User's Manual for IRLibCP.

A Library for Receiving, Decoding and Sending Infrared Signals Using Circuit Python

(June 2017)

This library is designed for using Adafruit Circuit Python on various Adafruit Express boards for receiving, decoding and sending infrared signals such as those used by infrared remotes on consumer electronics. It is a translation from the original IRLib2 written in C++. A big disclaimer is in order. This package is only the second piece of Python code I've ever written. My guess is that serious Python programmers will read this code and cringe in horror. This should be considered an extreme early beta version that hopefully will grow, become more stable, and more efficient. Also the sample code and even this documentation is very preliminary. I've barely proofread this document which is basically an edited version of the original IRLib2 documentation.

Most of the design decisions were based on the idea that we wanted to mimic the API of the original IRLib2 in C++. Variable names and module names following the same conventions as that original work. While it is my hope that eventually this documentation will involve sufficiently to allow this library to stand on its own, I admit at this point it might be useful if you were already familiar with the original C++ version.

If you are familiar with the original IRLib2 you should skip to Appendix A first to read about the differences between this library and the C++ version. We will also make note of things differences throughout the document as the issues arise.

NOTE: The entire basis of the hardware specific portion of this canoe depends upon the Circuit Python "pulseio" module. Details on this module can be found at https://circuitpython.readthedocs.io/en/latest/shared-bindings/pulseio/__init__.html This module is only available on the "Express" versions of Adafruit boards specifically: Feather M0 Express, Circuit Playground Express, and Metro M0 Express. Although Circuit Python will run on standard Feather M0, Arduino Zero, ESP 8266 and other boards, those boards do not support "pulseio" and therefore cannot be used with this package.

All of the sample code has been written for the Circuit Playground Express which has a built-in IR LED and a built-in IR receiver. The code references "board.REMOTEIN" for input and "board.REMOTEOUT" for output. However the code has also been tested on a Feather M0 Express and you can substitute whichever digital input and output pins you want to use. For example during development we used "board.D5" for input and "board.D9" for output because we are connected are IR LED driver circuit and IR receiver to those pins. See the hardware section for how to attach IR LEDs with driver circuits and IR receivers.

Because of the memory limitation of a Circuit Python platform, it may not be possible to implement programs using all of the supported protocols simultaneously. Also Circuit Python does not allow us to be quite as clever as we were in the C++ version of the library when it comes to combining multiple protocols into a single class automatically. If you need to use

multiple protocols will have to write a few lines of code yourself to combine them into a single sending or decoding class.

Section 1 of this document is a complete reference of the library with details on individual protocols. We recommend however that you skip to the tutorials section 2 in order to get started.

This document is current for the original release of IRLibCP and may not be fully updated if there are minor bug fix releases. Always see CHANGELOG.txt for the latest information.

1. IRLib Reference

This section is intended to be a complete reference to the modules and classes used in the library. It is divided into the following sections:

1.1 Receiver Class

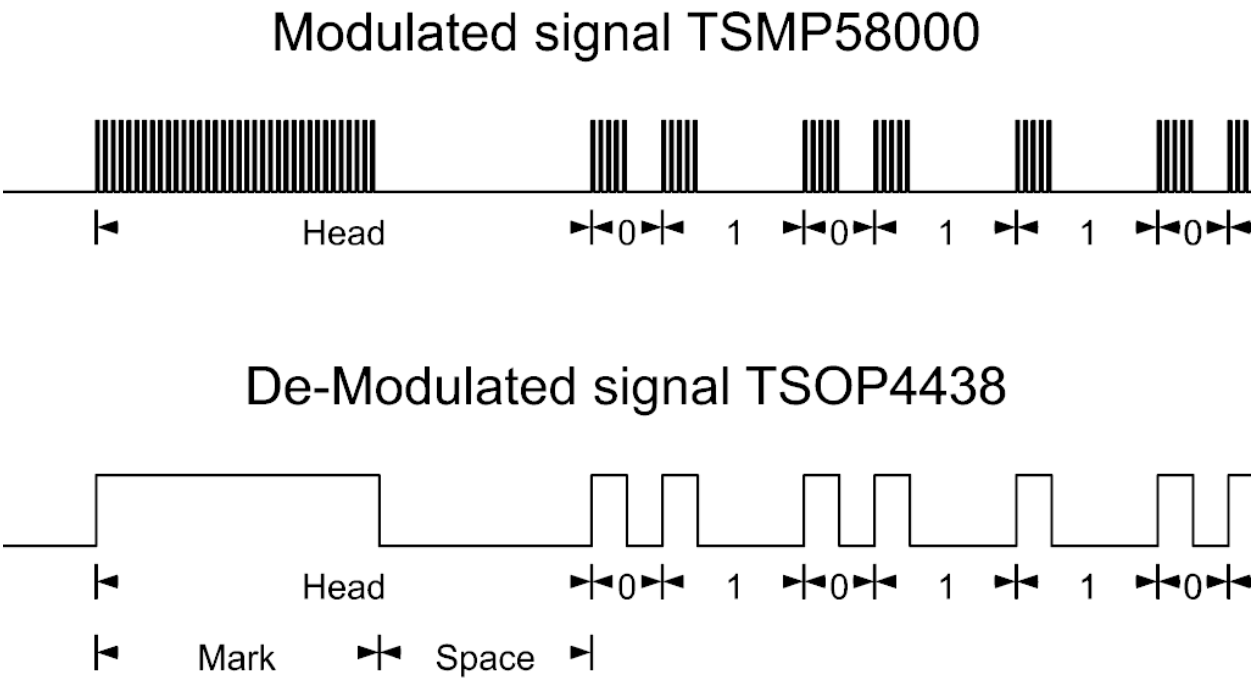
IR data is received on an input pin of the Arduino as a square wave of pulses of varying lengths. The length of the on and off pulses encodes the data. It is the job of the receiver class to record the timing of the pulses and the spaces between them. The data is stored in an array and passed to the decoder class. This section contains an overview discussion of the receiving process, with an explanation of the receiver class.

Only one receiver class has been implemented in Circuit Python. It is in the module IRrecvPCI and is similar to the identically named class in the original IRLib2. It uses pin change interrupts to detect the transition of the signal from high to low and back again. It makes use of the pulseio module to receive the signals and record the time intervals between transitions. IRLibCP does not implement a timer driven receiver classes such as IRrecv nor a loop-based received such as IRrecvLoop nor can it be used to detect frequency modulation of the signal like the IRfrequency class.

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. Frequency detection can only be done with the original IRLib2 written in C++.

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".

You should import the IRrecvPCI module and create an instance of the receiver class. The constructor requires you to pass it the pin number to which you connected your TSOP receiver device. This means that you should also import the "board" module to pass the pin number. For example…

```
import board
import IRrecvPCI
#create instance of receiver using pin D5
myReceiver=IRrecvPCI.IRrecvPCI(board.REMOTEIN)
```

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".

The data is stored in an array type 'H' which is unsigned short values named IRrecvPCI.decodeBuffer. NOTE: In the original IRLib2 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. However IRLibCP does not record that data and begins with the timing of the first received mark. Even-numbered elements of the array contain the duration of the marks and odd-numbered elements contain the duration of the spaces.

Because the receiver is interrupt driven, once you have initialized it, 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 myReceiver.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.

