Infrared Transmit Validation for the Raspberry Pi

Rob Simon

Jim Ladd

August 11, 2020

Contents

Introduction

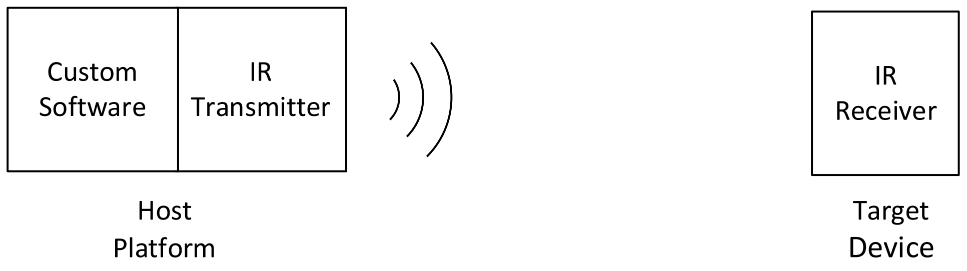
Infrared (IR) technology has been used to wirelessly control consumer electronics since 1980 [1]. Sometime later, computers began emulating the IR signals so the control could be done programmatically. While this approach allows computer interfacing where none was intended, the issue is that the signal is unidirectional. There is usually no way to ensure the target device received and successfully processed the IR signal. A project at SOFWERX relied on IR to remotely control a target device. In theory, the approach was attractive but, in practice, it was unreliable. This paper describes an approach to increase the stability of this type of computer interface by validating the IR signal transmission.

Problem Domain

The goal of the project was to programmatically control a target device. The only interface to this device is a hand held remote control similar to the ones used for consumer electronics. The remote control relied on a unidirectional infrared signal. There is no feedback loop to validate that the signal was received and processed correctly by the target device.

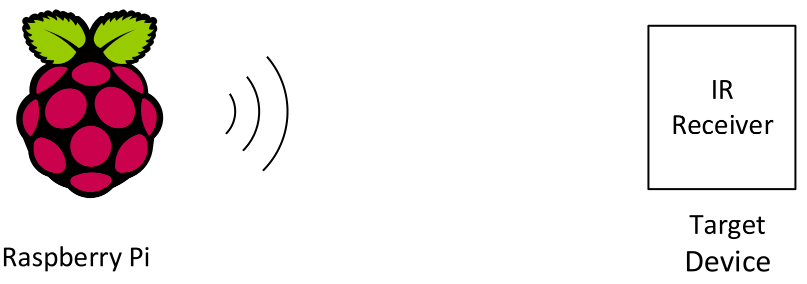
Since the target device was already selected when we inherited the project, we moved ahead with the IR-based communications with the hope that we could overcome the challenges of this technology. From past experience, IR-based integrations can be difficult to achieve acceptable reliability.

The high level diagram of the problem space is shown below:

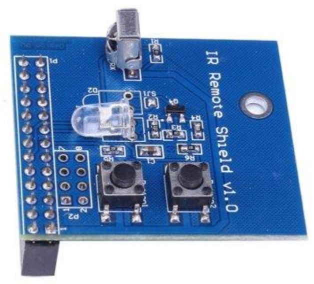


Solution Domain

The path from the problem space to the solution space is littered with more decisions, some solid and some questionable. The major decision was to use the Raspberry Pi (RPi) single board computer as the host platform. The RPi is a great selection for these types of projects. It is inexpensive, expandable, and well known. Our hardware architecture is slowly emerging as shown in the diagram below.

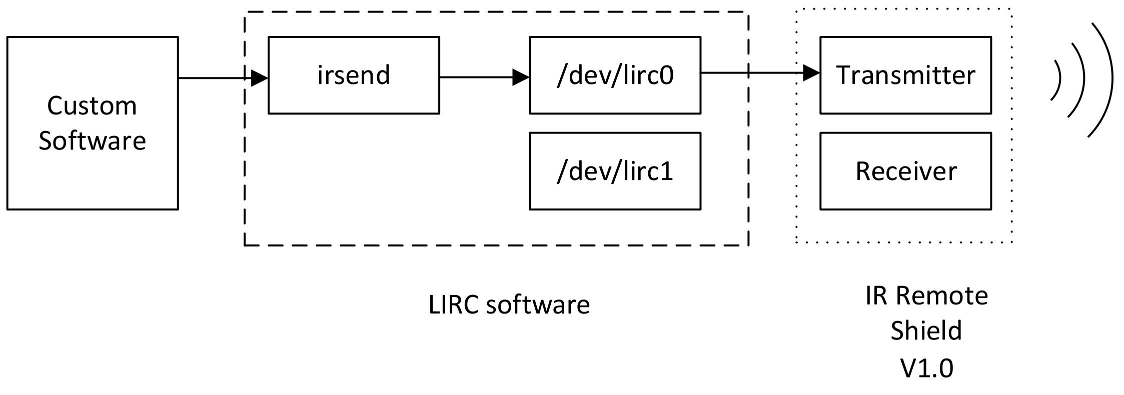


The RPi boards do not provide an IR transmitter as a built-in feature. There are add-on or expansion boards available. These boards use the RPi's General Purpose Input/Output (GPIO) interface. The IR board selected from the initial effort is shown below. There is no vendor designation, only the label of "IR Remote Shield v1.0". This board supports both IR transmit and IR receive operations. It is readily available and extremely inexpensive.



