

LINUX APPLIANCE DESIGN

A REAL WORLD GUIDE TO
BUILDING LINUX APPLIANCES

BOB SMITH, JOHN HARDIN, GRAHAM PHILLIPS & BILL PIERCE



12

INFRARED REMOTE CONTROL



Remotes have become our most natural means of controlling certain appliances, and if you're like us, you'll occasionally spend several minutes trying to locate a remote control rather than trying to engage some device's obscure front panel. In the future, we expect to see more appliances forfeiting the front panel altogether in favor of the keypad of a simple remote. Here's a rule of thumb: If you are building an appliance that might be enjoyed by people while they are sitting on a couch, include a remote control.

In this chapter we will cover the following topics:

- Communicating with infrared light
- Hardware for remote control receivers
- Installing and configuring LIRC for the Laddie appliance

Communicating with Infrared Light

The infrared (IR) light used by remote controls has a wavelength close to, but greater than, visible light. Because it's close to visible light, it travels in straight lines and reflects off of surfaces, but it doesn't go through opaque objects. This limits the applications for which IR is useful. It's good for controlling a set-top box, but not so good for opening a garage door, if there's a solid wall between the transmitter and the receiver.

For the most part, the fact that IR is invisible is a good thing. It may be harder to debug something that's invisible, but when you're watching the late-night movie on your new Linux-based DVR, it's nice to know you can turn down the volume without shining visible light onto the screen.

NOTE *If you wish you could see the light from a remote control, perhaps to verify that a unit isn't broken, you can look at it using a cell phone camera. These cameras are sensitive to infrared, and on cameras we've experimented with, they display this "color" as bright white.*

Protocols for Encoding Remote Control Commands

In order to transfer information, a remote control transmitter and its receiver must use the same standard or *protocol* for encoding commands. A remote control protocol specifies the following three things:

- How it represents ones and zeros
- How these ones and zeros are combined or *framed* to form messages
- How these different messages are to be interpreted

Companies that build remote-controlled devices don't generally publish their protocols, but it's not hard to reverse engineer the basic commands, and the Internet has plenty of information from people who have done just that. As an example, we'll consider a protocol Sony has used for some of its televisions. If you do some research on the Internet, you might see this protocol referred to as the Sony Integrated Remote Control System (SIRCS) protocol. We chose to use this protocol for the Laddie appliance because the protocol is easy to understand. It is also easy to produce: We purchased a universal remote (RCA RCU410) and programmed it to "Sony TV" (Code 002).

This Sony TV protocol uses pulse-coded data encoding. With this encoding, a bit is represented as a variable-width *pulse*, or presence of light, followed by a constant-width *space*, or absence of light. Based on our own timing measurements with our handheld remote, a zero has a 650-microsecond pulse, a one has a 1,300-microsecond pulse, and each is followed by a 500-microsecond space. These encodings are illustrated in Figure 12-1.

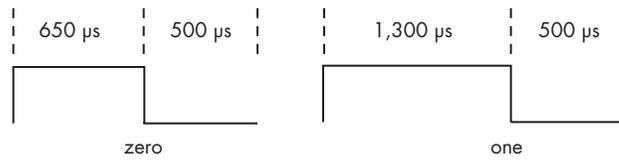


Figure 12-1: Zeros and ones in the Sony TV protocol

A frame in the Sony TV protocol (again, as measured for our particular remote) begins with a header consisting of a single 2,500-microsecond pulse followed by a 500-microsecond space. A seven-bit command immediately follows the header, and a five-bit address follows the command. Both the command and the address are transmitted with the least-significant bit (lsb) first. Figure 12-2 shows the waveform for the TV/Volume- command.

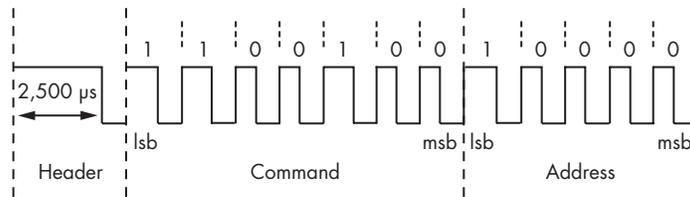


Figure 12-2: The TV/Volume- command in the Sony TV protocol

The address specifies a device (in our case, always 0x01 for TV), and the command specifies the input to that device. Table 12-1 lists some of the command codes for the Sony TV protocol.

Table 12-1: Device Addresses and Command Codes in the Sony TV Protocol

Command code	Command
0x00–0x09	1–9, 0
0x10	Channel+
0x11	Channel-
0x12	Volume+
0x13	Volume-
0x15	Power

For the Laddie appliance, we use the commands Channel+, Channel-, Volume+, Volume-, and Power. Of course, the Laddie appliance doesn't really have channels or volume levels; we have simply chosen these as convenient inputs for the framebuffer menu navigation.

There are many other remote control protocols, and each represents a set of engineering trade-offs. For example, Panasonic's REC-80 protocol uses constant-width pulses and encodes zeros and ones by the length of the space

between pulses. This approach can lead to longer battery life because it minimizes the amount of time the remote control spends emitting light. Other protocols save battery life by transmitting a short “Repeat” command when a button is held down, rather than repeatedly transmitting the entire command, as the Sony TV protocol does.

NOTE To learn about some of these other protocols, visit <http://sbprojects.com/knowledge/ir/ir.htm>.

Reducing Interference by Modulating the Infrared Signal

So far, we’ve treated infrared pulses as if they corresponded to steady beams of light. But consider Figure 12-2, and suppose that some flickering light bulb were to generate pulses of IR that overlapped some of the spaces in a message. Clearly, such interference could make it impossible for a receiver to correctly interpret the message. The solution is to modulate the pulses of IR light. In a modulated pulse, the IR light is actually turning on and off at a fixed frequency, typically between 30 and 60 kHz (kilohertz). Like picking out a voice in a crowded room, the receiver can use this frequency as a signature to discriminate the intended signal from the background noise. Because of this modulation, the pulse-coded zeros and ones of the Sony TV protocol are more accurately depicted as in Figure 12-3.



Figure 12-3: Modulated pulses in the Sony TV protocol

We measured the modulation frequency for our remote as roughly 40 kHz. Thus, for our remote, the zero “pulse” in Figure 12-3 actually consists of $40,000 * 0.000650 = 26$ much shorter pulses.