Here is an extremely simple sample sketch showing the use of the IRrecvPCI class and the NEC protocol decoder.

```
import board
import IRrecvPCI
import IRLib_P01_NECd

myDecoder=IRLib_P01_NECd.IRdecodeNEC()
myReceiver=IRrecvPCI.IRrecvPCI(board.REMOTEIN)
myReceiver.enableIRIn()
print("send a signal")
while True:
   while (not myReceiver.getResults()):
         pass
   if myDecoder.decode():
         print("success")
   else:
         print("failed")
   myDecoder.dumpResults()
   myReceiver.enableIRIn()
```

1.1.2. The IRrecvPCI class

The receiver class contains only three methods. There is an initializer method __init__(pin), enableIRIn(), and getResults(). We will discuss each of these below.

1.1.2.1 Constructor IRrecvPCI

Before you can do anything you must create a receiver object. Here is an example creating "myReceiver" and telling at that your receiver device is connected to pin D5. Any digital input pin that can use pin change interrupts can be specified.

```
myReceiver=IRrecvPCI.IRrecvPCI(board.REMOTEIN)
```

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. You would invoke it as follows…

```
myReceiver.enableIRIn()
```

This method sets the input pin to input mode and uses the pulseio.PulseIn method to set up a buffer a maximum of 150 entries to receive timing values. The pin change interrupt is enabled and is IR signals are received, the timings are recorded. It

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. NOTE: we do not support the auto resume feature of the original IRLib2.

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 method returns "True" if a complete frame of data has been received and returns "False" otherwise. For example…

```
while (not myReceiver.getResults()):
      pass
```

The pulseio.PulseIn system continues to record data and doesn't have a method for determining for certain when a frame of data has completed. The getResults method notes the length of the receive buffer each time that you call it. If the length is nonzero and does not change over a period of a half second, it presumes that the frame is complete. It then scans through the interval data to see if there is an especially long gap. If it finds such a gap it discards the remaining values so that you are only given the data for a particular frame. This means that if one or two frames of data come in rapid succession, part of the second frame may be discarded.

When a complete sequence has been received by the class, it turns off the PulseIn receiving and copies the data to an array of type 'H' called IRrecvPCI.decodeBuffer. You can access that buffer directly for whatever purpose such as retransmitting the raw data values or you can use one of the decoding methods we provide to analyze the data and turn it into a more compact and usable value. Once getResults returns true if you want to continue receiving data you must again call enableIRIn().

1.2 Decoder Classes

IR signals are encoded as a stream of on and off pulses of varying lengths. The receiver class only records 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. NOTE: we have not yet implemented the hash code decoder method from the original IRLib2.

Several of the protocols are decoded into 32-bit unsigned integer values however Circuit Python does not support unsigned integers that large. In the original IRLib2 there was one protocol specifically Samsung36 which was 36 bits long. It return values in 2 variables. One was a 16 bit "address" value and the other was a 20 bit "value". In IRLibCP for any 32-bit protocol

we return the high order 4 bits in the "address" and the lower order 28 bits in the "value". The Samsung36 continues to split address and value as 16 bits and 20 bits respectively but all other protocols greater than 28 bits uses the 4 bit/28 bit split.

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.

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. The base class also defined some other common methods such as those used by RC5 and RC6 protocols.

Unlike the original IRLib2 which had a very clever method of combining multiple protocols into a single decoding class, there is no automatic way to combine protocols without writing the code to do so yourself. Below is an example of how to receive and decode a single protocol in this case the NEC protocol. Later we will give you an example of how to combine multiple protocols. However because the memory restrictions our experience shows that you cannot create a single decoder to decode more than about five or six protocols at once.

```
import board
import IRrecvPCI
import IRLib_P01_NECd

myDecoder=IRLib_P01_NECd.IRdecodeNEC()
myReceiver=IRrecvPCI.IRrecvPCI(board.REMOTEIN)
myReceiver.enableIRIn()
print("send a signal")
while True:
   while (not myReceiver.getResults()):
         pass
   if myDecoder.decode():
```

```
        print("success")
    else:
        print("failed")
    myDecoder.dumpResults(True)
    myReceiver.enableIRIn()
```

In this example we create an instance of the NEC decoder that we call myDecoder. Each protocol module is of the form "IRLib_Pnn_NAMEd.py" where "nn" is a two digit protocol number and "NAME" is the name of the protocol. After the name you have the letter "d" for decode. The sending modules use the same system and conclude with the letter "s" for sending. We find myReceiver.getResults() has returned "True" we invoke the decoder using the decode method. It returns True on success and False otherwise. We then use myDecoder.dumpResults() to see the results of the decoding process regardless of whether it succeeded or failed. The myDecoder.dumpResults() method has an optional parameter that if "True" gives you a verbose output and if "False" gives you a brief output. The default if not specified is "True". After decoding, we then tell the receiver that we finished and we are ready for another frame of data by calling myReceiver.enableIRIn().

In the sample example we wait for getRresults to return True before doing anything else. However in a more practical example we would go off and do other things and only occasionally poll getResults to see when the data was available. We can do this because the receiver class is interrupt driven.

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 IRLibDecodeBase.py.

The constructors for the base class and for any of its derived classes take no input. Although not part of the base class, each protocol class has a method

```
myDecoder.decode()
```

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 28-bit variable myDecode.value. The number of significant bits is stored in myDecode.bits. Some protocols have more than 28 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.py as follows…

```
UNKNOWN=0
NEC=1
SONY=2
RC5=3
RC6=4
PANASONIC_OLD=5
JVC=6
```

```
NECX=7
SAMSUNG36=8
GICABLE=9
DIRECTV=10
RCMM=11
CYKM=12
Pnames= ["UNKNOWN", "NEC", "Sony","RC5", "RC6", "Panasonic old",
"JVC", "NECx", "Samsung36", "G.I.Cable", "DirecTV", "rcmm", "CYKM"]
```

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…

```
success
Decoded NEC (1): Value:0x1a0f00f Adrs: 0x6 (32 bits)
Raw samples(67):  Head: m9041 s4535
 0:m536 s607     1:m562 s1707              2:m563 s1708   3:m562 s579
 4:m541 s598     5:m561 s580               6:m560 s582    7:m539
s1732
 8:m538 s1733    9:m566 s574              10:m566 s1705  11:m565 s576
12:m564 s578    13:m565 s575              14:m561 s579   15:m561 s582

16:m558 s1710   17:m560 s1714             18:m536 s1734  19:m566
s1705
20:m535 s605    21:m569 s571              22:m537 s604   23:m534 s608
24:m561 s580    25:m560 s579              26:m570 s571   27:m569 s571
28:m559 s1712   29:m568 s1703             30:m537 s1734  31:m569
s1703

33:m551
```

The output identifies this as NEC protocol which is protocol “1”. The received value in hexadecimal is 0x61a0f00f and is 32 bits long. Because Circuit Python cannot handle 32 bit values, the return value has been split into the higher order 4 bits in the address field and the low order 28 bits in the value field. The stream contained 67 intervals which is the number of raw samples. It then dumps out all of the values from the decode buffer which is IRrecvPCI.decodeBuffer. The first 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.

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 NEC (1): Value:0x1a0f00f Adrs: 0x6 (32 bits)
```

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

```
   myDecoder.ignoreHeader=True
```

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.

Apart from decode(), dumpResults() and the ignoreHeader value the remaining items in the base encoding class are for internal use only.

1.2.3. Decoding Multiple Protocols

In to decode more than one protocol you must combine multiple protocol decoding modules and the base decoder module into a custom class that can decode the protocols you want. In our experience there is insufficient memory to decode more than about five or six protocols in a single class and this is only after having precompiled the code from its original filename.py into filename.mpy. If you do not precompiled the code you may only have room for two or three protocols. Here is a sample program called multiple_protocol.py that illustrates how to do it.

