);   //stop bit of "1"
  if(maxExtent) {
#ifdef IRLIB_TRACE
    Serial.print("maxExtent="); Serial.println(maxExtent);
   Serial.print("extent="); Serial.println(extent);
   Serial.print("Difference="); Serial.println(maxExtent-extent);
#endif
   space(maxExtent-extent);
   }
   else space(spaceOne);
};
```

Some protocols require that we keep track of how long it takes to send the data. We then need to pad out the overall length of a frame by sending a long space at the end. We know the overall extent that a frame is supposed to last. So we compute our sending time and then subtract that from the target length of time and that tells us how much blank space. Not all protocols require this calculation. We accumulate our timing data by visualizing the variable "extent" to zero.

We want to send the highest order bits first. If we are using a 32-bit protocol then the highest order bit is already in our variable unsigned 32 bit value "data". However suppose we are using 22 bits. We do not want to send 10 bits of zeros before our actual data. If we do a binary left shift of the data by 10 bits then our highest order data bit is in bit number 32. Because our generic routine needs to handle any number of bits we simply do…

```
  data = data << (32 - numBits);
```

Next we need to initialize the transmitter by passing it the modulation frequency using the base method enableIROut(uint8_t). We pass the integer eight bit frequency in kilohertz as a parameter. Now we want to send the header mark and space. Our generic routine has to handle protocols that do not use headers. If you pass zero for those parameters it skips the header. When implementing your routine you already know whether or not you need a header and you can simply call the mark and space functions with the proper values like this.

```
  mark(headMark);
  space(headSpace);
```

The mark(…) method turns on the modulation of the IR LED at the proper frequency and then delays the number of microseconds that you specify. The space(…) turns off the IR LED and then delays the specified number of microseconds.

Next we loop through the bits one by one. We do a logical AND between the data and the value TOPBIT. That value has been defined in IRLibProtocols.h as

```
#define TOPBIT 0x80000000
```

This value is a 32 bit value with the highest order bit turned on. We use it to mask off that highest order bit and decide whether it is a one or a zero. We then call mark(…) and space(…) with the appropriate values. We then shift the data left one more bit so that the next pass through the loop we have the next bit in the highest order slot.

Finally we send a closing mark if necessary followed by at least some amount of space. If you don't know the proper amount of lead out for your particular protocol you still need to send something in the way of a space because that turns off the transmitter. As you can see we use the spaceOne value as a default. Although you don't see it in action, the mark(…) and space(…) functions also add their timing values to the variable "extent" so that at the end of the transmission we know the total time to transmit the signal.

Now let's take a look at the DirecTV sending routine. If you're not familiar with this protocol, you should first read the documentation section that describes how it encodes two bits at a time. Basically it uses variable length marks and spaces. One bid is encoded by a short or long mark and the next bit is encoded by a short or long space.

```
void IRsendDirecTV::send(uint32_t data, bool first, uint8_t khz) {
  enableIROut(khz);
  if(first) mark(6000); else mark(3000);
  space(1200);//Send header
  for (uint8_t i = 0; i < 8; i++) {
    if (data & 0x8000) mark(1200); else mark(600);
    data <<= 1;
    if (data & 0x8000) space(1200); else space(600);
    data <<= 1;
  };
  mark(600);
  space(longLeadOut?50*600:15*600);
};
```

This particular protocol has different varieties which use different modulation frequencies so we have accommodated that capability. It also uses a different length header mark to determine whether it is a first frame or a repeat frame. The header space is always constant. Because we know our number of bits is constant, we did not bother to pre-shift the data to get it all the way left in a 32 bit variable. We simply mask off the 16th bit because we know it is a 16-bit protocol. Depending on if that data is a zero or one we send a short or long mark. Then we shift over one bit to get to the next one and send the variable length space and shift again. Even though this is a 16 bit protocol we go through the loop just eight times because we are dealing with bits two at a time. We conclude with a mark followed by a lead out space. Different varieties of this protocol use different length of lead out space and we accommodate that.

We also invite you to look at the send routine for the RCMM protocol which also uses double bit encoding. For each bit it uses a fixed length mark followed by variable length space that can be 4 different values. The shortest indicates 00. The next longer one indicates 01 and the next 10. Finally the longest space indicates 11. We have to accommodate variable number of bits because there are multiple versions of this protocol. Note that we have to mask off the bits 2 at a time. You should be able to look at the code and see how we implemented it.

### 3.2.4.2 Implementing a Specialized Decoding Protocol

As before we will start with looking at the internals of the generic code and then move on to some specific examples namely DirecTV and RCMM. The listing below is a somewhat stripped-down or simplified version of the generic code that you can find it IRLib2/IRLibDecodeBase.cpp

```
bool IRdecodeBase::decodeGeneric(uint8_t expectedLength,
      uint16_t headMark, uint16_t headSpace,  uint16_t markData,
      uint16_t spaceOne, uint16_t spaceZero) {
  resetDecoder();
  uint64_t data = 0;
  bufIndex_t Max=recvGlobal.decodeLength-1;
  if (recvGlobal.decodeLength != expectedLength)
     return RAW_COUNT_ERROR;
  if(!ignoreHeader) {
    if (!MATCH(recvGlobal.decodeBuffer[1],headMark))
       return HEADER_MARK_ERROR(headMark);
  }
  if (!MATCH(recvGlobal.decodeBuffer[2],headSpace))
     return HEADER_SPACE_ERROR(headSpace);
  offset=3;//skip initial gap plus two header items
  while (offset < Max) {
    if (!MATCH (recvGlobal.decodeBuffer[offset],markData))
       return DATA_MARK_ERROR(markData);
    offset++;
    if (MATCH(recvGlobal.decodeBuffer[offset],spaceOne)) {
      data = (data << 1) | 1;
    }
    else if (MATCH (recvGlobal.decodeBuffer[offset],spaceZero)) {
      data <<= 1;
    }
    else return DATA_SPACE_ERROR(spaceZero);
    offset++;
  }
  bits = (offset - 1) / 2 -1;//didn't encode stop bit
  // Success
  value = (uint32_t)data;           //low order 32 bits
  address = (uint16_t) (data>>32);  //high order 16 bits
  return true;
}
```

The very first thing you must do is call the resetDecoder() base method. This initializes a number of important values and make sure that everything is ready to be decoded. If you do not use the decodeGeneric(…) function, be sure to make this the first thing that you do.

The generic routine allows for the possibility that we do not want to measure the number of slots that we are going to use or that there may or may not be header marks and spaces. We have stripped those conditions out of the listing above just to simplify our discussion.

The philosophy of decoding reminds me of the old joke that asks "How do you carve a statue of an elephant? You start with a really big rock and chip away everything that does not look like an elephant." Our job in decoding is to throw out everything that doesn't look like our protocol. We are basically looking for reasons to reject this data and failing to do so means that we will accept it.

The quickest way to reject a set of data is if it is the wrong length. While we cannot yet know if this data represents the proper number of bits, we do know whether or not it uses the proper number of slots in our decode buffer. Most of the time this value follows the rule (bits*2)+4 however in the case of the double bit protocols this isn't true. For most protocols we will know however how many entries in the buffer we should have so we test that first. Note that if it doesn't match we say "return RAW_COUNT_ERROR". That particular macro is normally defined simply as "false" which means that we failed to decode this particular protocol.

There are several of these error macros that you will see us use throughout the decoding process. If you turn on the debugging features these macros not only supply us with a return value of "false", they also print an error message on the serial monitor. All of these macros are defined near the end of the file IRLib2/IRLibDecodeBase.h.

We want to check the length of the header mark. But we would not do so if the user has set the "ignoreHeader" option. See the documentation for that value in the discussion of the base decode class. Next we check the length of the header space. Each of these has their own custom error macros.

We are now ready to look at the data bits beginning with the third entry in our buffer. We go through them all until we get to the second to last one. That is because the last slot in the buffer is for our stop bit which is not encoded.

This particular decoder uses fixed length marks for every bit. We test the item of the buffer with our expected length of the mark using the MATCH(v,e) macro that we described earlier when we were decoding the ditto of the Tutorial protocol. Notice that we pass a parameter to the error macro. The parameter is the expected value. This value is only used for the debug output. In this instance if the length of a mark was wrong your debug output would say something like…