The "IR Remote Shield v1.0" product does not include drivers or interface software. Fortunately, there is an open source effort that provides the low level processing required to send and receive IR signals. LIRC or Linux Infrared Remote Control allows the IR signals to pass through the RPi. LIRC was first established in 1999 and has several distribution packages including one for the Raspberry Pi platform [2].

The LIRC package provides a set of executable programs that are used for sending and receiving the IR signals. The most common program for sending is the "irsend" executable. This program is invoked through the python subprocess library. A diagram of the major components of the LIRC are show below.



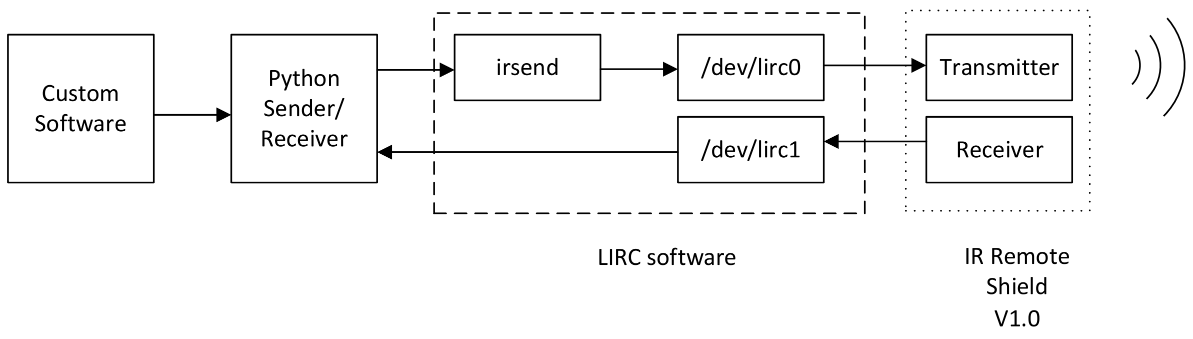
An example of an irsend invocation in python is shown below:

```
subprocess.call(["irsend", "SEND_ONCE", "lircd.conf", "KEY_MODE"])
```

This line of code invokes the "irsend" executable with three arguments. The "SEND_ONCE" value denotes that the data should be sent only one time. The "lircd.conf" is the name of the device in a configuration file. The irsend program will read the configuration for this name and use that information to drive the structure of the IR waveform. The "KEY_MODE" represents a button on the remote control. The irsend program looks up this name in the configuration and retrieves the numerical value. This value is the hexadecimal value of the pulses and spaces for that specific button.

The target device can be controlled though the sequence of different button presses. The lengths of these sequences range from a single button press up to a total of 13 presses. If all went well, we would have finished the coding of the button sequences and be done with the project. However, during the execution of the longer button sequences, the target device would enter an error state. We even had the custom software on the RPi drive multiple target devices and they consistently error at the exact same step in the process.

After much investigation and testing, we narrowed the culprit of the issue to either the LIRC software or the IR expansion board. Since the IR interface is unidirectional, the custom software had no mechanism to confirm or validate the information that was transmitted. Reviewing the IR expansion board's layout, we discovered that the transmit circuit was independent of the receive circuit. In theory, custom software could 1) begin receiving the inbound IR signal, 2) in a separate thread, transmit the button press data, and 3) compare the received information to the sent data. This software would create a simple feedback loop and, hopefully, catch signals that were not being transmitted correctly. The software architecture for this feature is shown below.



To understand how the IR works at a more granular level, please refer to the *IR Remote Control Primer* [3] article. The material presented is very relevant to this project.