```
# IRLibCP by Chris Young. See copyright.txt and license.txt
# Sample program to decode multiple protocols.
# In this case NEC, Sony and RC5
import board
import IRLibDecodeBase
import IRLib_P01_NECd
import IRLib_P02_Sonyd
import IRLib_P03_RC5d
import IRrecvPCI

class MyDecodeClass(IRLibDecodeBase.IRLibDecodeBase):
   def __init__(self):
         IRLibDecodeBase.IRLibDecodeBase.__init__(self)
   def  decode(self):
         if IRLib_P01_NECd.IRdecodeNEC.decode(self):
               return True
         if IRLib_P02_Sonyd.IRdecodeSony.decode(self):
```

```
            return True
        elif IRLib_P03_RC5d.IRdecodeRC5.decode(self):
            return True
        else:
            return False

myDecoder= MyDecodeClass()
myReceiver=IRrecvPCI.IRrecvPCI(board.REMOTEIN)
myReceiver.enableIRIn()
print("send a signal")
while True:
   while (not myReceiver.getResults()):
        pass
   if myDecoder.decode():
        print("success")
   else:
        print("failed")
   myDecoder.dumpResults(True)
   myReceiver.enableIRIn()
```

Note that when creating the custom class is a derived class of IRLibDecodeBase but does not need to be a derived class of the component protocols. We are not sure why this works. In C++ this combination class would've had to be derived from all of the component protocol classes. Similarly in the initialization routine we can get by with only initializing the base class. This is because none of the protocol classes have anything that needs to be initialized other than those components from the base class. The only method in our derived class is the decode method which individually calls the decode methods of each of the protocols and if it returns true then we return true.

Complete details on the individual protocols can be found in section 1.4 Protocol Details.

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.

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. NOTE: Because Circuit

Python does not support 32-bit integers, we have split 32 bit values into 2 variables. The high order 4 bits are called the "address" and the low order 28 bits are called the "data". Values that you acquired from the original IRLib2 written in C++ should be completely compatible if you split the 32 bit values along those lines. Note that the Samsung36 protocol continues to split into a 16-bit address and a 20 bit data value as in the original IRLib2.

We have not tested any applications which both send and receive IR data in the same program. In the original IRLib2 there were some restrictions on doing so. But because we do not yet understand the internal details of the pulseio module we do not yet know if there are any such restrictions. So do so at your own risk.

1.3.2. IRsendBase Class

The code for the base sending class is implemented in IRLibSendBase.py. We will not go into describing the internal methods used by the abstract base class and only focus on the methods used by the individual protocols. The constructor takes a single parameter which is the output pin to which you connected your IR LED and driver circuit. The "send()" method is used to actually transmit the data. The number of parameters and the meaning of those parameters to the "send" method depends on the particular protocol. Here is a sample program that sends a 32 bit value using the NEC protocol.

```
import board
import IRLib_P01_NECs

mySend=IRLib_P01_NECs.IRsendNEC(board.REMOTEOUT)
Address=0xF
Data=0xEDCBA98
print("Sending NEC code with address={}, and
data={}".format(hex(Address),hex(Data)))
mySend.send(Data, Address)
```

In this example we create an instance of the NEC sender that we call mySen. Each protocol module is of the form "IRLib_Pnn_NAMEs.py" where "nn" is a two digit protocol number and "NAME" is the name of the protocol. After the name you have the letter "s" for send. The decoding modules use the same system and conclude with the letter "d" for decoding.

1.3.3. Sending Multiple Protocols

If you need to send more than one protocol you need to create a custom sending class that combines multiple protocols into a single class and then chooses the protocol based on a parameter passed to the "send" method. Here is a sample program that sends 3 different protocols.

```
# Sample program for sending multiple protocols.
import board
import IRLibSendBase
import IRLibProtocols
import IRLib_P01_NECs
import IRLib_P02_Sonys
import IRLib_P05_Panasonic_Olds
```

```
class MySendClass (IRLibSendBase.IRLibSendBase):
   def __init__(self,outPin):
         IRLibSendBase.IRLibSendBase.__init__(self,outPin)
   def send(self,protocolNum, data, data2=0):
         if protocolNum==IRLibProtocols.NEC:
               print("Sending NEC value:{}
address:{}".format(hex(data),hex(data2)))
               IRLib_P01_NECs.IRsendNEC.send(self,data,data2)
         elif protocolNum==IRLibProtocols.SONY:
               print("Sending Sony value:{}
bits:{}".format(hex(data),data2))
               IRLib_P02_Sonys.IRsendSony.send(self,data,data2)
         elif protocolNum==IRLibProtocols.PANASONIC_OLD:
               print("Sending Panasonic_Old
value:{}".format(hex(data)))

   IRLib_P05_Panasonic_Olds.IRsendPanasonic_Old.send(self,data)
#end of MySendClass

mySend=MySendClass(board.REMOTEOUT)
mySend.send(IRLibProtocols.NEC,0x1a0f00f,0x6)
mySend.send(IRLibProtocols.SONY,0x540c,15)
mySend.send(IRLibProtocols.PANASONIC_OLD,0x37c107)
```

Note that the custom sending class needs only to be derived from the base class and not from the individual protocol classes the same way that multiple protocol decoders were created in the previous sections. Because different protocols use different numbers of parameters you need to make sure that you specify defaults in case certain parameters are not passed and that your custom sending routine has enough parameters to handle all of the variations of the individual protocols. We have created a sample program called send_patterns.py which will handle all 11 supported protocols. You might want to start with that program and trim it down to use only the protocols you need.

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 send() method makes use of its various parameters differently in some protocols than others. The 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.

Each protocol module is in 2 different files: one for sending and one for decoding. The module names are of the form IRLib_Pnn_NAMEx.h where "nn" is the protocol number, "NAME" is the name of the protocol, and the "x" is either "d" for decoding or "s" for sending. For example the Sony decoder is IRLib_P02_Sonyd and the sender is IRLib_P02_Sonys.

Because Circuit Python does not support 32 bit integers, any protocol that uses more than 28 bits divides the value so that the high order 4 bits are in a field called "Address" and the low order 28 bits are in the data value. The exception is Samsung36 36 bit protocol which is

always divided into the high order 16 bits for the address and the low order 20 bits from the data.

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. IRLibCP 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.

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". IRLibCP does not do this for you. It will send you a value of -1 to designate that this is a repeat code.

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. IRLibCP 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 -1 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. IRLibCP 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. Because IRLibCP cannot measure frequency, 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 IRLib_P01_NECd.py and IRLib_P01_NECs.py. The definition for the send method for this protocol is…

```
  def send(self, data, address=0,kHz= 38):
```

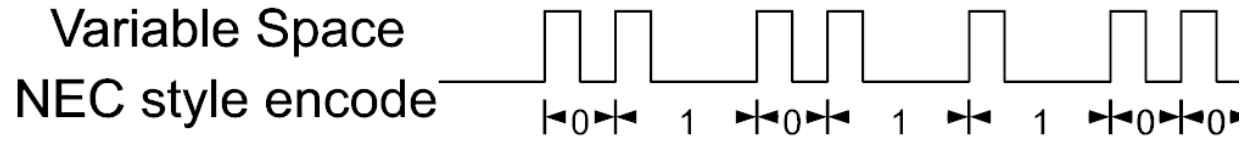
As mentioned before the high order 4 bits should be set in the "address" field in the low order 28 bits in the data field. The optional third parameter allows you to support Pioneer by changing the frequency to 40 instead of the default 38. Here are some examples if you create an instance of the NEC sending class by itself.