```
 Protocol failed because data mark wrong.
Error occurred with decodeBuffer[5]= 403 expected: 550
```

Once we have determined that the mark is of legal length we then compare the next slot which is a space relative to a long space for a data bit one. If that is legal, we shift the data left one bit and then do a logical AND with a "1". If it is not legal then we need to check if it is a short space to see if it is a zero. If it matches, we simply do the left shift and leave the zero lowest bit as zero. If both long and short space measurements fail then we return with an error message and the data stream is rejected as not being this protocol. We continue in the while loop until we have exhausted all of the data.

While writing this documentation we realized we did not measure the length of the final mark. It is highly unlikely that everything else would've matched and the closing mark would be the wrong duration. We've decided to skip this step.

For this generic decoder we do not know the number of bits so we compute it by the formula shown. For some decoders you can set the number of bits directly. Our generic decoder actually computes its data in a 64-bit integer because some of our protocols use more than 32 bits. We split the data into a 32-bit value and another 16-bit address. Technically the code could handle a 64-bit protocol but we have not found any so we limit it to 48 bits and saved a tiny amount of RAM. We conclude by returning "true" indicating that this is the protocol we were looking for. The function that calls genericDecode(…) has to set the value protocolNum.

We now invite you to take a look at the decode function for DirecTV. We will not reproduce the code here. It basically uses the same principles that we outlined in the generic routine. In this instance we know that the decodeLength should be 20 because there are no variation of number of bits in this protocol.

The DirecTV protocol uses the length of the header mark to determine whether this is an additional frame or a repeat frame. We report this in the "address" variable. Normally that variable holds the upper 16 bits of a value that is greater than the normal 32 but we have repurposed it as a flag to indicate this is a repeat frame.

Recall that this protocol encodes a bit for the marks and another bit for the space. A long mark indicates one and a short mark indicates zero. This is followed by a long space or a short space indicating a one or a zero. At the end we report that this is 16 bits, we copy the data and set the protocolNum and return true.

We will take a more detailed look at the RCMM decoder. There are multiple possibilities for the number of bits for this protocol so we need a rather long compound if statement to check the decodeLength. The length of the header mark and space are tested as always and then we begin looping through the data bits. The protocol uses a fixed length mark but a variable length space that can have 4 different lengths which encode two bits at a time.

This creates a problem because the legal values are 278, 444, 611, 778 which are fairly close together. Normally our MATCH(v,e) macro uses a percentage tolerance of 25%. But if you take 25% of 778 you get 194.5 and subtract that from 778 you get 583.5 which is lower than the third value. So if you had a value of 611 which should've encoded to "10" it would be a legal value within 25% of 778. Instead of using a percentage plus or minus tolerance we need to use an absolute tolerance. We have chosen to use 80 µs as a tolerance and that seems to work well. Note that depending on your particular hardware you made to also adjust the myDecoder.markExcess value.

Most of the time our traditional percentage match will work fine. You only need to distinguish between a short and a long value and typically they vary by 200% or more. But if you come across another protocol where you are trying to finally distinguish the lengths of relatively large values as we do in this protocol, then you might consider using the ABS_MATCH(v,e,t) where "v" is the value, "e" is the expected value and and "t" is the tolerance. The MATCH macros and its varieties are all defined in IRLibDecodeBase.h.

Our traditional MATCH(v,e) actually redirects to PERC_MATCH(v,e) that does the actual calculation. You can change the default behavior of the entire library by defining a value

```
#define IRLIB_USE_PERCENT
```

This will redefine MATCH(v,e) to use ABS_MATCH(a,b,t) with a default tolerance defined by DEFAULT_ABS_TOLERANCE.

#### 3.2.4.3 Testing and Debugging

The ultimate test of whether or not you have succeeded in implementing the protocol is to see if you're sending routine will actually control the device you want to control and to see if your decoding routine produces strings of bits that seem reasonable. If you have 2 Arduino devices available we suggest also that you set up one is a transmitter and one as a receiver. Put a sending routine in one and a decoding routine in the other and see if you can decode your own signals. This is also a good way to calibrate your myDecoder.markExcess value for your particular hardware and conditions. For sending we recommend that you modify the sample program in IRLib2/examples/patterns/ to add your new protocol. If there are variations to the protocol or if you want to test various bit patterns you can add those to the program in the same way we implemented variations of the standard protocols.

If for some reason your decoder is rejecting what you think should be a legal signal, you can turn on the trace debugging system. This will enable all of those ERROR_whatever macros that we put in the decoder to give you an idea of why the code was rejected. What value did it see and what value did you expect to see. You enable that system by putting the following define at the top of your decoding sketch…