Controlling an Appliance with Infrared Light

Now that we’ve seen how infrared light can convey information, we can design a system for implementing remote control of an application. Figure 12-4 illustrates such a system.

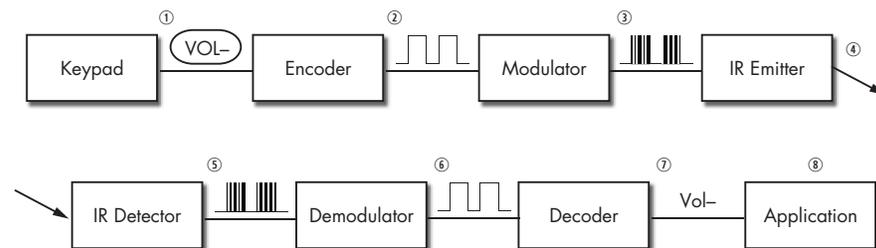


Figure 12-4: A complete remote control system

On the transmitting side, a typical handheld remote control performs the following steps:

1. It scans a *keypad*.
2. It *encodes* the input as a waveform according to some protocol.
3. It *modulates* this encoding with a frequency between 30 and 60 kHz.
4. It *emits* a corresponding pattern of infrared light.

On the receiving side, a typical IR receiver performs the following stages:

5. An infrared *detector* converts the input signal to a voltage waveform.
6. A *demodulator* removes the 30 to 60 kHz modulation.
7. A *decoder* analyzes this waveform and determines the corresponding command.
8. An *application* responds appropriately to this input.

Hardware for Remote Control Receivers

In this section we'll focus on IR receiver hardware. Designing remote control transmitters is beyond the scope of this chapter, but our recommendation is to take the same approach we took with the Laddie appliance: Use an off-the-shelf, universal remote.

Detecting and Demodulating the IR Signal

The two stages of detecting and demodulating an IR input can be handled by a single, commercial, off-the-shelf component. Figure 12-5 shows three examples of this part, all produced by Sharp Microelectronics.



Figure 12-5: Infrared detector/demodulators

For the Laddie appliance, we chose the 40 kHz unit from Sharp's GPIUV70QS series (pictured on the right in Figure 12-5). Also popular is the TSOP17XX series from Vishay. Each of these parts is easy to use with perfboard or solderless breadboard prototyping materials. Each requires a 5V power supply and provides an output corresponding to the demodulated IR waveform (see step 6 in Figure 12-4). When evaluating a particular device, make sure the demodulation frequency is appropriate for your chosen protocol. For low-power applications, you'll also want to compare the power requirements for different devices.

There isn't a standard nomenclature for these devices. When you're searching for information, expect to see names like "IR Remote Receiver," "Photo Module for Remote Control," or "Infrared Detecting Unit for Remote Control." In the remainder of this chapter, we'll refer to them as *infrared detector/demodulators* to emphasize the two functions they provide.

BUILDING A SIMPLE IR DETECTOR

If you have access to an oscilloscope and would like to see a remote control signal *before* it is demodulated, you can build the simple IR detector circuit depicted in Figure 12-6. All you need is a power supply (a 9V battery is fine), an appropriately valued resistor, and a phototransistor.

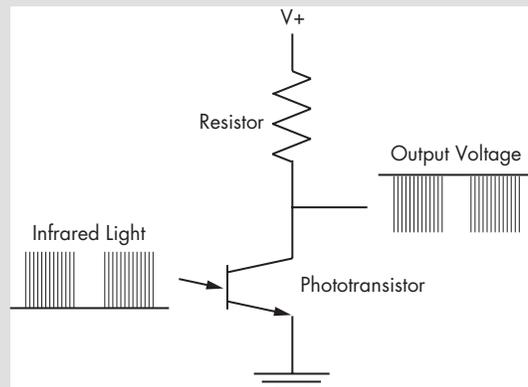


Figure 12-6: A simple IR detector

You'll want to choose the resistor so that the output voltage is about half of the supply voltage when the remote control is off. When we tried this, we used a 100K resistor, and we held the remote control very close to the detector. We had good results both with a Radio Shack infrared phototransistor (catalog number 276-142) and when using another, unidentified, phototransistor we happened to have lying around. The output will be weak, but it should be adequate for an oscilloscope. By the way, this experiment is also useful if you need to determine the modulation frequency for an unfamiliar protocol.

Decoding Remote Control Waveforms

In the previous section we introduced off-the-shelf devices that will respond to a remote control IR signal and produce a demodulated voltage waveform. Before designing a circuit to take advantage of these devices, we need to decide how we will decode that waveform. Decoding can be broken into two steps: *measuring* the timing of the pulses and spaces that comprise a waveform and *interpreting* this sequence of timings to identify the intended message. We have three options for designing a decoder, depending on which of these two tasks we assign to external hardware and which we assign to the appliance's processor. We'll briefly discuss these three options before describing the approach we took for the Laddie appliance.

Measuring and Interpreting in External Hardware

It's possible to build receiver hardware that performs all of the decoding tasks: measuring the waveform, determining the corresponding command, and then transmitting that command as one or more serial bytes to the appliance's processor. Figure 12-7 illustrates this approach. Here, the receiver has recognized the waveform for the Volume- command and has produced the single ASCII character *D* for *down*.



Figure 12-7: Decoding waveforms in external hardware

The website <http://linuxtoys.org/xirrc/xirrc.html> describes an example of this approach in which a preprogrammed Microchip PIC microcontroller is used to decode Sony remote control commands and transmit command characters to a serial port.

A limitation of this approach is that the receiver supports only one remote control protocol. On the positive side, this approach makes it incredibly easy to add remote control to your appliance. Just plug in the receiver and listen for commands on the serial port.

Measuring in Hardware and Interpreting on the Appliance

In order to accommodate any remote control protocol, we can build receiver hardware that measures waveforms but passes the timing information to the appliance's processor for interpretation. Figure 12-8 illustrates an approach in which the timing of the pulses and spaces is encoded as a series of bytes, each representing time in 50-microsecond increments.



Figure 12-8: Measuring waveform timing in external hardware

Since the appliance is given a complete representation of the input waveform, it can, in theory, decode waveforms for any protocol. The LIRC website provides a link for a protocol called Universal Infrared Remote Transceiver, second version (UIRT2), which works out the details of this approach. It is described at <http://users.skynet.be/sky50985>.