```
mySender.send(0x234abcd,0x1)  #send NEC protocol
mySender.send(-1)             #send NEC ditto sequence
mySender.send(0xb12c3ff,0xa, 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 IRLib_P02_Sonyd.py and IRLib_P02_Sonys. The definition for the send class is…

```
  def send(self, data, bits):
```

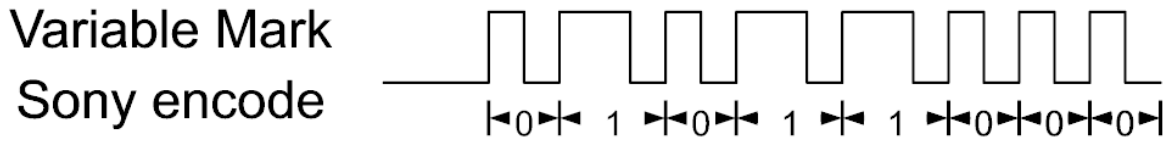
Because the longest Sony variety is only 20 bits we do not need to split it up between the address and data values. Here is how to send a code using Sony protocol.

```
mySender.send(0x12abc,20)   #send 20 bits Sony protocol
mySender.send(0x12ab, 15)   #send 50 bits Sony protocol
```

The Sony specification requires that every time you press a button, the same signal is sent three times. The sending method automatically sends the signal three times for you. When receiving Sony protocol is unlikely that you will be able to capture and process the first frame fast enough in order to detect the second or third frames of repeated data. Circuit Python simply is not quick enough to capture three successive frames in a short amount time. 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 IRLib_P03_RC5d.py and IRLib_P03_RC5s.py. The definition for the send method for this protocol is…

```
   def send(self, data, numBits=13,kHz=36):
```

Because the longest version of this protocol is 14 bits we do not need to make use of the address parameter. The default number of bits is 13 with an optional choice of 14 bits. The default frequency is 38 kHz with an option of 57 kHz for the 14 bit version. Here is an example code.

```
  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
```

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

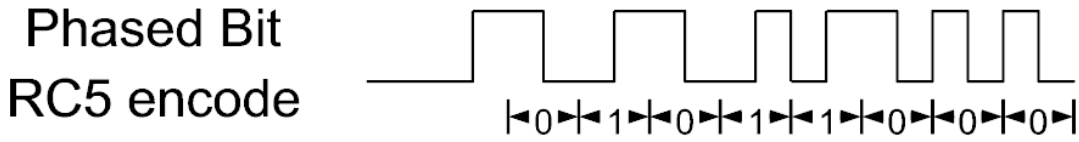
use phase encoding however the logic is reversed and that a space/mark indicates "0" and vice versa. Because these two groups of protocols share this common unusual feature they are derived from the IRdecodeRC and IRsendRC abstract classes which in turn are derived from the base classes.

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 bit 16 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 IRLibCP 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. In the original IRLib2 it would've been difficult to encode and handle data greater than 32 bits. Therefore 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. The original IRLib encodes only the 32 data bits. When converting the code to

Circuit Python we had to split the 32 bit value into 2 parts with the high order 4 bits as an address and the lower order 20 bits as the data. We could have gone ahead and encoded the OEM and trailer bits into the address field but we want to maintain complete compatibility with the original IRLib2. 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	0x00008000

The code for this protocol is in IRLib_P04_RC6d.py and IRLib_P04_RC6s.py. The definition for the send class for this protocol is…

```
def send(self, data, address,numBits=20):
```

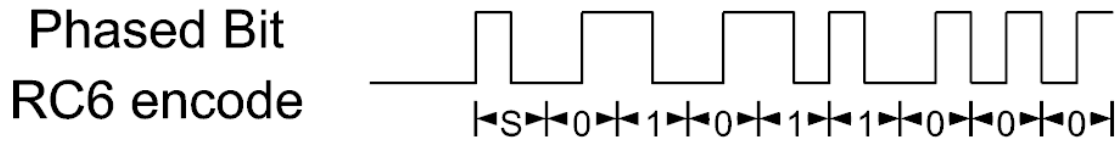
Because it gets complicated determining whether or not the second or third parameter should be considered address or numBits this routine always requires you to specify both data and address even though only the 32-bit version uses the address field. The default for numBits is 20 but you still always have to specify address. Naturally for any use other than 32-bit, that address field should always be zero. We could have reversed those two parameters but that makes multiprotocol sending routines more difficult to implement and in some ways violates our design that the first parameter is always data and if there is an address it should always be the second parameter. Here are some sample codes…

```
  mySender.send(0x12ab,0)              #send RC6 protocol
  mySender.send(0x12ab,0,20)           #send RC6 same as previous
  mySender.send(0xc12abc,0,24)         #send RC6-6-20
  mySender.send(0xc123abc,0,28)        #send RC6-6-20 Replay
protocol
  mySender.send(0x234abcd,0x1,32)      #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.

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, Spectrum

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 Spectrum 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 IRLib_P05_Panasonic_Oldd.py and IRLib_P05_Panasonic_Olds. The definition for the send class is…

```
  def send(self, data):
```

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 IRLib_P06_JVCd.py and IRLib_P06_JVC.py and the definition for the send routine is…

```
  def send(self, data, first=True):
```

The "first" flag 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…

```
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
```

Because a repeat signal does include the full 16 bits of data we did not implement this using a 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 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 IRLibCP cannot distinguish between NEC and NECx.

Because this is a 32-bit protocol the high order 4 bits is passed as the address and the lower order 28 bits is in the data field.

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 -1. 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. The code for this protocol is in IRLib_P07_NECxd.py and IRLib_P07_NECxs.py. The definition for the send class for this protocol is…

```
  def send(self, data, address):
```

Here are some examples…

```
mySender.send(0x234abcd,0x1)   #send NECx protocol
mySender.send(-1)              #send NECx 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 IRLib_P08_Samsung36d.py and IRLib_P08_Samsung36s.py and the definition is…

```
  def send(self, data, address):
```

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.

```
mySender.send(0x28d7,0x0400)   #send Blu-ray play
mySender.send(0xc837,0x0400)   #send Blu-ray stop
```

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/ Spectrum 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 -1.

Note that the timing of the ditto sequence for GICable is nearly identical to the ditto used by the NEC1 protocol. IRLibCP 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 IRLib_P09_GICabled.py and IRLib_P09_GICables.py and the definition for the send class for this protocol is…

```
  def send(self, data):
```

Here are some examples…

```
mySender.send(0x12ab)    #send GICable protocol
mySender.send(-1)        #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.

Because it would be difficult to create a multiprotocol send class with more than four parameters, we use 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. Also the nature of Circuit Python is such that lead out times in practical use will be much longer than specified because the code isn't fast enough to send multiple signals immediately after one another.

The code for this protocol is in IRLib_P10_DirecTVd.py and IRLib_P10_DirecTVs.py. The definition prototype for the send class for t's for his protocol is…

```
def send(self, data, first=True,kHz=38):
```

Here are some examples…

```
  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.