```
#define IRLIB_TRACE
```

You will then get a message at the beginning of each attempted decoding telling you which protocol is working and why that protocol was rejected. Of course you can also always put other Serial.println("message") debugging statements in your code wherever you feel it would be useful.

## 3.3 Final Thoughts

Of course the problem with understanding an unknown protocol is that you don't know what you don't know. We've tried to give you an overview of everything that we have encountered in implementing the protocols so far. We have been reluctant to add additional protocols based solely upon IRP notation because we do not have any consumer electronic devices available to us that actually use those protocols. If you have access to a device and believe that you know the specifications or even have an IRP definition, please contact us and we will be happy to help you implement it if you will help us by testing the final product. If we have a user who can verify that what we end up producing really works on actual equipment then we will be happy to support that protocol in future releases. The restructuring of the code for IRLib2 will make it much easier to add new protocols without major disruption to the rest of the code.

Additionally if you are building a device of your own such as a robot or some other specialized application and you are not trying to control some existing device, you might want to consider making up your own private protocol for your purposes only. You don't want to have a situation where you tell your robot to turn around and it ends up changing the channel on your TV. You have seen how the existing protocols work. Make something up that you are sure is so different that it will not interfere with any of your existing consumer devices.

Make your protocol easy to implement using the generic send and decode functions. Namely give it a reasonably long header mark and space followed by data bits encoded with variable length spaces. Choose a long space that is two or three times as long as the short space. The key to making sure that you don't interfere with other protocols will probably be choosing the number of bits. Make it something really strange like 17 or 19 that is unlikely to be used by some other device. They like to use bit lengths that are multiples of four as you have seen numbers like 16, 20, 24, 32, and 48. Always use 38 kHz modulation frequency. That way you are sure your receiver device will accurately detect it. Using a strange frequency will not necessarily guarantee that some other device would not receive it. As we have seen the standard 38 kHz receivers can easily detect modulation from 36-40 and some up to 57 or 58 kHz. The bandpass filter used by these devices is generally reasonably broad so frequency is not a very good way to distinguish yourself.

Once you have designed your own protocol and implemented sending and decoding routines you have a choice of either building a custom Arduino based transmitter or using a universal learning remote to learn signals from your Arduino.

That's all the advice we have for now. We will be happy to help you if we can whenever we have the free time to do so.

# Appendix A – Why and How We Made IRLib2 So Different

From its initial release in January 2013 up through 2016, we have discovered a number of issues with our original design for IRLib1 that we wanted to try to address in a future version. The difficulties which had arisen were significant enough that we felt that the only way to address all of the issues was to do a major top-down rewrite and reorganization of the software. We soon discovered that many of the changes we wanted to make, along with new features such as auto resume, were so significant that it was going to be difficult if not impossible to make the code backwards compatible. Once we had realized backwards compatibility was not going to be possible, we also decided that the naming conventions and functions could use an overhaul. Many of the variable names were confusing and we were using inconsistent style in the naming of variables and functions. We will now go through an explanation of the issues we hoped to correct in this major renovation.

The problems fall into four major categories

- Hardware interrupt and timer conflicts
- Wasteful use of limited RAM and flash program memory
- Complex procedures for including only needed protocols
- Variable names which are difficult to understand

We believe we have come up with a reasonable solution to all of these issues in the reorganization and rewrite of the code. In the following sections we will address each of these issues and in the last section will explain why backwards compatibility could not be maintained in the face of very useful new features.

## A.1 Hardware Interrupt and Timer Issues

Arduino compatible microcontrollers have extremely limited resources. Hardware timers, dedicated PWM output pins, and pin change interrupt dedicated pins are a premium. For example on the Arduino Leonardo the default timer for the modulating frequencies and the IRrecv 50 µs interrupts is TIMER1 however that is also used by the standard Arduino Servo library. The Arduino Tone library also can conflict with interrupts and timers used by IRLib.

Interrupt Service Routines or ISRs are global functions in the library. Normally if you have a function in a library that is not used, the linking loader will eliminate it from your uploaded object code. However the nature of ISRs is such that the loader cannot know for sure whether or not you actually make use of the ISR. Therefore it must always include it even if you are not making use of that particular feature of the library. An application which made its own use of timer driven interrupts will not properly link with IRLib1 even if you do not create an instance of the IRrecv objects. Similarly the IRfrequency and IRrecvPCI classes make calls to the built-in Arduino function attachInterrupt() which can cause problems. Any function containing a call to that function can cause a conflict with other libraries even if you never create an instance of one of those classes.

If your application only uses the IRrecvLoop receiver class and/or only does sending and no receiving, there would still be conflicts with these other interrupt driven classes. Our solution under IRLib1 was to create flags in IRLib.h which would trigger conditional compilation to temporarily remove these unused classes from the library. The problem with that system is that the flag that you comment or un-comment had to be in IRLib.h and could not be in your own application. That is because all of the code in a library is precompiled separately from the code

in your sketch. The IDE will only recompile it if you change platforms such as from the Arduino Uno to Arduino Leonardo. It will also recompile if you have made any edits to any of the library files.

The problem with changing such a flag within the library itself is that if you change the flag and forget that you changed it, later when compiling a different application you have to remember to change the flag again. There is no way to get this conditional compilation to be controlled solely by the application code itself. Except for one way… make it all separate libraries. So that is what we have done with IRLib2.

The IRrecv, IRrecvPCI, and IRfrequency classes are now each in their own folders and essentially are libraries unto themselves. If you do not include their individual header files, they will not be included in your final sketch. Because the IRrecvLoop does not make use of hardware interrupts or timers, it can remain in the IRLib2 folder along with much of the remaining code without causing any conflicts with other libraries. The base receiving code is also in the IRLib2 folder where it can be accessed by the other receiver classes but in itself does not cause any conflicts with other libraries.

Rather than call these library folders IRrecv, IRrecvPCI, and IRfrequency we decided to highlight the fact that they were still part of the entire package named IRLib therefore we named them IRLibRecv, IRLibRecvPCI and IRLibFreq respectively.

Note that initially we tried to create header files and code files for these individual receiver classes but include them all in the same IRLib2 folder. But even if you have these classes implemented in separate header files and cpp files, the IDE still considers everything within a particular folder to be part of a single library. To ensure that there were no conflicts with unused portions of the code, each of these receiving classes had to be in its own separate library folder.

## A.2 Limited Memory Resources

Arduino-like platforms are very limited devices. RAM memory is at an extreme premium and even program flash memory can be quite limited depending upon your application. For example I have a device that is an IR remote, connects to my iPhone by Bluetooth Low Energy, and has a small 128 x 64 pixel OLED display. It will barely compile on an ATmega32u4 processor for lack of program memory.

Even though we've solved the problem of interrupt and timer conflicts with unused portions of the receiver code, there is additional overhead associated with any receiver class that is not necessary if your application only does sending. That overhead is the collection of global variables and the buffer of timing data that is used by receiving and decoding classes. The default size of this buffer is 100 16-bit words of RAM memory which as we have mentioned is at a premium on these platforms. Normally all of the data associated with an object is stored within the object class itself. However these variables including the large buffer have to be declared global because an ISR cannot be passed a parameter.

We could have created a single global pointer to a structure and if the structure was not needed in a send only application it would waste only four bytes. However the indirect reference to the data slows things down and requires additional code because you are constantly dereferencing a pointer for each access. We want the ISR to operate as quickly as possible and the dereferencing throughout the rest of the code will consume valuable code space.

Therefore we decided to separate out the sending portion of the library into a separate folder and standalone library so that an application which only sends does not receive and decode can eliminate the overhead of this large buffer.

We could have created a library folder called IRsend but for reasons we will explain below it seemed more appropriate to call it IRLibProtocols. The base class for sending as well as the code for individual protocols is available in this library folder and can be included without causing any of the receiving or decoding code or variables to be linked.

## A.3 Simplifying Protocol Expansion and Inclusion

The next major issue is the handling of an increasing number of protocols. The original IRremote library upon which this code has been based included the first four protocols that we support. The initial release of IRLib1 expanded this to seven protocols. Additional protocols were provided as standalone example sketches that showed how to create a hybrid class that included the original seven protocols plus whatever additional protocol you wanted. This was an interesting educational exercise if you wanted to study how to extend a library using object-oriented coding techniques in C++. However as a practical way of managing protocols it was quite cumbersome.

The original libraries IRremote and IRLib1 also have a combo class for decoding and sending signals of any of the four or seven supported protocols. While it was possible to code your own combo class which included only the protocols you wish to use, again it required something beyond beginner's level coding skills. We needed a way to allow the end-user to create an application which used only the protocols that their particular sketch needed to use. Otherwise there was a great deal of unnecessary code in your application that would never get used.

### A.3.l Including Only Necessary Protocols

The first issue is how to eliminate the protocols that you do not actually use. One solution which has been adopted by the most recent versions of IRremote is to use a series of conditional compile flags in the header file that allow you to turn off the protocols you don't want to use. That presents the same problem that we had with conditional compile flags turning off or on the use of the receiver and decoder classes. You cannot set the flag in your application sketch. It had to be edited in the header file of the library itself. We wanted to create a way to simply make use of "#include <whatever.h>" statements within the application program and that would determine which protocols would be included in your master combo sending or decoding classes.

We also wanted to make it easier to add additional protocols to the library. It would be nice if the code for each protocol was contained in a single file that could be easily added to the package without modifying major portions of the code itself.

One of the problems is that each protocol includes both a decoder and a sender portion. Any application which includes the decoder portion of a protocol must necessarily include the decoder base code and must include access to the global receiver variables including that big buffer for receiving and decoding. It would be extremely confusing to have each protocol be a standalone library folder it even more confusing for each protocol to have a separate folder or separate files for decoding versus sending.

Normally we think of a library as having two parts: a header file such as library.h which contains prototypes for all of the classes and functions as well as definition of global variables and separately a C++ code file such as library.cpp which contains the actual code that implements the functions described in the header file. When you do "#include <library.h>" in your application program it virtually cuts and pastes that actual file into your program. It in effect says "Here are some functions and variables that are not in this compilation but which you will find precompiled out there somewhere. This is all the information that you need to know so that you can compile your program in a way that will link to the precompiled code." However the library.cpp file does not get included in your sketch and does not get recompiled every time you recompile your program.

But it is only by convention that prototypes and external references appear in a library.h file and that it contains no actual implementation code. There is nothing prohibiting you from putting actual C++ code in library.h. I realized this one day when I came across an Arduino compatible library which contained only a filename.h and no filename.cpp at all. By putting all of your actual implementation into a header file you are cut and pasting **everything** into your application and it gets recompiled every time you compile your sketch.

For applications on PCs or phones or tablets or just about anything other than a small microcontroller, this is a really bad idea. Applications for such devices can contain hundreds of thousands or even millions of lines of code. It takes a very long time to recompile the entire program and its libraries each time. Also except in the world of microcontrollers, there exists such things as dynamic linked libraries or DLLs. These libraries get accessed by an application on-the-fly and are only loaded into memory as needed. So for normal computer programming purposes, it would be crazy to put all of your code into a header file. But for our application on an Arduino-like microcontroller, we are only talking about a few dozen lines of code per protocol. The flexibility that we obtain by this "compile everything every time" process is well worth a couple of extra seconds of compile time. It makes our code much easier to understand and our applications easier to write.

Here's how it works…

The file IRLib2/IRLibDecodeBase.h contains a line near the beginning