As a related example, the Irman remote control receiver, available at <http://www.evation.com/irman/index.html>, takes an innovative approach and encodes any remote control command waveform by creating “pseudo-random” signatures of six bytes. Irman works on the assumption that different waveforms from a given remote will almost invariably have different signatures. Because this technique applies to any waveform, this type of receiver can work with any of the common remote control protocols, but the user has to train it to interpret the signatures it derives.

As a third example, the LIRC website refers to the USB-IR-Boy project. This project uses the inexpensive Freescale MC68HC908JB8 microcontroller with built-in USB support to provide IR waveform timing values. It also includes a Linux device driver to make these timing values available via the `/dev/usbirboy` device file. Information is available at <http://usbirboy.sourceforge.net>.

Measuring and Interpreting on the Appliance

The third approach to decoding waveforms is appealing because of its simple hardware requirements. All the hardware has to do is power an infrared detector/demodulator and provide the signal as an input to the appliance’s processor. On the negative side, this method does place extra demands on the processor. Specifically, the processor must respond to an interrupt every time the input signal transitions high or low in order to measure timing information. Nevertheless, because of the simple hardware requirements, we have chosen this third approach for the Laddie appliance. In the next section we will work through the details of building this kind of receiver.

Infrared Remote Control Hardware for the Laddie Appliance

If you’re not comfortable with building hardware, you might seek out a friend who is, or—this is our recommendation—jump in and build it yourself anyway. It’s a good first project and a satisfying one because of the new mode of control it gives you for your Linux projects. You will certainly find the remainder of this chapter more educational if you have hardware to experiment with.

To integrate our simple IR receiver with an appliance, we need two things: a power source for the IR detector/demodulator and an input that generates interrupts. The good news is that a typical serial port satisfies both requirements. The output pins on a serial port provide adequate power, and its Data Carrier Detect (DCD) input pin generates interrupts. The bad news is that the serial port output voltages range from 3.7V to 12V on the positive side and from -3.7V to -12V on the negative side. Moreover, the serial port inputs require a swing between these same positive and negative ranges. An IR detector, however, expects a clean 5V power supply (for some parts, 3.3V); it outputs a 0V to 5V signal (for some parts, 0V to

3.3V). Thus, if you want to use a serial port input, you will need additional circuitry to provide the required voltage for the detector and to shift the detector output to valid serial port levels. Figure 12-9 illustrates this kind of circuit.

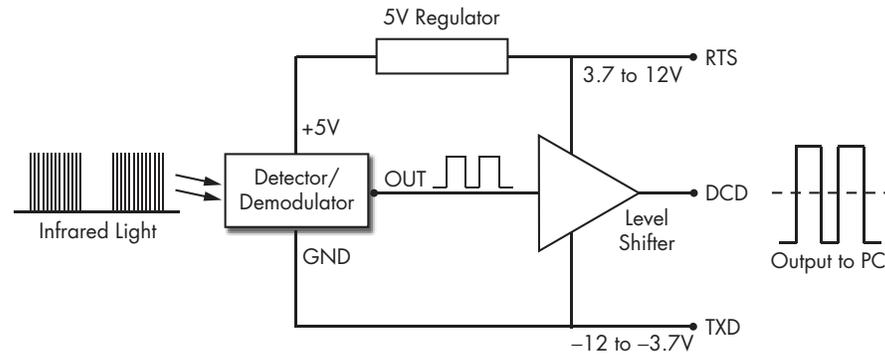


Figure 12-9: A block diagram for a simple IR receiver

Here we assume the serial port has been configured to keep the Request to Send (RTS) output at a high voltage level and the Transmit Data (TXD) output at a low voltage level. The 5V regulator provides the voltage required by the detector. The level shifter provides the correct voltage levels to the DCD input.

NOTE *If you look at IR receiver circuits presented on the Web, you'll find some that take a simpler approach, omitting the level-shifter and providing an output that swings between 0V and 5V. This may work for your computer. If not, or if you want a more robust solution, take the approach we've chosen here.*

Figure 12-10 shows the schematic we chose for the Laddie IR receiver.

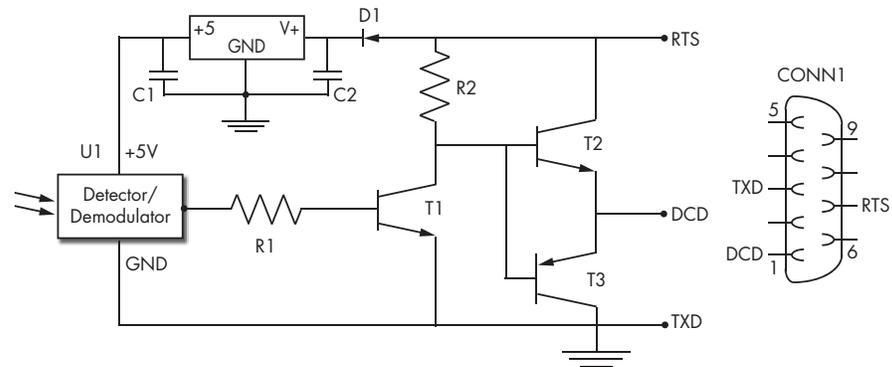


Figure 12-10: The IR receiver schematic used for the Laddie appliance

Here, U1 is an off-the-shelf IR detector/demodulator, U2 is a linear voltage regulator that provides 5V to U1, and the transistor/resistor circuit is the level-shifter that provides an output appropriate for the serial port. The diode, D1, protects the circuit in case the RTS signal is improperly initialized,

which may cause it to go negative relative to TXD. Table 12-2 provides the specific parts that we've used, with their approximate costs. Where we've listed multiple parts, you can assume they are interchangeable. Between Digi-Key (<http://www.digikey.com>) and Jameco Electronics (<http://www.jameco.com>), you shouldn't have any trouble finding these parts.