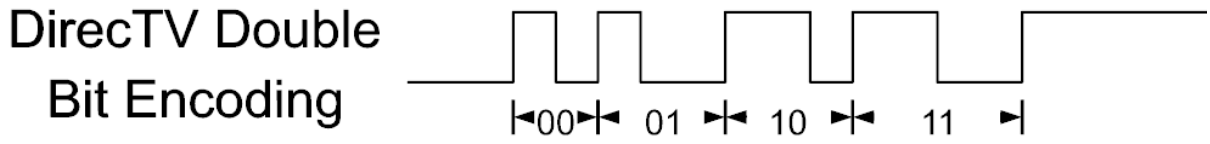
As mentioned earlier, this protocol uses an unusual encoding system that varies both the length of a mark and the length of a space in order to encode 2 bits for each mark/space pair. Marks and spaces are either 600 µs or 1200 µs long. The encoding goes as follows…

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 IRLib_P11_RCMMd.py and IRLib_P11_RCMMs.py. The definition prototype for the send class is…

```
    def send(self, data, address,bits=12):
```

Because there is a 32-bit version of this protocol you must always specify both data and address even if it is a 12 or 24 bit version of the protocol. Here are some examples…

```
mySender.send(0x12a,0,12)           #send 12 bits rcmm protocol
mySender.send(0x123abc,0,24)        #send 24 bits rcmm protocol
mySender.send(0x234abcd,0x1,32)     #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

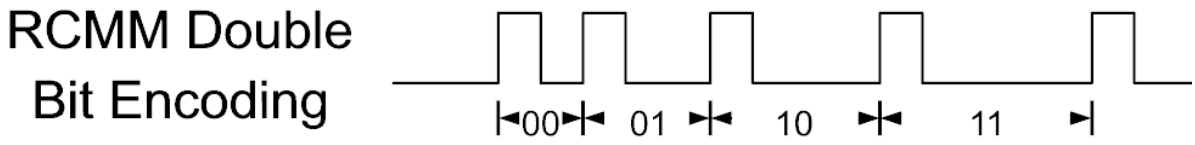
This presents a problem because normally IRLib decodes with a tolerance of 25% however if you take 25% of the 778 value you get 194.5 which is excessive. So instead of comparing values based on a percentage of the total, we implemented an alternative system that is a tolerance plus or minus some absolute value. That value can be found in IRLib_P11_RCMMd.py.

```
RCMM_TOLERANCE= 90
```

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.

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)+



1.5 Hardware Considerations

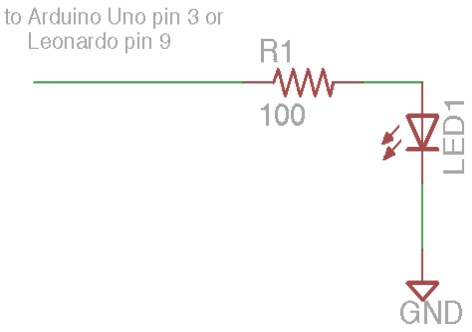
If you are using the Circuit Playground Express it has built-in IR transmitter and receiver circuitry. You simply use "board.REMOTEIN" for input and "board.REMOTEOUT" for output. But for other boards such as Feather M0 Express or Metro M0 Express you will have to add some extra hardware. Specifically an IR LED possibly with a driver circuit for output and some sort of infrared receiver for input purposes.

Many of the hardware issues you have to deal with in the original IRLib2 were because IRLib2 handle the hardware specific issues internally. However IRLibCP relies entirely on the pulseio module in Circuit Python and we do not need to deal with those issues. Only the PCI version of the receiver class is available. There is no timer driven receiver class so we do not need to concern ourselves with timers. We are unaware of what restrictions if any there are on pin choice for either input or output. We refer you to the documentation for pulseio to resolve any of those issues. Only the "Express" versions of Adafruit boards support pulseio even though other less powerful boards can run Circuit Python.

NOTE: the illustrations in this section were created for original IRLib2 on 8 bit AVR processors such as Arduino Uno and Leonardo. We have not bothered to update these illustrations for this version. You are not restricted to the pin numbers referenced in the illustrations. Also any reference to +5v should be considered +3.3v for the supported boards.

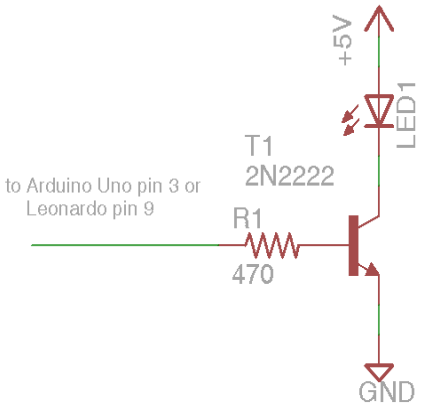
1.5.1. IR LED Driver Components and Schematics

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.



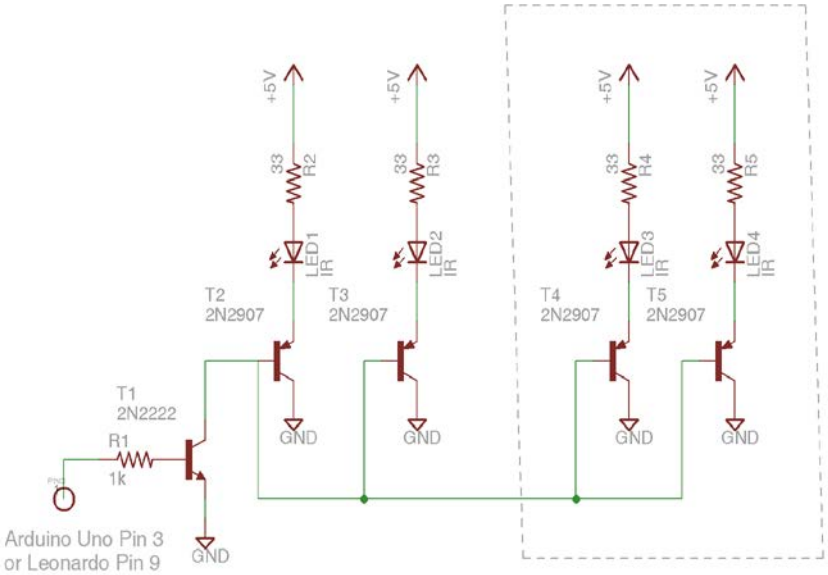
Make sure you get the polarity of the LED correct. The shorter of the two leads should connect to ground. The longer lead connects to the resistor which in turn connects to the Arduino.

The output pins of an Arduino cannot supply much current. 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 +3.3v 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.

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.



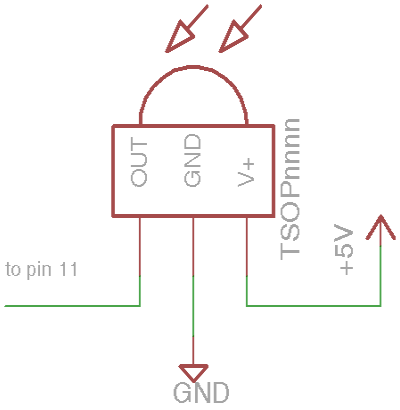
The circuit starts with an NPN transistor connected to the Arduino output pin via a 1K resistor. That NPN then drives up to four PNP transistors that drive the LEDs. Note that we have added a 33 ohm resistor to each LED to limit the current. This resistor was added because I designed the circuit to be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

1.5.2 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.

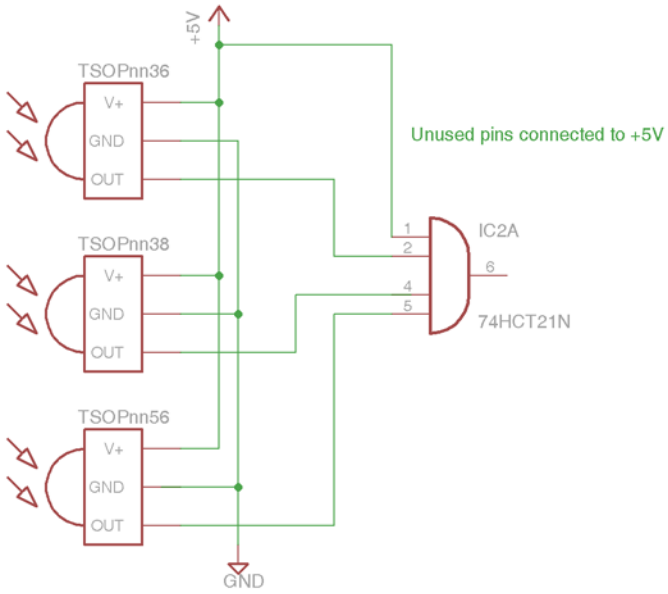


While the schematic is trivial, the challenge is finding the proper device for receiving. Although we deal with IR signals as though they are square waves with pulses varying from a couple of hundred microseconds up to a few thousand microseconds, in fact those pulses are actually modulated at a frequency 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. NOTE: This paragraph was originally written for use on 5v 8 bit Arduino systems. There should be a 3.3v compatible version of this chip but we have not taken the time to research or test the circuit at 3.3v.

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.

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 "examples". 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. We presume that you already have some familiarity with using Circuit Python and know how to load programs onto an Adafruit "Express" style control board. We also presume you have some sort of serial terminal program connected to the board so that you can see output from Circuit Python "print()" statements. NOTE: Circuit Python on boards that are not "Express" such as ESP 8266 do not support the pulseio and cannot be used with IRLibCP.

NOTE: the illustrations in this section were created for original IRLib2 on 8 bit AVR processors such as Arduino Uno, Mega and Leonardo. We have not bothered to update these illustrations for this version. Refer to pulseio documentation for which pins are available for input or output. Any references to +5v should be changed to +3.3v because all of the Adafruit Express boards use that voltage.

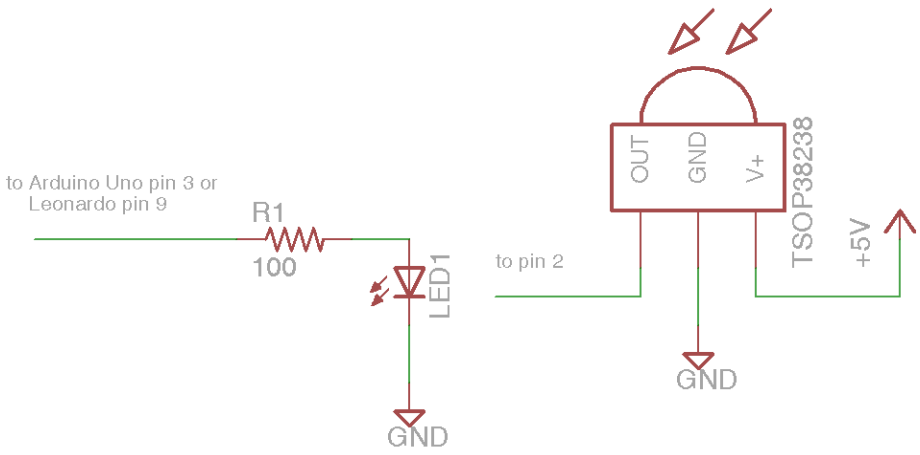
All of the modules in this package have been provided both as source code in filename.py and in compressed or precompiled filename.mpy versions. The source versions are in the "source" folder and the compile versions are in the "mpy" folder. We recommend to begin with copying all of the .mpy files onto your device by dragging and dropping into the virtual USB drive named CIRCUITPY. The example programs are provided only in filename.py form in the "examples" folder. You can copy these as needed onto the CIRCUITPY and once they are there you can either import them using the serial terminal or rename them as "code.py" to have them run automatically.

2.1 Setting up Your Hardware

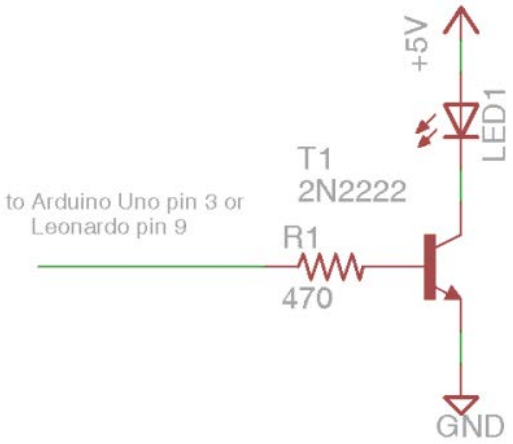
If you are using a Circuit Python Express than it already has built-in IR transmit and receive circuitry. You do not need to worry about attaching additional devices to your board. For other boards such as Feather M0 Express or Metro M0 Express to receive IR signals you will need a receiver device such as a TSOP4438 or something similar. In section 1.5.2 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.

Any input or output pin that can be supported with pulseio can be used. If you're not using Circuit Playground Express you will have to modify the sample code to specify your input and output pin numbers. For example during development we used a Feather M0 Express with input on "board.D5" and output on "board.D9".

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. Above is a schematic of what that looks like.

Note the polarity of the LED again. The long leads goes to +3.3v 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. Unfortunately due to memory limitations we cannot create a decoder that can decode all 11 supported protocols at once. We will begin with a very simple example that presumes you already know what protocol your remote uses. Then we will show you how to create a multiple protocol program that supports five or six protocols at once. If you do not know

which protocol your remote uses you could try both of those multiprotocol programs and hopefully determine which protocol you need.

To use any of these example programs you will need to drag-and-drop the necessary .mpy files onto your CIRCUITPY drive as well as the files for the programs themselves such as "decode_single.py" or whatever the sample program is. It doesn't hurt to have unused .mpy files on your CIRCUITPY drive as long as you have space for them. The only tight restrictions you will find will be how many of those modules you can actually import before you run out of memory.

All of our examples presume you are connected to your device using a serial terminal program. You have the option of either importing the example code such as:

```
import decode_single
```

… To invoke the program or rename the program file to "code.py" and it will automatically run for you.

2.2.1 The decode_single Example

This is the program decode_single.py which can be found in the examples folder.

```
# Sample program to decode one protocol in this case NEC
import board
import IRrecvPCI
import IRLib_P01_NECd

myReceiver=IRrecvPCI.IRrecvPCI(board.REMOTEIN)
myReceiver.enableIRIn()
myDecoder=IRLib_P01_NECd.IRdecodeNEC()
print("send a signal")
while True:
   while (not myReceiver.getResults()):
         pass
   if myDecoder.decode():
         print("success")
   else:
         print("failed")
   myDecoder.dumpResults(True)
   myReceiver.enableIRIn()
}
```

This starts with importing three modules. The "board" module is so that we can specify the input port for receiving IR signals. The second module "IRrecvPCI" contains the receiver code using pin chains interrupt. And finally the "IRLib_P01_NECd module is the decoder for the NEC protocol. There are currently 11 supported protocols. Each of the module names is of the form "IRLib_Pnn_NAMEd.py" where nn is the protocol number and "NAME" is the protocol name. The letter "d" at the end means this is the decoding module. There is also a different module which ends in a letter "s" for sending IR signals. Copying the decode_single.py file along with IRrecvPCI.mpy and IRLib_P01_NECd.mpy onto your CIRCUITPY drive. You will need a terminal program connected to the device so you can see the output from the "print"

statements. You can then either rename decode_single.py into code.py so it will automatically run or on the terminal type "import decode_single".

Here's how the code works. 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. Created instance of the receiver object called "myReceiver". The parameter passed in the constructor is the input pin for the IR receiver circuit. Here we are using "board.REMOTEIN" on the Circuit Python Express. However if you were using a Feather M0 Express or Metro M0 Express you would enter the pin number where you connected your IR receiver. It might be something like "board.D5" for example. We then called the myReceiver.enableIRIn() method. This initializes the receiver and tells it to begin listing for a signal.

We create a decoder object called "myDecoder". This decoder will analyze the received timing patterns that come from the receiver and determine if it matches the NEC protocol.

We next print a prompt telling you to press on your remote control. We then go into an infinite loop using "while True". Inside that loop we continuously call the myReceiver.getResults() function. It will return true when a frame of IR data has been successfully received. Although we sit in this loop waiting for the signal to go true, you could be doing other things and only occasionally polling getResults(). You do not need to set and wait for it to turn True like we do here. When it does return true, it stops listening for additional data. We then invoke the decoder and it will return true if it succeeds in matching the pattern to the NEC protocol or false otherwise.

Whether or not we succeed or fail, we will call the myDecoder.dumpResults(True) method. This will print out information about the timing signals received and will give you the decoded value if it was an NEC protocol. The parameter "True" says the we want verbose output. If it is "False" you just get a one line output of the decoded value.

After we have the code in the data printed the results we call the myReceiver.enableIRIn() method again to tell the receiver we are done processing the previous data and it should go listen for more. If you were to call enableIRIn before we were done decoding it, we would risk our data getting overwritten by the next frame of data.

Here is an example of some typical output from dumpResults…