```
#define IRLIBDECODEBASE_H
```

Mostly it ensures that we do not recompile the header file twice if it somehow accidentally gets included more than once. But it also lets us know for certain that the application wants to do decoding.

Each protocol is implemented in its own header file such as IRLib_P01_NEC.h. It contains both the decoding and the sending code for the NEC protocol. The decoding code is wrapped in conditional compilation directive as follows…

```
#ifdef IRLIBDECODEBASE_H
class IRdecodeNEC: public virtual IRdecodeBase {
  /*
   * decoding code goes here
   */
};
```

```
#endif //IRLIBDECODEBASE_H
```

So if you do not do an "#include <IRLibDecodeBase.h>" prior to including the protocol file, the decoding portion of the protocol code does not get compiled. Similarly in the file IRLibProtocols/IRLibSendBase.h there is a line

```
#define IRLIBSENDBASE_H
```

and in the NEC protocol file there is a segment of conditional code such as…

```
#ifdef IRLIBSENDBASE_H

class IRsendNEC: public virtual IRsendBase {
  /*
   * sending code goes here
   */
};
#endif //IRLIBSENDBASE_H
```

and similarly if you do not include the send base header file prior to the protocol file then the sending portion of the protocol code does not get included.

So now whether you want to do sending only or decoding only or do both, you can easily control in your application program exactly which protocols you want to use. The only limitation is that if your sketch decodes one protocol and sends another one you would have to make some accommodations. We may create a sample sketch eventually which would handle this rare circumstance.

### A.3.2 Creating Protocol Combo Classes

So we now have a way to include into our application only the protocols we want and only the sending or decoding portions of those protocols which we want. That still leaves us with the task of combining the individual protocols into a single decoding class which will handle multiple protocols or a single sending class which we can send any data by passing the protocol number as a parameter. First let's look at what we're trying to create if we did it by hand.

Suppose we wanted a decoder which would handle NEC, Sony, and JVC protocols but no others. We would define it as follows:

```
class IRdecode:
   public virtual IRdecodeNEC,
   public virtual IRdecodeSony,
   public virtual IRdecodeJVC
{
   public:
     bool decode(void) {
       if (IRdecodeNEC::decode()) return true;
       if (IRdecodeSony::decode()) return true;
       if (IRdecodeJVC::decode()) return true;
     )
}
```

While that is not an extremely complicated piece of code, it would be nice if somehow we could create some sort of series of macros which would assemble it for us. The handwritten code for a master sending routine using the same three protocols would look something like this:

```
class IRsend:
   public virtual IRsendNEC,
   public virtual IRsendSony,
   public virtual IRsendRC5,
   public virtual IRsendJVC
{
   public:
     void send(uint8_t protocolNum, uint32_t data, uint16_t data2) {
        switch(protocolNum) {
          case NEC:  IRsendNEC::send(data,data2); break;
          case SONY: IRsendSony::send(data,data2); break;
          case JVC:  IRsendJVC::send(data,(bool)data2); break;
        }
     }
};
```

This is a little more complicated. The NEC protocol needs 2 parameters the second of which is the frequency change to 40 kHz if you want to use NEC code as Pioneer protocol. However the second parameter of Sony is the number of bits and the second parameter for JVC is a flag telling whether or not this is an additional frame or a repeat frame. It is further complicated because some protocols require yet another parameter. The opportunity to get confused over what parameters mean for different protocols is pretty high. It would be nice if the library could handle all these details for us. We have created such a system in the file IRLibProtocols/IRLibCombo.h. Here's how it works.

Looking at these two samples of code, we can see that there are 4 places per protocol that have a special line of code. We need to define 4 macros to supply that line of code. For the decoder we need to define a macro which supplies lines that look like this…

```
   public virtual IRdecodeNEC,
```

and another macro supplying lines that look like this…

```
   if (IRdecodeNEC::decode()) return true;
```

In the protocol header file we do the following defines…

```
#define PV_IR_DECODE_PROTOCOL_01  public virtual IRdecodeNEC
#define IR_DECODE_PROTOCOL_01 if(IRdecodeNEC::decode()) return true;
```

There are similar defines in each of the protocol header files as well as a pair of macros used to create the combo sending class. In IRLibCombo.h there is a section of code that looks like this…

```
#ifdef IRLIBDECODEBASE_H
class IRdecode:
```

```
   PV_IR_DECODE_PROTOCOL_01
   PV_IR_DECODE_PROTOCOL_02
   PV_IR_DECODE_PROTOCOL_03
   PV_IR_DECODE_PROTOCOL_04
…  /*Additional omitted here to shorten the example*/
{
public:
   bool decode(void) {
         IR_DECODE_PROTOCOL_01
         IR_DECODE_PROTOCOL_02
         IR_DECODE_PROTOCOL_03
         IR_DECODE_PROTOCOL_04
…  /*Additional omitted here to shorten the example*/
         return false;
   }
};
#endif //IRLIBDECODEBASE_H
```

At the beginning of IRLibCombo.h there is code which ensures that all of these macros actually exist so that the code will compile. There are statements such as…

```
#ifndef IRLIB_PROTOCOL_01_H
   #define IR_DECODE_PROTOCOL_01
#endif
```

This ensures that if we did not include protocol 1 which is NEC, that all of its macros are defined as empty. That works really well for the "if (decode()) return true;" statements. However for the "virtual public IRdecodeNEC" statement at the beginning of the class definition we have a problem. The list of such statements have to be separated by commas. If we put the commas in IRLibCombo.h but some of the macros are defined as empty then we get 2 consecutive commas which is illegal. If we put a comma on each macro there is a problem because the last one in the list must not have a comma.

Our solution was to put the comma before the "virtual public IRdecodeNEC" statement on all but the first. We cannot determine if we are the last of the included protocols but we can determine if we are or are not the very first. At the bottom of each protocol header file is the define

```
#define IRLIB_HAVE_COMBO
```

This serves as a notice to all subsequently included protocols that there is already at least one protocol in the list ahead of you. Therefore when you define your macros you must put a comma in front of your list macros. And if this particular flag is not defined, then you are the first of the included protocols and you should not put the comma in front of your defines. That code looks like this…

```
#ifdef IRLIB_HAVE_COMBO
   #define PV_IR_DECODE_PROTOCOL_01 ,public virtual IRdecodeNEC
   #define PV_IR_SEND_PROTOCOL_01   ,public virtual IRsendNEC
#else
   #define PV_IR_DECODE_PROTOCOL_01  public virtual IRdecodeNEC
```

```
   #define PV_IR_SEND_PROTOCOL_01    public virtual IRsendNEC
#endif
```

This entire system of flags and macros is actually slightly more complicated then we have described it here. But despite its complexity, it is well documented and relatively simple to add additional protocols to the system. The end result is that from an end user's perspective it is now extremely easy to create a combo decoding or combo sending class using nothing more than a particular sequence of include statements. As described elsewhere in the document and the sample stretches, you begin with including the base sending and/or decoding header files followed by the individual protocols and conclude with including IRLibCombo.h. So our earlier example looks like this…