The LIRC package provides a configuration mechanism for tailoring the IR signaling to different protocols, vendors, and devices. One or more device definitions can be specified in the configuration file. In this project, the name of the file is, confusingly, the same as the device name used within the file. The path to the configuration file in a standard installation is:

```
/etc/lirc/lircd.conf
```

This file contains the details of the IR protocol for the remote control and device. The protocol details along with definitions of a few “buttons” are shown below.

```
begin remote

  name            lircd.conf
  bits            16
  flags           SPACE_ENC|CONST_LENGTH
  eps             30
  aeps           100

  header          9139  4495
  one             622  1631
  zero            622   517
  ptrail          622
  repeat          9141  2213
  pre_data_bits   16
  pre_data        0x33B8
  gap             108580
  toggle_bit_mask 0x0

      begin codes
          KEY_POWER                0x807F
```

```
                KEY_1                        0x8877
                KEY_2                        0x48B7
                KEY_3                        0xC837
                KEY_4                        0x28D7
                KEY_5                        0xA857
                KEY_6                        0x6897
                KEY_7                        0xE817
            .
            .
            .
                KEY_DIM                      0x58A7
                KEY_A4                       0x14EB
                KEY_C-F                      0xD827
            end codes
      end remote
```

To further understand how the configuration data is actually used, we leverage the output of one of the LIRC programs called "mode2". The mode2 software is a command line executable that outputs the pulse/space values when an IR signal is received. These values signify something in the configuration file. Once the "header" is received, the following pulse/space pairs should denote either a "one" or a "zero". In turn, every four binary values represent a hexadecimal value. When 32 binary values (or 8 hexadecimal values) are received, the combined characters or the numerical value can be mapped to a button or key such as KEY_POWER or KEY_MODE. The following is the beginning of an actual output of the mode2 program. The first column denotes if the data presents a space or a pulse. The second column is the value of the space/pulse. The third column (or #) translate to a value in the configuration file such as "header", "one", "zero", etc. The final column (or ##) is the hexadecimal value of the last four binary numbers received.

```
space 16777215
pulse 9131
space 4519  #header
pulse 594
space 519   #zero
pulse 623
space 516   #zero
pulse 619
space 1658  #one
pulse 594
space 1633  #one      ##3
pulse 619
space 519   #zero
pulse 619
space 519   #zero
pulse 621
space 1657  #one
pulse 596
space 1631  #one      ##3
```

```
pulse 619
space 1661   #one
pulse 592
space 544    #zero
pulse 595
space 1656   #one
pulse 595
space 1656   #one        ##B
```

Now that the decoding of space/pulse pairs is known, the design of how to codify this process must be performed. One approach is to use the mode2 as a command line program similar to that of the irsend program. The key difference is that with the mode2, the output of the program must be captured. Also, the mode2 program must be terminated when all of the space/pulse data is received. While this could be done at the operating system level, a design that was more native to python was desired. We were not alone in this train of thought. A posting on a GitHub project in 2016 suggested a python version of mode2 be developed and included a snippet of code as a suggestion [4]. We decided to use this snippet as the beginning of our custom sender/receiver software.

Two python files were developed in our attempt to validate the IR transmission. The irsend2.py file is responsible for creating a thread that reads the LIRC device (i.e. /dev/lirc1) used for receiving inbound signals. Once the thread is created, the input "key" is sent via the information in the specified configuration file. Once the listening thread has received all of the inbound data, it joins with the main thread with the "pulse/space" data similar to the mode2 program.

The second python file, decode.py, is responsible for translating the "pulse/space" data into one of the keys specified in the configuration file. This value is then compared to the key that was sent. If the pair match, a Boolean true is returned to the calling function. If the values do not match, a Boolean value of false is returned. In the event of a false value, we know that an error occurred at some point in the IR signal send/receive cycle. We can then resend the key that produced the error which, theoretically, should prevent the clock from receiving a faulty signal.

Looking Back and Moving Forward

The good news is that the irsend2 software, even in the rough proof-of-concept state, performed as expected. It was able to detect when the IR signal did not match the intended value. The bad news is that we suspect the hardware is at fault. Even with different expansion boards (but of the same model), the signals received did not match the intended values. The only option is to replace this expansion board either from a different vendor or a custom board.

Although the LIRC software has been around for many years, the last update to the code was done three years ago in September of 2017. Such inactivity is a concern. For example, in April of 2019, a new kernel for the Raspberry Pi was released and caused issues with several working installations of LIRC. Also, some of the programs in LIRC have simply stopped working with the new version of the kernel. A better approach would be a lightweight python-based version of the irsend and mode2 software.

Where to Find

The link to the Git repository is:

https://github.com/sofwerx/irsend2

References

[1] S. Beschloss, "Object of Interest: Remote Control," New Yorker, 22 11 2013. [Online]. Available: https://www.newyorker.com/tech/annals-of-technology/object-of-interest-remote-control?utm_source=onsite-share&utm_medium=email&utm_campaign=onsite-share&utm_brand=the-new-yorker.

[2] "LIRC Home Page," [Online]. Available: https://www.lirc.org/. [Accessed 10 08 2020].

[3] Phidgets, "IR Remote Control Primer," [Online]. Available: https://www.phidgets.com/docs/IR_Remote_Control_Primer. [Accessed 10 08 2020].

[4] "LIRC mode2 support #13," [Online]. Available: https://github.com/tompreston/python-lirc/issues/13. [Accessed 10 08 2020].