```
success
Decoded NEC (1): Value:0x1a0f00f Adrs: 0x6 (32 bits)
Raw samples(67):  Head: m9041 s4535
 0:m536 s607     1:m562 s1707             2:m563 s1708   3:m562 s579
 4:m541 s598     5:m561 s580              6:m560 s582    7:m539
s1732
 8:m538 s1733    9:m566 s574             10:m566 s1705  11:m565 s576
12:m564 s578    13:m565 s575             14:m561 s579   15:m561 s582

16:m558 s1710   17:m560 s1714            18:m536 s1734  19:m566
s1705
20:m535 s605    21:m569 s571             22:m537 s604   23:m534 s608
24:m561 s580    25:m560 s579             26:m570 s571   27:m569 s571
```

```
28:m559 s1712   29:m568 s1703            30:m537 s1734  31:m569
s1703

32:m534
```

The output identifies this as NEC protocol which is protocol “1”. The received value in hexadecimal is 0x61a0f00f and is 32 bits long. Because Circuit Python cannot handle 32 bit values, the return value has been split into the higher order 4 bits in the address field and the low order 28 bits in the value field. The NEC protocol is always 32 bits long however other protocols may use other bit length and may have different varieties. For example Sony protocol has 8, 15, or 20 bit varieties. In this case the stream contained 67 intervals which is the number of raw samples. It then dumps out all of the values from the decode buffer which is IRrecvPCI.decodeBuffer. When a signal is on we call it a "mark". The empty interval between signals we call a "space". The first 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.

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 NEC (1): Value:0x1a0f00f Adrs: 0x6 (32 bits)
```

For most applications all you need is that 32 bit hexadecimal number because it uniquely identifies which button was pushed on the remote. You could then use those values for a variety of purposes. For example you were controlling a robot. If you detected the 32 bit code for pressing the right arrow button you would make the robot turn right, left arrow would make it turn left, up arrow for forward and down arrow for backwards. You really don't need to know the details of the binary sequence. You just need to use it to identify which button was pushed on the remote. Furthermore we can use this same 32 bit number in our sending routines to re-create that same signal. We do not need to store all 67 timing values to re-create the signal.

If you're wondering how we get that 32 bit number let's take a closer look at the verbose dump. Most protocols begin with a header which consists of an abnormally long mark and space. In this case we had a mark of 9041 and a space of 4535. We happen to know that the NEC protocol specifies a header of 9000, 4500 so these values are within the margin of error. 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 536, 607 followed by bit 1 which has a mark of 562 and a space of 1707. If you look at each bit, some have marks and spaces that are nearly equal to one another while others have spaces which are approximately three times the normal value but marks that are still in the normal range. We interpret each of those long spaces 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 0x61a0f00f which is what we reported. The NEC protocol actually specifies 564, 564 as a "0" and 564, 1692 (which is 3*564) as the pattern for a "1". We use a margin of error of 25% and these values are well within that range.

This system of constant length marks and variable length spaces is typical of most protocols. However for example Sony does the opposite with variable length marks and fix length spaces. 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.

Some protocols behave differently depending on whether you press the same button repeatedly or if you hold the button down and it automatically repeats. Specifically NEC protocol transmits a special sequence as a repeat code. Unfortunately the repeat code for NEC does not tell you what it is that is repeated. Furthermore IRLibCP does not store previous codes to tell you what you need to repeat. So your application program must deal with repeat sequences on its own. It's up to you to decide whether or not to ignore a repeat or to make use of it. We decode the special NEC repeat sequence called a "ditto" as a value of "-1". Here is a dump of what an NEC ditto sequence looks like.

```
success
Decoded NEC (1): Value:-0x1 Adrs: 0x0 (0 bits)
Raw samples(3):  Head: m8988 s2295
 0:m514
```

Some protocols use repeat flags or toggle bits or other systems to designate a repeat while other protocols have no way to distinguish between repeated button presses or a held button. See the reference section under the specifics for each protocol to determine how it handles repeat sequences.

2.2.2 Decoding Multiple Protocols

The previous example only works if you know that your particular remote is using NEC protocol. While most applications will only need to use one protocol, some point prior to that you need to identify which protocol you want. Here is an example program from the examples folder "decode_multiple.py" that combines decoders for NEC, Sony, and RC5 protocols.

```
# Sample program to decode multiple protocols.
# In this case NEC, Sony and RC5
import board
import IRLibDecodeBase
import IRLib_P01_NECd
import IRLib_P02_Sonyd
import IRLib_P03_RC5d
import IRrecvPCI

class MyDecodeClass(IRLibDecodeBase.IRLibDecodeBase):
   def __init__(self):
         IRLibDecodeBase.IRLibDecodeBase.__init__(self)
   def  decode(self):
         if IRLib_P01_NECd.IRdecodeNEC.decode(self):
               return True
         if IRLib_P02_Sonyd.IRdecodeSony.decode(self):
               return True
         elif IRLib_P03_RC5d.IRdecodeRC5.decode(self):
               return True
         else:
```

```
              return False

myDecoder=MyDecodeClass()

myReceiver=IRrecvPCI.IRrecvPCI(board.REMOTEIN)
myReceiver.enableIRIn()
print("send a signal")
while True:
   while (not myReceiver.getResults()):
         pass
   if myDecoder.decode():
         print("success")
   else:
         print("failed")
   myDecoder.dumpResults(True)
   myReceiver.enableIRIn()
```

In addition to importing "board" and "IRrecvPCI" we also need to import the base decoder module as well as the individual modules to decode the protocols. Be sure to drag-and-drop the .mpy files for all of these modules onto your CIRCUITPY drive along with decode_multiple.py. You can rename the program as code.py to automatically run or you can type "import decode_multiple" to initiate it. You need to be connected to a terminal program to see the serial output.