```
#include <IRLibDecodeBase.h>
#include <IRLibSendBase.h>
#include <IRLib_P01_NEC.h>
#include <IRLib_P02_Sony.h>
#include <IRLib_P06_JVC.h>
#include <IRLibCombo.h>
IRsend mySender;
IRdecode myDecoder;
```

Putting together this stack of includes is much, much simpler than writing out code for the combo classes on a case-by-case basis. The only restriction is that the lowest numbered protocol among those you are using must be the first one included. Although subsequent protocols need not be in order, we recommend you do so. In the above example we could've included JVC prior to Sony however if we later decided to remove NEC then the code would not compile properly until you put Sony ahead of JVC.

## A.4 Why We Broke Backwards Compatibility

If you want to blame someone for why IRLib2 is not completely backwards compatible with IRLib1 you can blame it on programmer Gabriel Staples. (That's a joke. It's not his fault… It's entirely mine.) Gabriel is a talented programmer who invested many days effort significantly improving IRLib1 by branching the code on GitHub, fixing bugs, and inventing the major new feature of IRLib2 that is not related to the reorganization of the code. He invented the auto resume feature. He also called my attention to the need to ensure atomic access to multi-byte variables accessed both within and without ISRs. This is an issue I had not previously considered. He spent days working on it getting it to work properly for no personal reward. We are extremely grateful for all of his hard work and he deserves significant credit for his contributions.

I'm ashamed to report I don't believe I used a single line of his code. Without going into a lot of details, there were a couple of features he included for which I disagreed. But the main reason that I rewrote his code from scratch was because I need to thoroughly understand how it works in order to fully document it and maintain it. Although the auto resume feature which I implemented probably does not contain any of his code, it could never have been written without the work he invested in the project. His code served as my guide in my rewrite. I am not so egotistical as to refuse to put someone else's code in my project. I just needed to more deeply understand how it worked and that process ended up with me rewriting it myself.

His code would've maintained backward compatibility with IRLib1 because it was in fact simply patches to that code. But while working on the project I began to conceptualize some things differently. For example we have always needed for decoders and receivers to communicate with one another. In the past, we would pass a pointer to your decoder object in your call to myReceiver.GetResults(&myDecoder); However under auto resume, the receiver needs access to the decoder buffer possibly prior to any call to GetResults. It made more sense to me to have both the receiver buffer and the decoder buffer as global variables that could be accessed by either the receiver or the decoder. Therefore it was no longer necessary to pass a pointer to the decoder in GetResults. That was the first bit of compatibility that I broke.

Then I took a long look at the difference between myReceiver.enableIRIn() and myReceiver.resume(). I began to realize how confusing it was deciding which one should be called when. You would call enableIRIn() during your setup procedure and use resume() when you were done decoding and ready to resume receiving. However if you did a send function it would trash your use of the hardware timers and you needed to again enableIRIn(). It seems like the difference between these functions was minimal and was better to just get rid of resume() and always use enableIRIn() to simplify things.

Another reason to get rid of resume() was that it now functions differently when using auto resume in IRLib2 versus using an external decode buffer in IRLib1. Under IRLib1 if you were using an extra buffer your code would look something like this…

```
void loop() {
  if (My_Receiver.GetResults(&My_Decoder)) {
    //Restart the receiver so it can be capturing another code
    //while we are working on decoding this one.
    My_Receiver.resume();
    My_Decoder.decode();
    My_Decoder.DumpResults();
  }
}
```

Immediately after GetResults() returns true you would call the resume() method. But under the new auto resume system, the code would look like this…