Table 12-2: Parts List for the Laddie Appliance's IR Receiver

Ref	Part	Cost	Description
U1	Sharp GP1UV701QS, GP1UV70QS, GP1UW701QS, GP1UW700QS	\$1.50	Infrared detector/demodulator
U2	LM78L05, LP2950CZ5	\$0.80	5V linear voltage regulator
C1	0.47 uF or higher	\$0.15	Electrolytic capacitor
C2	0.47 uF or higher	\$0.15	Electrolytic capacitor
D1	1N4148, BAT46	\$0.35	Diode
R1	220K, ¼ watt	\$0.06	Resistor
R2	100K, ¼ watt	\$0.06	Resistor
T1, T2	PN2222A, 2N3904	\$0.20	NPN transistor
T3	PN2907A, 2N3906	\$0.20	PNP transistor
CONN1	DB9 socket	\$0.50	9-pin d-sub serial connector, female

Figure 12-11 shows a few of our prototypes. We recommend starting with a solderless breadboard and 22 AWG solid (non-stranded) wire, as pictured on the left. You'll also need a wire stripper for the 22 AWG wire and a voltmeter for debugging. The one place you'll want to use a soldering iron is to connect wires to the DB9 serial connector. In the left picture, two wires from the DB9 connector provide power and ground to the strips along the edges of the breadboard. The third wire provides the DCD signal back to the computer via the serial cable.

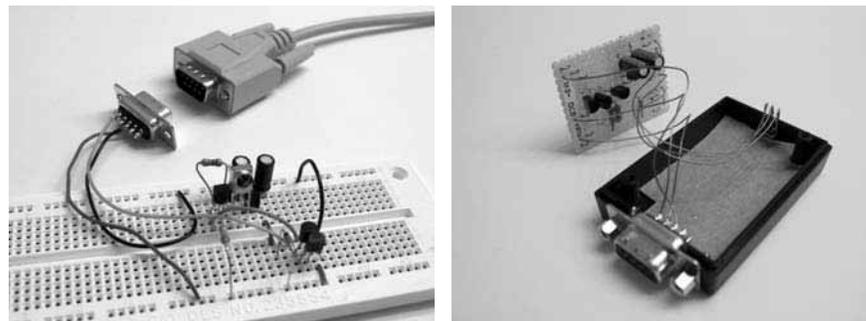


Figure 12-11: Two prototypes for an infrared remote receiver

Once you have a working circuit, you can build something more permanent using perfboard and a plastic enclosure. In the picture on the right, only one side of the enclosure is shown. We drilled a hole at one side for the IR detector/demodulator and cut a hole on the other side for the DB9 connector. A piece of cardboard holds the detector/demodulator in place.

When building your prototype, follow these steps to make sure the circuit is operating properly:

1. Build the circuit, but don't connect the serial port or the IR detector/demodulator. In place of the RTS and TXD pins, use a 9V battery for power. Use the positive battery terminal in place of the RTS input and the negative terminal in place of the TXD input.
2. Verify that the voltage between the regulator output and the negative battery terminal is 5V.
3. Verify that the voltage between the circuit output and the negative battery terminal is at least 8V. (The "circuit output" is the point that you will later connect to the DCD pin of the serial port.)
4. Now connect the open end of R1 to the 5V output of the regulator, and verify that the voltage between the circuit output and the negative battery terminal is zero volts.
5. Finally, complete the circuit by adding the IR detector/demodulator and connecting your computer's serial port.

The remaining tests for your IR receiver hardware require the LIRC software. In the next section, we'll introduce the LIRC software package and describe how we incorporated it into the Laddie appliance.

Installing and Configuring LIRC for the Laddie Appliance

The LIRC software package can be downloaded from <http://www.lirc.org>; it includes an extensive collection of device drivers, daemons, and tools for controlling user applications with remote control hardware. We don't have room to cover all of these elements here, but we will present the layers that make up this software architecture, and we will describe in detail the particular device driver and daemon that are appropriate for the Laddie appliance. Once you've understood this subset, you should find it easy to master any other parts of the architecture required for your own appliance.

Figure 12-12 provides a high-level view of the LIRC software architecture as it applies to the Laddie appliance. At the right of the diagram, we've shown how elements of the LIRC architecture correspond to our earlier, more general discussion of IR receivers.

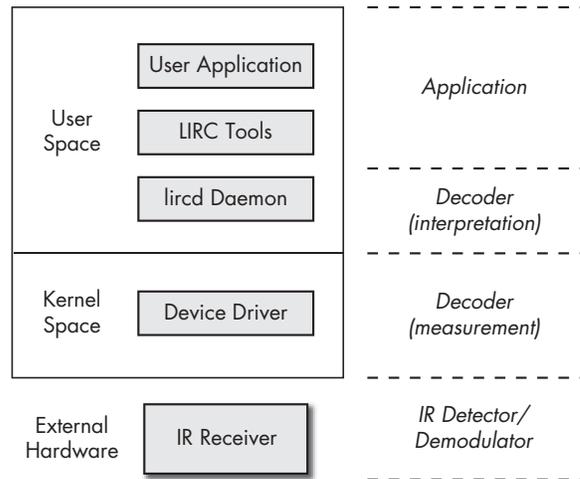


Figure 12-12: The LIRC architecture

In kernel space, a device driver accesses the receiver hardware through an external port. For our appliance, this driver is provided by the LIRC package and uses interrupts to perform waveform timing on the input waveform.

NOTE *As you saw in the section “Decoding Remote Control Waveforms” on page 203, there are some IR receivers that perform the waveform timing and possibly even the waveform interpretation in external hardware. For these receivers, the kernel device driver may be a generic Linux serial driver or a USB driver.*

In user space, for systems such as ours that don’t perform interpretation in external hardware, we use the `lircd` daemon. This daemon accesses a configuration file that characterizes the remote control’s command protocol and analyzes the timing information provided by the device driver to generate the corresponding commands. In some cases, the user application will access the output of the `lircd` daemon directly. The Laddie appliance takes this approach. For applications that were not built with the `lircd` daemon in mind, several LIRC tools are available to process the output of the `lircd` daemon and provide program input, execute appropriate commands, or simulate mouse or keyboard events. We’ll discuss these tools later in the section “LIRC Tools for Controlling Applications” on page 218.

In the remainder of this chapter, we will describe in detail the elements of the LIRC architecture and show how we configured LIRC for the Laddie appliance.

Installing the LIRC Software

The LIRC package is included on this book’s companion CD, and we recommend you use the CD when working through the examples in this chapter. However, if you need to set up your own system in the future, we will describe the steps we took to install the package.