After importing all the modules, we then create a custom class called MyDecodeClass. It initializes the base code but does not need to initialize each of the independent protocols because they themselves do not need any initialization other than the base coat. A custom decode routine individually calls the decoders for each of the protocols and if any of them succeed it will return true. If all three fail it returns false. The rest of the program is identical to our previous decode sample. Here is some sample output showing how the Sony and RC5 protocols are decoded.

```
Adafruit CircuitPython 0.10.1 on 2017-05-20; Adafruit
CircuitPlayground Express with samd21g18
>>> import decode_multiple
send a signal
success
Decoded NEC (1): Value:0x1a0f00f Adrs: 0x6 (32 bits)
Raw samples(67):  Head: m9018 s4558
 0:m562 s577     1:m562 s1710             2:m569 s1702   3:m567 s574
 4:m566 s573     5:m566 s574              6:m567 s574    7:m565
s1707
 8:m562 s1709    9:m541 s599             10:m541 s1730  11:m539 s601
12:m539 s603    13:m566 s575             14:m565 s575   15:m564 s579

16:m561 s1706   17:m563 s1714            18:m536 s1736  19:m534
s1731
20:m538 s602    21:m537 s609             22:m561 s574   23:m538 s605
24:m562 s578    25:m562 s578             26:m561 s579   27:m561 s580
```

```
28:m559 s1712   29:m568 s1704            30:m536 s1734  31:m565
s1707

32:m563
success
Decoded Sony (2): Value:0x540c Adrs: 0x0 (15 bits)
Raw samples(31):  Head: m2415 s601
 0:m1237 s576    1:m612 s602             2:m1219 s594   3:m611 s603
 4:m1185 s631    5:m592 s619             6:m615 s598    7:m590 s622
 8:m616 s596     9:m613 s599            10:m619 s601   11:m1207
s600
12:m1218 s597   13:m611 s604            14:m585
success
Decoded RC5 (3): Value:0xbcd Adrs: 0x0 (13 bits)
Raw samples(19):  Head: m1775 s1834
 0:m1782 s1847   1:m845 s961             2:m867 s936    3:m862 s941
 4:m1751 s970    5:m868 s1851            6:m841 s963    7:m1758
s1851
 8:m862
```

Note that the Sony protocol specifies that signals are sent three times consecutively however IRLibCP isn't fast enough to detect rapidly sent consecutive signals so it is only able to decode the first one of three that were actually sent by the remote. Note that Sony uses variable length marks and fixed length spaces which is the opposite of NEC. It is the only known protocol to do so. The RC5 protocol uses a special phase encoded system that is unlike either NEC or Sony. See the reference section for more details on how RC5 and RC6 protocols use phase encoding. Note also that because Sony and RC5 are shorter than 28 bits we do not need to make use of the Address field.

Here is sample output with the dumpResults(False) verbose setting off.

```
Adafruit CircuitPython 0.10.1 on 2017-05-20; Adafruit
CircuitPlayground Express with samd21g18
>>> import decode_multiple
send a signal
success
Decoded NEC (1): Value:0x1a0f00f Adrs: 0x6 (32 bits)
success
Decoded Sony (2): Value:0x540c Adrs: 0x0 (15 bits)
success
Decoded RC5 (3): Value:0xbcd Adrs: 0x0 (13 bits)
```

Unfortunately memory restrictions will not allow us to make a decoder which will handle all 11 supported protocols at once. We have provided in the examples folder 2 sample programs named "decode1-6.py" and "decode7-11.py" which will decode protocols 1 to 6 or 7 to 11 respectively. Make sure you have all of the needed .mpy files on your CIRCUITPY drive when running these programs. You can use these programs to determine what protocol your particular remote uses. If unfortunately it does not use one of our 11 supported protocols there are ways that you can capture the raw data and retransmit it if necessary. We have not yet developed modules to support this but we will in upcoming versions. In the original IRLib2

documentation there is an extensive section on how to implement new protocols that you might want to check out.

2.2.3 Additional Decoding Examples

This is a preliminary beta release. We will have additional decoding examples in future releases.

2.3 How to Send IR Signals

In the previous section we showed you how to receive and decode IR 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.

If you are using Circuit Playground Express then you already have a built-in IR transmitter. On other platforms see the beginning of this tutorial section where we showed you how to create an IR transmitter circuit. 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 examples/send_single.py.

```
# Sample program for sending one value of NEC protocol.
import board
import IRLib_P01_NECs

mySend=IRLib_P01_NECs.IRsendNEC(board.REMOTEOUT)
Address=0x6
Data=0x1a0708f
print("Sending NEC code with address={}, and
data={}".format(hex(Address),hex(Data)))
mySend.send(Data, Address)
```

We import the "board" and "IRLib_P01_NECs" modules. Note that this module name ends in "s" for sending rather than "d" for decoding as in the previous examples. We create a sending object and pass it the pin number for our IR transmitter. Here we have used the board.REMOTEOUT built-in transmitter for the Circuit Playground Express however for other boards you can use any PWM capable pin. For example on Feather M0 Express we used board.D9 to which we had connected an IR LED and driver circuit.

Because NEC protocol uses 32 bits we had to put the high order 4 bits in the Address and lower order 28 bits in the Data fields. This particular sample pattern is the mute button on a TV I own that uses NEC protocol.

Execute the program from the terminal by typing "import send_single" and it will transmit the proper IR sequence. While it's not a very useful program it illustrates how to use sending. We could write additional programs that would for example send a particular code whenever

you pressed one of the buttons on the Circuit Playground Express or perhaps using capacitive touch on one of the input pads to send different codes. We will develop further examples later on.

We have been able to create an example that will send patterns from all 11 supported protocols in the same program without running out of memory. Check out a sample program called "examples/send_patterns.py". It sends various patterns of data using all of the variations of all 11 supported protocols. Near the bottom of the program there is a line:

```
Test=IRLibProtocols.NEC
```

You can edit this line for any of the supported protocols or send it to -1 and it will cycle through all of the options automatically. If you have another Arduino or Express board running IRLibCP using a decoder program you can see the results of what is transmitted. Note these are simply bit patterns designed to make it easy to see what is being transmitted. They do not necessarily represent valid codes for those particular protocols. This program was used during development and testing of the protocols to make sure transmission and reception and decoding are occurring properly.

We will have more useful examples in later releases of this package.

2.4 Sending and Receiving in the Same Program

Theoretically it should be possible to both send and receive in the same program. This capability is in the original IRLib2 but has not yet been tested in IRLibCP. Watch this space for future example programs illustrating illness feature.

3. Implementing New Protocols

Please refer to the documentation from original IRLib2 written in C++ for how to implement additional protocols.

Appendix A – Differences between IRLibCP and original IRLib2 written in C++

IRLibCP only uses IRrecvPCI receiver class. There is no IRrecv 50 µs interrupt driven class or IRrecvLoop receiver class.

IRLibCP does not include any frequency detection capability such as IRfrequency

IRLibCP does not implement any auto resume features. Thus if multiple frames of data come rapidly it will only be able to detect and decode the first frame. Other data will be lost.

IRLibCP does not yet implement protocol 12 for mouse and keyboard manipulation.

Because Circuit Python does not implement 32-bit unsigned integers, protocols such as NEC and others that have greater than 28 bits will use data split into 2 fields. The higher order 4 bits will go in the "address" field and the lower order 28 bits go in the "value" field. The Samsung36 continues to split its data into a 12 bit address and 20 bit data field. The "ditto" repeat code used by NEC, NECx, and G.I.Cable protocols is represented by "-1" rather than 0xffffffff.

There is a known issue with pulseio and PWM output in that when it is initialized or de-initialized there is a brief glitch of signal sent. We have adjusted timing so that this glitch typically does not interfere with subsequent decoding of the signals but it does introduce noise into the system and limits how rapidly you can send successive signals.

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

I began computer programming writing and BASIC when I was in high school in the early 1970s. I went to college and wrote Fortran and Pascal and God help me even a little COBOL as well as other programming language that don't exist anymore such as Algol and PL/1. For the last 25 years I've written nothing but C and C++ with minor dabbling's in JavaScript and PHP. But converting IRLib2 into Python is only the second Python program I've ever written. I'm sure experienced Python programmers will cringe when a see my code. Please be patient with me. I'm still learning. If you want to give me some constructive tips please send them to me at cy_borg5@cyborg5.com.