```
void loop() {
  if (myReceiver.getResults()) {
    myDecoder.decode();
    myDecoder.dumpResults();
    myReceiver.enableIRIn(); //must be after you are finished
  }
}
```

The call to resume() or enableIRIn() or whatever you call it has to be after you have completely finished processing your data. Because the actual functionality of resume() was so significantly different, I wanted to highlight this fact by getting rid of it altogether and using enableIRIn() instead.

Under IRLib1 the use of an extra buffer was a method of the decoder class. But under the auto resume system it makes more logical sense that it is a function of the receiver object rather than the decoder so that was moved as well.

An additional item was called to my attention while working on Mr. Staples code. In some instances he used extremely descriptive variable names. Some of them longer than I would have liked but descriptive nevertheless. In other instances variable names like "buffer1" and "buffer2" made it extremely difficult to keep track of which buffer was holding which data under what circumstances. Many of the variable names used in IRLib1 were holdovers from the original IRremote library and were not the most useful or descriptive. I never did like that global variable "irparams". Once I had made the decision that we were not going to maintain complete backward compatibility, I went on a rampage renaming variables into what I hoped would be clearer. For example those global variables are all related to the receiver and they are global so let's call them "recvGlobal" instead of "irparams". Instead of buffer1 and buffer2 we now have recvBuffer and decodeBuffer which lets you know what they are really all about. Some of my variable names are longer than I would have liked but the readability and understandability of the code is much better than it was.

Finally all of this renaming of variables led me to the decision that I need to be more rigorous about my coding style and my capitalization of variables. In Appendix C I have outlined the coding standards that I tried to use in this rewrite. I'm requesting that anyone who contributes code to this library in the future tries to maintain those standards as well. In return I promise I will be much more likely to incorporate your code as is without trashing it and rewriting it like I did with Mr. Staples fine contribution.

I would also like to take this opportunity to suggest that if you have an idea for a major change to the code that you discuss it with me before you invest a lot of time that might go to waste. If we can come to a common vision on what the new features should be and how they should work, then we can work together implementing them and not waste time working on features that are not going to get incorporated in the long run. Of course the beauty of open source is that if you do want to go a different direction from me, you are free to make your own GitHub fork and create your own libraries. That's what I did with the original IRremote.

Other changes in the code which violated backwards compatibility concern constructors for the receiver classes. Previously some of the constructors would be passed pin numbers however the PCI giving receivers you would pass interrupt numbers. The numbering system for interrupt numbers used by Arduino's attachInterrupt() is a bit bizarre to begin with. Their interrupt numbers do not correspond directly to the interrupt numbers on the device's datasheet. As they have begun to expand into more platforms, their approach to this is evolving. They recommend you do not call attachInterrupt() with a pin number directly but rather you should use the pin number and the function digitalPinToInterrupt() to convert that pin number into the proper interrupt number. That made our life easier because it means that all of the receiver classes are passed pin numbers instead of some using interrupt numbers and others using and others using pin numbers. It makes it easier to experiment switching back and forth between the various receiver classes.

One of the side effects of this change is that all of our sample sketches now use pin 2 as the default input pin rather than 11. The IRrecv and IRrecvLoop classes can still use any

available digital input pin however making pin 2 the default in our examples makes it easy to switch to the PCI version.

Other features such as the copyBuf function which were of minimal use have been eliminated. Other changes are noted throughout the documentation.

We hope this rather lengthy explanation of our thought processes and motivations in this major rewrite of the library have been useful and educational.

# Appendix B Understanding IRP Notation

When implementing a new protocol it's always much easier if you have some sort of documentation explaining the protocol rather than having to completely reverse engineer the timing by analyzing samples. As you search around the Internet for such information you will find that many times infrared protocols are described in what is called IRP notation.

As I have researched the protocols we have already implemented and as I have planned future protocols, my go to source for this research has been the following webpage…

http://www.hifi-remote.com/johnsfine/DecodeIR.html

It documents dozens of protocols including all those we currently support. It does so using this strange IRP notation. The official definition of this notation is really complicated and difficult to understand. That webpage includes a section titled "Brief and incomplete guide to reading IRP". One of the problems with that explanation and many other portions of that document is that it was written for people who wanted to implement these protocols using a special programmable remote known as a JP1 compatible remote along with analysis and programming software designed for that use.

Much of the information that document deals with is the interpretation of the individual bits of data to represent things like device number, sub device, function, subfunction, checksums and other data. In general IRLib doesn't really care about those things. It just sees some number of bits of data and doesn't care what they represent. In this section we will give a modified explanation of how to read IRP protocol dealing primarily with the issues that we care about in IRLib.

Here are a few IRP definitions that we will refer to in the discussion.

Sony 12: {40k,600}<1,-1|2,-1>(4,-1,F:7,D:5,^45m)+
Panasonic_Old: {57.6k,833}<1,-1|1,-3>(4,-4,D:5,F:6,~D:5,~F:6,1,-44m)+

**Basic information:**

Each definition begins with some basic information enclosed in scroll brackets. The first item is the modulation frequency. For example the definition for Sony says "40k" which means 40 kHz modulation. Note that we store modulation frequencies in an uint8_t variable but some of the frequencies include a digit after the decimal point. For example Panasonic_Old specifies 57.6k which we round up to 58.

The next item inside the scroll brackets is the base timing units. In the Sony example it says 600. That means that all of the timing values used throughout the protocol will be some multiple of 600 µs unless otherwise specified. For example the header information for Sony is given as 4,-1. Positive numbers mean a mark and negative numbers mean a space. These are multiplied by the base timing unit. Therefore they header for Sony is 4*600,1*600 or 2400µs mark followed by 600µs space.

An optional third item in the scroll brackets is whether data is transmitted lsb or msb first. The explanation says that if not specified, the default is least significant bits first. But I think that may be a typo. Either that or we are misunderstanding what they mean by that. It may be that because we ignore the actual contents of the bitstream that this doesn't matter. The order in which we send our data is the same order in which it was received by the decoder. For this

reason some of the binary values we used represent the internal data may be different than those used by other reference material such as LIRC or other databases. As long as we are consistent between the way we receive and the way we send it doesn't matter the order of the bits.

**Bit Encoding Rules:**

The next item in the definition is an explanation of how individual zeros and ones are encoded. The information is enclosed in angle brackets and separated by a | character. The numbers are the multiples of the base time unit. In our Sony example it is <1,-1|2,-1>. The first pair of numbers describes how to encode a zero and the second set of numbers describes how to encode a one. We multiply these numbers times Sony's base timing unit of 600. That means that Sony encodes a binary zero as 600 mark, 600 space and a binary one 1200 mark, 600 space. Alternatively our Panasonic_Old encodes bits where the length of the spaces changes rather than the length of the marks. It uses the definition <1,-1|1,-3> and a base timing unit of 833. This means that a binary zero is 833, 833 and a binary one is 833, 2499.

The vast majority of protocols will only have 2 entries in this section for encoding a "0" and a "1". However some protocols encode bits two at a time. When we encountered the IRP notation for this protocol we weren't exactly sure how to interpret it. For example the definition for DirecTV says that it uses <1,-1|1,-2|2,-1|2,-2>. After getting a look at an actual dump of one of these signals it became obvious that each pair of numbers actually encodes 2 bits. So the first two numbers in the sequence represent 00, next is 01, and 10 and 11. Also the RCMM protocol uses a different system for encoding 2 bits at a time. See the section on those particular protocols for more details.

**Stream Data:**

The next item in the sequence is an explanation of the actual stream of bits which are transmitted. The specification is enclosed in normal parentheses. Most protocols start out with some fixed header information that is defined using 2 numbers. The first is positive indicating a mark and the second is negative indicating a space. These numbers are multiplied by the base timing unit. We already illustrated that the Sony header is defined as (4,-1… With a base timing unit of 600 means they header is actually 2400 mark, 600 space. Anytime within the stream data that you see a number by itself such as this. It means an individual mark or space which is a multiple of the base timing unit.

Note that the end of the Panasonic_Old sequence just inside the close parentheses you will see …1,-44m). That "1" is indicating that we need a mark of the base timing unit which in this case is 833 followed by 44 milliseconds of space to terminate the sequence. The suffix "m" tells us that it is in milliseconds rather than some multiple of the base unit.

In between the header information and any closing information you will see a number of specifications that begin with a capital letter. These are bit fields within the overall stream of bits. We interpret the letters to mean things like D=device, F=function, S=subfunction, C=check bits, T= toggle bits. With the exception of toggle bits, we pretty much ignore the distinctions. These capital letters are followed by a colon and a digit optionally followed by another colon and another digit. The explanation given is as follows.

**Bitfield**: D:NumberOfBits:StartingBit. E.g. if D=47= 01000111, D:2:5 means x**10**xxxxx. D:2:5 = 10b = 2. ~ is the bitwise complement operator. ~D =10111000. Specifying the StartingBit is optional. D:6 is equivalent to D:6:0.

Let's look at our entire Panasonic_Old definition

Panasonic_Old: {57.6k,833}<1,-1|1,-3>(4,-4,D:5,F:6,~D:5,~F:6,1,-44m)+

This tells us it is 57.6 kHz modulation. The base timing unit is 833. A zero is encoded by 833, 833 and one is 833, 2499. The bitstream itself begins with a header of 4*833, 4*833. This is followed by D:5 five bits of device code followed by F:6 six bits of function code. We then take the bitwise complement of the device and function codes and repeat them. We conclude with a single mark of 833 followed by 44m of blank space. Although IRLib treats this as a single stream of 22 bits, you could add additional code to the decoder to assure that the second group of 11 bits is the bitwise complement of the first group of 11 to ensure that this was a valid sequence.

Sometimes a particular field of bits is so complicated you need a separate expression to define it. Typically this is done with a complicated system of check bits. For example the GICable specification is

GICable: {38.7k,490}<1,-4.5|1,-9>(18,-9,F:8,D:4,C:4,1,-84,(18,-4.5,1,-178)*)
where {C = -(D + F:4 + F:4:4)}

As you can see it has an F:8 eight bits of function followed by D:4 which is four bits of device followed by C:4 four bits of check bits computed by the formula given at the end. In that formula F:4 means the lowest order four bits of the function code and F:4:4 means for bits of the function code starting with fourth bit. Recall that the definition is F:number bits:starting bit. So in this case is the highest order for bits of the function. Add those together with the device code and make them negative and take 4 bits of that and you get the check bits. In our implementation, we ignore all of that and simply treat it as 16 bits of data. We do not compute check bits or verify any relationship between bit fields.

**Extent and Repeat:**

Most protocols conclude with some length of a mark which we describe as a stop bit. And this is followed by some amount of blank space which we call the "extent". In the case of Sony we noted that the extent was -44m which is 44 milliseconds of blank space. Sometimes the extent does not have a suffix so it is a multiple of the base time. For example NEC2 uses…1,-78 with a base time of 564. This means it concludes with a mark of 564 µs and a space of 78*564= 43992µs. If the extent is denoted with a carrot ^ rather than minus sign is not the length of the trailing space rather it is the entire length of the frame. For example RC6 has the extent of ^114m which means that the entire sequence needs to be padded out until the whole frame is 144 milliseconds long. That means occasionally we need to keep track of how long a sequence is so that we know how much blank space we need to fill up at the end.

After the close parentheses defining a sequence of bits there is an indication of how that sequence might be repeated. A trailing + means send one or more times. A trailing 3 means send 3 times; 3+ means at least 3 times. A trailing * means send zero or more times.

If you have read through the detailed sections on each protocol you know that NEC1, NECx1, and GICable all designate repeat codes to designate a special sequence known as a "ditto". Let's look at the NEC specification to explain how that works.

NEC1: {38k,564}<1,-1|1,-3>(16,-8,D:8,S:8,F:8,~F:8,1,^108,(16,-4,1,^108)*)

It uses 38 kHz modulation with a base time of 564 µs. A zero is 564, 564 and a one is 564, 3*564. The header contains 16*564, 8*564 followed by 32 bits of data. Although we really don't care about the individual fields we will note that it is eight bits of device code, eight bits of either subdevice or subfunction we aren't sure which, eight bits of function followed by the bitwise complement of the same eight function bits. If concludes with a mark of 564 and uses and extent of 108*554 which means that the entire sequence of the frame should be padded out to that value. The length of the stream of data bits themselves changes depending on how many zeros and how many ones there are. That means that the length of the lead out space must be adjusted to make the total length of the frame that value. After the basic specification there is another sequence in parentheses with an asterisk at the end. That means that the first frame is followed by zero or more copies of what is in the inner parentheses. This is the famous ditto sequence. It starts with a header of 16*564, 4*564 followed by a mark of 564 and sufficient space to make the entire sequence 108*564. There is no data in the ditto.

This should give you enough information to understand most IRP definitions. We've also pointed out some of the things that you can ignore in the reference material we have linked for you. More details are available in the reference material if you're interested in going deeper into this specification.

# Appendix C. Programming Style

Despite what I've said in Appendix A about the way I threw away a contributor's major new feature and rewrote it from scratch, I do very much welcome contributions to the code large and small. But I would invite you to get in contact with me before making major changes so that we can reach an agreement on how a new feature should be implemented. That way you don't end up wasting time working on something that might not get incorporated into the final version or might need to be rewritten to make it compatible with other things I'm working on.

One of the things you could do to ensure your code makes it into this library is to try to adopt some of the coding styles that I'm using. Because IRLib2 was a significant rewrite and we have renamed many variables and functions, we took the opportunity to try to standardize coding style. You can research the topic all over the Internet and discover that there is no single official coding style established. Rather than debate the merits of various styles I'm just going to tell you what I am using and I hope you will use it when submitting any improvements to this library. There are probably areas where I'm not even following my own rules so this is really just suggested guidelines rather than some absolute requirement.

**Capitalization:**

All functions and variable names should use what is called "camelCase". That is an initial lowercase character and a capital letter for subsequent words within variable name. It is of course named that because the capital letters create humps in the middle. There are no underscores. In the past we had used some "TitleCase" which is sometimes referred to as "PascalCase" or alternatively had used "snake_case" but it appears that most Arduino code uses camelCase so that's what we're using. Some people like to use one style for variable names and another for function names and another for class names but I like to pick a style and stick to it. The only exception is macros. As is the case with most style standards, macros are all uppercase separated by underscores such as "REALLY_IMPORTANT_MACRO".

**Indentation:**

I grew up reading Kernighan and Ritchie's book The C Programming Language, and it uses a style where the open brace appears at the end of a line, the contained code is indented, and the closing brace is in line with the initial line of code like this…