We downloaded version lirc-0.8.1 from <http://www.lirc.org> and installed it with these commands:

```
./setup.sh
./configure --with-kernel-dir=/usr/src/linux-2.6.10 --with-driver=serial
make
make install
```

The `setup.sh` script asked us to make choices about our installation. Under the Driver Configuration (driver:serial io:0x3f8 irq:4) menu, we chose the **Home-brew (16x50 UART compatible serial port)** driver, selected **COM1 (0x3f8, 4)** for the base address and IRQ, and disabled all driver-specific options. Under the Software Configuration menu, we disabled all options. Then we selected **Save Configuration and exit**.

LIRC is a package that allows you to decode and send IR and other signals of many (but not all) commonly used remote controls. It includes daemons that decode the received signals as well as user space applications that allow controlling a computer with a remote control.

The `./configure --help` command provided a long list of driver choices, as well as a dauntingly long list of configuration options. For our appliance, the defaults were generally appropriate. We only needed to provide the location of our Linux kernel source tree using the `--with-kernel-dir` option and to specify the serial driver with the `--with-driver` option.

Figure 12-13 shows how representative components installed by the LIRC package fit into the LIRC software architecture. Although the LIRC package includes utilities that support the X Window System, we don't show them here, since the Laddie appliance doesn't use X.

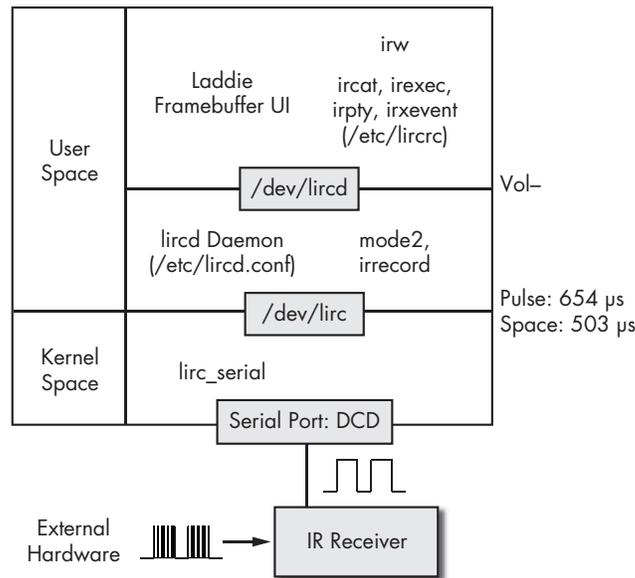


Figure 12-13: Components of the LIRC package

Before we dive into the details, let's start with a quick, bottom-to-top tour and explain how the LIRC receiver controls the Laddie appliance's framebuffer UI, which is described in Chapter 11. The IR receiver external hardware provides the remote control waveform to the DCD pin of a serial port. In kernel space, the `lirc_serial` device driver (one of many included in the LIRC package) monitors this pin and produces a binary stream of timing data via the device file `/dev/lirc`. In user space, the `lircd` daemon analyzes the timing data from the `/dev/lirc` device file to provide a sequence of command strings on the Unix socket `/dev/lircd`. The framebuffer UI connects directly to this socket in order to respond to user input.

Now for all the details we left out. In the remainder of this section, we will look more carefully at each of the layers of the LIRC software architecture.

Configuring the `lirc_serial` Kernel Device Driver

The `lirc_serial` device driver is actually implemented by two kernel modules, `lirc_serial.ko` and `lirc_dev.ko`, which were placed in the directory `/lib/modules/2.6.10/misc/` when we installed the software. In order to use these modules, we had to perform three additional steps: free up a serial port, create a device file, and load the modules into the kernel. We created a startup script, `lircd`, to perform these steps. We will review the steps here; you can see the complete code on the CD in the `/etc/rc.d/init.d` directory.

Freeing Up a Serial Port

For the Laddie appliance's IR receiver input port, we chose COM1 (`/dev/ttyS0`). The Linux kernel typically enables COM1 through COM4 as serial ports at startup; thus, we needed to free up COM1 for LIRC. To do this, we used the `setserial` command:

```
setserial /dev/ttyS0 uart none
```

By setting the type of the hardware (the UART) to *none*, this command disabled the specified port.

NOTE *UART stands for Universal Asynchronous Receiver Transmitter. A UART handles the low-level implementation of a serial link so that the CPU need only be concerned with providing bytes to transmit and processing bytes that are received.*

For the remainder of this chapter, we recommend that you boot the Laddie appliance using the Laddie CD and follow along with the exercises. After booting the CD, exit the framebuffer UI (press ESC), and log in as root with an empty password. Verify that port COM1 was configured properly by executing the following command at the `laddie:~#` prompt:

```
laddie:~# setserial /dev/ttyS0
```

You should see the following output:

```
/dev/ttyS0, UART: unknown, Port: 0x03f8, IRQ: 4
```

The UART type is unknown, which means the port is available.

Creating a Device File

You may recall from the previous chapter that we used a device file `/dev/fb0` to expose the framebuffer functionality. Similarly, we had to provide a device file to expose the `lirc_serial` functionality. The following code created the character device file `/dev/lirc` with major number 61 and minor number 0, as required for the `lirc` driver.

```
mknod /dev/lirc c 61 0
```

With the Laddie CD, verify that the `/dev/lirc` device file exists by using the command:

```
laddie:~# ls -l /dev/lirc
```

You should see the output:

```
❶crw----- 1 root root ❷61, ❸0 2007-01-27 08:03 /dev/lirc
```

This indicates that the file represents a character device that is ❶ readable and writable by root, ❷ with major number 61, and ❸ with minor number 0.

Loading the `lirc_serial` Modules into the Kernel

With the serial port available and the device file in place, we were able to load the `lirc_serial` device driver using the `modprobe` command:

```
modprobe lirc_serial
```

To verify that the `lirc_serial` device driver is loaded, execute the following:

```
laddie:~# lsmod | grep lirc
```

You should see output like this:

```
lirc_serial          13152  1  
❷lirc_dev            14804  1 ❶lirc_serial
```

The first column shows the loaded modules; the fourth column shows dependencies. Here we see that ❶ the `lirc_serial` module depends on ❷ the `lirc_dev` module.