```
if (foo<bar) {
  dealWithIt();
}
```

I do the same thing for class and function definitions such as…

```
void dealWithIt(void) {
  sayPrayers();
  haltAndCatchFire();
}
```

However I recently learned that strict K&R style would put the open brace on the next line like this…

```
void dealWithIt(void)
{
  sayPrayers();
```

```
  haltAndCatchFire();
}
```

I'm going to stick with keeping it at the end of the first line regardless of its usage. This is all just a matter of style.

**Naming conventions:**

While it's always possible to have variables clash with one another when using libraries from multiple sources, I have not bothered to make use of any namespaces. Arduino programs are reasonably small so the chance for conflicts is small. However I did decide that any function which is not part of a class and is global in nature should have the prefix "IRLib_" to highlight the fact that it is part of this library. The only other convention is that I tried to make things as descriptive as possible without "usingExtremelyLongVariableNamesForNoApparentReason". Even the use of a variable like recvGlobal.decodeBuffer seems too long for me but I decided it was necessary for clarity.

**New protocols and platforms:**

Initially I had decided in IRLib1 that I would not support new protocols within the library itself. I told contributors to submit example code in the same way that I had done for Samsung36, DirecTV etc. Under this new system, I welcome anyone adding new protocols for ordinary consumer electronic devices such as TVs, cable boxes, DVR's, Blu-ray etc.

I probably will not incorporate protocols for devices such as air-conditioners and fans because they are extremely long sequences that require larger than normal buffers. You can submit code that we will put in a special "unsupported" folder but we will not assign them a protocol number and will not include them in IRLibCombo.h.

The same holds true for protocols for toys such as IR controlled helicopters and drones and electronic pets such as the Zoomer dinosaur. We welcome the code but we will not make it an official protocol of the library. Such protocols should use protocol number 90. If you need to make it a permanent part of your library, you can edit the files yourself.

**Variable types:**

Because this code may be used beyond traditional 8-bit Arduino platforms, we want to make sure we are clear about the number of bits that various variables are using. So rather than specifying variable types as char, int, long int etc. we are using the built-in types uint8_t, uint16_t, uint32_t, and if necessary their signed alternatives. The only exception is that variables which store true/false values will continue to be defined as bool to highlight the fact that we are only interested in whether or not its value is true or false. If you define your own custom types you should use the "_t" suffix to highlight the fact that it is a variable type.

**Code comments:**

I'm not very rigorous about my commenting style but in general I use multiline comments using /*…*/ at the top of the file to explain its contents and between functions. Use groups of single-line amounts with double slashes inside functions or at the end of a line of code. In general I tried to keep line lengths for comments in the code itself something reasonable. I do not have a hard and fast rule on how many characters per line. Just try to make it readable.

**Final thoughts:**

As I have said, these are just guidelines to try to keep code consistent and readable and make it easier to maintain. And again I promise to be more likely to incorporate your

suggestions and submissions into the actual distribution. The strength of open source is that it is community driven. I want to take advantage of that community. If you have any questions contact me at cy_borg5@cyborg5.com