NOTE *If we had wanted to use a different port than COM1 for our IR receiver, we would have provided additional arguments to the `lirc_serial` driver with the `modprobe` command. To specify COM2, we would have used the command `modprobe lirc_serial irq=3 io=0x2f8`. The default `irq` and `io` values for COM3 and COM4 are (4, 0x3e8) and (3, 0x2e8), respectively.*

Testing the `lirc_serial` Driver

Now that we've verified that the `lirc_serial` driver is loaded properly, we can use the Laddie CD to test the IR receiver hardware we built earlier. We'll begin by reviewing what we want to test.

Recall that the signal provided by our LIRC receiver looks something like Figure 12-14 (at least in the case of a Sony TV Volume- command).

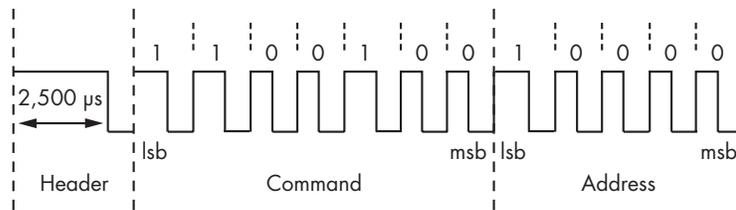


Figure 12-14: Waveform for the Sony TV Volume- command

The job of the `lirc_serial` kernel device driver is to measure the timing of spaces and pulses in this signal and provide that information via a device file. The particular waveform shown here complies with the Sony TV protocol, but the `lirc_serial` driver is designed to work with any protocol. The driver includes an interrupt handler that is invoked every time the DCD pin changes state. The handler uses a system timer to measure the pulses and spaces in microseconds, and then it emits this timing information via the `/dev/lirc` device file as a sequence of 32-bit words. In each word, bits 0 through 23 specify the length of the space or pulse in microseconds (with a maximum value of 0xFF FFFF). Bit 24 is zero for a space and one for a pulse. Bits 25 through 31 are always zero. To test the `lirc_serial` device driver, we'd like to verify that these values are generated when we press a remote control button.

Before we can access the `/dev/lirc` device file, we need to make sure it's not already in use by some other process. When the Laddie CD boots, it launches the `lircd` daemon in order to support the framebuffer UI. Since the `lircd` daemon accesses `/dev/lirc`, we prepare for our test by killing that process.

Execute the following commands at the Laddie appliance command prompt:

```
laddie:~# laddie stop
laddie:~# kill $(pidof lircd)
```

After the first command, you will need to wait a few moments for the Laddie application to stop. In the second command, the `pidof` function outputs the process ID of the `lircd` process. The `$(...)` construct provides this output as a parameter to the `kill` command, which terminates the specified process. We can now access the `/dev/lirc` device file for our own purposes.

As we saw in the previous chapter, we can use the `cat` command to access output that is provided via device files. Now we don't want to simply `cat` the output of `/dev/lirc` to the console, because some of the output data might be interpreted as control characters and the console could end up in an unusable state. One thing we can do is pipe that output through the `hexdump` utility, which translates binary data into printable ASCII hexadecimal characters.

Enter the following command:

```
laddie:~# cat /dev/lirc | hexdump
```

Now, any pulses that arrive on the DCD pin of serial port COM1 will be measured by the `lirc_serial` device driver, read by the `cat` command via the `/dev/lirc` device file, and displayed in ASCII hex by `hexdump`. To generate such pulses, connect your IR receiver to the COM1 port, point your remote control at the IR detector/demodulator, and press a button. (At this point, the particular kind of remote doesn't matter.) If everything is working properly, you should see output like the following:

```
00000000 df67 0061 099d 0100 01fb 0000 04f6 0100
00000010 0216 0000 04fd 0100 01fa 0000 026e 0100
00000020 022d 0000 028a 0100 01fb 0000 04f7 0100
00000030 01fd 0000 02a1 0100 01fa 0000 02a0 0100
00000040 01fb 0000 04fc 0100 01fb 0000 02a0 0100
00000050 01fa 0000 0283 0100 0218 0000 0289 0100
```

This is the output we generated by briefly tapping the Volume- button on the universal remote that we programmed for Sony TV.

If you're not sure your IR receiver hardware is working (or if you don't have an IR receiver at this point), you can still test the `lirc_serial` device driver by creating random pulses on the DCD pin of the serial port. One way to do this is to intermittently connect pin 1 (DCD) to pin 7 (RTS) of the serial port. If this doesn't produce a result, try intermittently connecting pin 1 to pin 3 (TXD). If your computer's serial port is built to standard specifications, it won't be a problem if you connect the wrong pins by mistake. Still, if you've just purchased a fancy, new laptop, you might want to try this experiment on a friend's Linux box first.

Using the `cat /dev/lirc | hexdump` command is a good exercise because it demonstrates that the output of `/dev/lirc` is simply binary data that can be read like a file. Neither `cat` nor `hexdump` know anything about infrared, yet they display the data just fine. But the output of `hexdump` isn't easy to read.

Fortunately, the LIRC package includes a utility, `mode2`, that does understand the output of the `lirc_serial` driver and can display it as pulse and space timing data.

Press `CTRL-C` to terminate the previous command, and execute the following one:

```
laddie:~# mode2
```

Now point an IR remote control at the receiver while pressing buttons. The command `mode2` reads the output of `/dev/lirc`, parses the 23-bit timing data and the one-bit pulse or space indicator, and produces a stream of pulse and space timing information. As an example, we observed the following train of space and pulse timings from `mode2` when we stimulated our IR receiver with a single Sony TV Volume- command.

```
laddie:~# mode2      space 568      pulse 1265
space 5794213        pulse 663      space 517
pulse 2471           space 494      pulse 663
space 496            pulse 1260     space 516
pulse 1282           space 497      pulse 633
space 546            pulse 685      space 546
pulse 1263           space 518      pulse 639
space 517            pulse 661      space 496
pulse 611           space 515      pulse 682
```

If you refer to “Protocols for Encoding Remote Control Commands” on page 198, you will notice that these timing values are noisier than the idealized waveform would suggest. The first pulse is roughly 2,500 milliseconds and corresponds to the header. The other pulses are roughly 1,300 or 650 milliseconds, corresponding to ones and zeros, respectively. The spaces are roughly 500 milliseconds, but note the large initial space value corresponding to the time between button presses. It is the job of the `lircd` daemon to reject the pulse trains that do not correspond to valid waveforms and to correctly interpret the ones that do. When you are finished experimenting with `mode2`, press `CTRL-C` to terminate the utility.

At this point, we have established that the `lirc_serial` device driver is working. In the next section we will provide instructions on configuring the `lircd` daemon, which will use the output of this device driver.

NOTE *If you would like to write a program that uses the output of the `lirc_serial` device driver directly, the source code for the `mode2` utility provides an example of how to access the `/dev/lirc` device file. This source is available from <http://www.lirc.org> and is also provided in the `lirc-0.8.1.tar.bz2` tarball in the `/usr/src/packages/` directory of this book’s companion CD.*

Configuring the lircd Daemon

The cleverest part of the LIRC package is the lircd daemon. This is the part that analyzes the noisy timing values coming from the /dev/lirc device file and produces a sequence of commands that are easily parsed by downstream LIRC tools or user applications.

In order for the lircd daemon to interpret the timing data from /dev/lirc, it has to understand the remote control protocol. The configuration file /etc/lircd.conf captures this protocol information. The following is the lircd.conf file used by the Laddie appliance with comments and a few of the button entries removed to save space.

```
begin remote
  name      SONY-TV
  bits      12
  flags     SPACE_ENC
  eps       30
  aeps      100
  header    2457  525
  one       1269  520
  zero      650   520
  gap       26076
  toggle_bit 0
  begin codes
    POWER          0xA90
    ENTER          0xD10
    VOL-           0xC90
    VOL+           0x490
    CH-            0x890
    CH+            0x090
  end codes
end remote
```

You don't need to understand the entries in this file to use LIRC, but we'll make a few comments here in case you want to edit the file manually. The *name* can be any string you like that describes the remote. The *bits* field is the total number of data bits (in our case, command-code bits plus address bits). The *eps* and *aeps* fields represent relative and absolute error tolerances (in our case, 30 percent and 100 microseconds). The *header*, *one*, and *zero* fields represent the pulse and space timings (in microseconds) for the header and data bits. There is a *gap* of about 26,000 microseconds between repeated commands, and there is no toggle bit that changes for repeated commands. Note that these fields reflect actual timings measured by the device driver and can vary from the protocol standard. The *codes* are the actual data bit sequences for the various commands. For the additional fields that may apply for other remotes, you can see the details at the WinLIRC web page, <http://winlirc.sourceforge.net/technicaldetails.html>.

If you have a remote that uses the same protocol as ours, you should be able to control the Laddie appliance without updating the `/etc/lircd.conf` file. (Again, we are using an RCA RCU410 universal remote, programmed as a Sony TV, code 002.) The LIRC website also provides configuration files for many remotes, but using the LIRC `irrecord` utility, it's easy enough to generate these files from scratch. The `irrecord` utility creates configuration files by monitoring the output of `/dev/lirc` while prompting the user for remote control input.

Now let's create a configuration file for your remote. As we mentioned when we were testing the `lirc_serial` device driver, we need to kill the `lircd` daemon before we access `/dev/lirc`. If you didn't kill `lircd` earlier, do so now:

```
laddie:~# kill $(pidof lircd)
```

To create a new `lircd` configuration file, rename or delete the old one, then run the `irrecord` command:

```
laddie:~# mv /etc/lircd.conf /etc/lircd.conf.bak
laddie:~# irrecord /etc/lircd.conf
```

Read the instructions printed by the `irrecord` utility carefully. The utility will prompt you to press remote control buttons in a particular sequence, and it will also ask you to assign names for the buttons you choose to program. Since you will be using the remote to control the Laddie framebuffer UI, you will need to provide the button names that the Laddie appliance expects. It doesn't matter how you assign the actual buttons, but you will need to use the following names in uppercase letters: `POWER`, `VOL+`, `VOL-`, `CH+`, and `CH-`. If you restart the `irrecord` utility, be sure to rename or delete the previous `/etc/lircd.conf` file first. Once you're satisfied with the configuration file, you are ready to test the `lircd` daemon.

NOTE *If you choose to download a configuration file for your remote from <http://www.lirc.org>, you will need to edit it to make sure the button names are the ones the Laddie appliance expects. Keep in mind that any updated configuration files will be replaced with the original files when you reboot the Laddie CD.*

Testing the `lircd` Daemon

To use your new `lircd` configuration file, start the `lircd` daemon with the command:

```
laddie:~# lircd
```

This command will complete immediately without printing anything. To verify that the daemon is running, execute the command:

```
laddie:~# pidof lircd
```

and verify that it returns an integer. The lircd daemon will read timing data from the /dev/lirc device file and, using the configuration specified in /etc/lircd.conf, provide button-press information at the Unix socket /dev/lircd in the form of newline-delimited ASCII strings.

Unix sockets are different from regular files or device files. In particular, you can't use the system call open() to access them; you have to use connect() instead. This means that we can't simply use cat to examine the output of /dev/lircd the way we did with /dev/lirc. Let's write a simple program, socket_cat, that does allow us to view this output.

NOTE *If you are eager to test the lircd daemon and would rather skip this exercise, you can use the LIRC utility irw, with no arguments, to display the output of /dev/lircd. However, the program socket_cat will help you understand how the Laddie appliance works, since it uses the same approach as socket_cat to access remote control button presses.*

If you've programmed with sockets before, the following program will look familiar. We use the function ❶ socket to create an unnamed, Unix internal socket. We use the function ❷ connect to connect to the named socket /dev/lircd. Then we ❸ loop forever, copying all received data to the standard output.

```
#include <unistd.h>          /* read, write */
#include <sys/un.h>          /* sockaddr_un */
#include <sys/types.h>       /* socket, connect */
#include <sys/socket.h>      /* socket, connect */
#include <string.h>          /* strcpy */

int main(int argc, char *argv[])
{
    int fd, i;
    char buf[128];
    struct sockaddr_un address;
    address.sun_family=AF_UNIX;
    if(argc<2){
        printf("Usage: socket_cat <unix socket path>\n");
        return;
    }
    strcpy(address.sun_path, argv[1]);

    ❶ fd=socket(AF_UNIX, SOCK_STREAM, 0);
    ❷ if(connect(fd, (struct sockaddr *)&address, sizeof(address)) == -1){
        perror("Connect");
        exit(1);
    }
    ❸ for(;;){
        i=read(fd, buf, 128);
        write(STDOUT_FILENO, buf, i);
    };
}
```

This program is on the CD at `/Code/src/examples/socket_cat.c`. Build and run the program, using the following commands:

```
laddie:~# cd /Code/src/examples/socket_cat
laddie:~# make
laddie:~# ./socket_cat /dev/lircd
```

Then press a few buttons on your remote. You should see output like the following.

```
00000000000000c90 00 VOL- /etc/lircd.conf
00000000000000490 00 VOL+ /etc/lircd.conf
00000000000000890 00 CH- /etc/lircd.conf
00000000000000090 00 CH+ /etc/lircd.conf
00000000000000a90 00 POWER /etc/lircd.conf
00000000000000a90 01 POWER /etc/lircd.conf
00000000000000a90 02 POWER /etc/lircd.conf
```

Each string includes a 16-character hexadecimal command code, a hexadecimal repetition count, a command string, and a name for the remote (which defaults to the name of the lircd config file). Note how, at the end of this sequence, the repetition count increases when the POWER button is held down continually.

The 16-character command codes are generally not useful, since all relevant information is captured by the names of the commands and the remote. However, it is interesting to see how the command code corresponds to the input waveform. Note, for example, that `0xc90` is the hexadecimal representation for the 12 bits (left to right) in the command waveform for the Sony TV Volume- command that we saw in “Protocols for Encoding Remote Control Commands” on page 198. When you are done with `socket_cat`, press CTRL-C to terminate the program.

NOTE *When a remote control button is pushed, depending on the button and the protocol, the commands can repeat pretty quickly. For the Laddie framebuffer UI, we took advantage of the repetition count associated with the lircd output to ignore all but the first command associated with each button press.*

LIRC Tools for Controlling Applications

As you saw in the previous section, it is simple to write a program that responds to remote control commands via the `/dev/lircd` socket. But what if you want to use a remote to control a program that already exists, but was designed, say, for keyboard input rather than remote control input? In fact, the LIRC package addresses this need with tools that connect to the `/dev/lircd` socket and produce the kinds of output that many programs do expect.

The `ircat` tool is a good example because it is the simplest; it prints user-specifiable, newline-delimited strings to the standard output when remote control buttons are pressed. If you have a program that takes commands

from standard input, you can control it with LIRC by piping the output of `ircat` to your program. To map remote control buttons to appropriate output, configure the file `/etc/lircrc`. The HTML documentation provided with the LIRC package provides details on the format of this file.

Similarly, the LIRC package provides an `irpty` utility for simulating keyboard input, an `irxec` utility for invoking system calls, and an `irxevent` utility that generates X events (for systems running X). Again, these actions are mapped to remote control buttons according to the `/etc/lircrc` file. All of these utilities use an API called the `lirc_client` library to access the `/dev/lircd` socket. The source code for the `ircat` tool provides a simple example of how to use this library.

Finally, the LIRC package contains a daemon `lircmd` that uses remote control input to emulate a mouse. This daemon connects to the `/dev/lircd` socket and produces mouse events on the pipe `/dev/lircm`. The configuration file `/etc/lircmd.conf` selects the protocol for X mouse events (e.g., IntelliMouse) and specifies how remote commands map to mouse movements and button presses. The `XF86Config` file must be updated to include `/dev/lircm` as an input device. Again, the LIRC HTML documentation provides details.

Controlling the Laddie Appliance

For the Laddie appliance, we installed the LIRC package and configured the `lirc_serial` driver and `lircd` daemon as described in the previous sections. Since we built the Laddie appliance from scratch, we did not need to use LIRC tools like `ircat` or `irpty`; instead, we wrote code similar to the `socket_cat` example, which accessed the `/dev/lircd` socket directly.

As discussed in the previous chapter, Laddie's framebuffer user interface is built on the Simple DirectMedia Layer (SDL) library. Since SDL includes its own event handler which, in particular, handles keyboard presses, it was a simple matter to incorporate remote control events. We created a separate `lircHandler()` thread to read commands from the `/dev/lircd` socket, parse these commands, and then push appropriate keyboard events onto the SDL event queue. Specifically, we responded to the `Channel+/-` and `Volume+/-` remote control commands by simulating the SDL keypress events for the Up, Down, Right, and Left arrow keys, respectively. We responded to the remote control Power command with the SDL Enter keypress event. In Laddie's SDL event handler, we responded to these keypress events by calling navigation commands in Laddie's menu object. This use of the `/dev/lircd` output to control Laddie's framebuffer menu is illustrated in Figure 12-15.

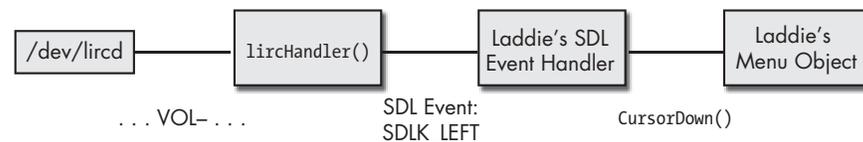


Figure 12-15: Controlling the Laddie framebuffer UI

If you would like to see the details of the `lircHandler()` thread, you can take a look at the `/Code/src/fbmenu/lirc_if.cc` file on the CD.

If you have built the IR receiver we described in this chapter and successfully worked through the exercises, you should be able to use it to control the Laddie framebuffer UI. We had stopped the Laddie appliance daemons in order to do the exercises; you will need to restart them now. Do this with the following command:

```
laddie:~# laddie start
```

This will take a few moments, after which you should see the framebuffer UI. Now experiment with the buttons you programmed when you used `irrecord` to create the `lircd.conf` file. You should be able to navigate through the menu buttons and switch menu pages.

Summary

Infrared light is a useful means for controlling an appliance when the appliance is in the line of sight. To be effective, infrared light must be modulated by the transmitter with a signature frequency, and this modulation must be removed by the receiver. Fortunately, there are commercial devices that make it easy to meet these requirements. For IR receivers, we introduced infrared detector/demodulators and showed how to use them in simple IR receiver circuits. For IR transmitters, we recommended using universal remotes.

We also described the Linux Infrared Remote Control (LIRC) package as a useful tool for controlling appliances. This package provides device drivers and daemons for measuring and interpreting infrared waveforms, as well as utilities for controlling appliances. Although we didn't discuss the entire LIRC package, we did describe those elements of the package that we used for the Laddie appliance. We hope this overview of infrared communications and this example application of the LIRC software package will be a useful starting point if you decide to use infrared remote control for your own appliance.



