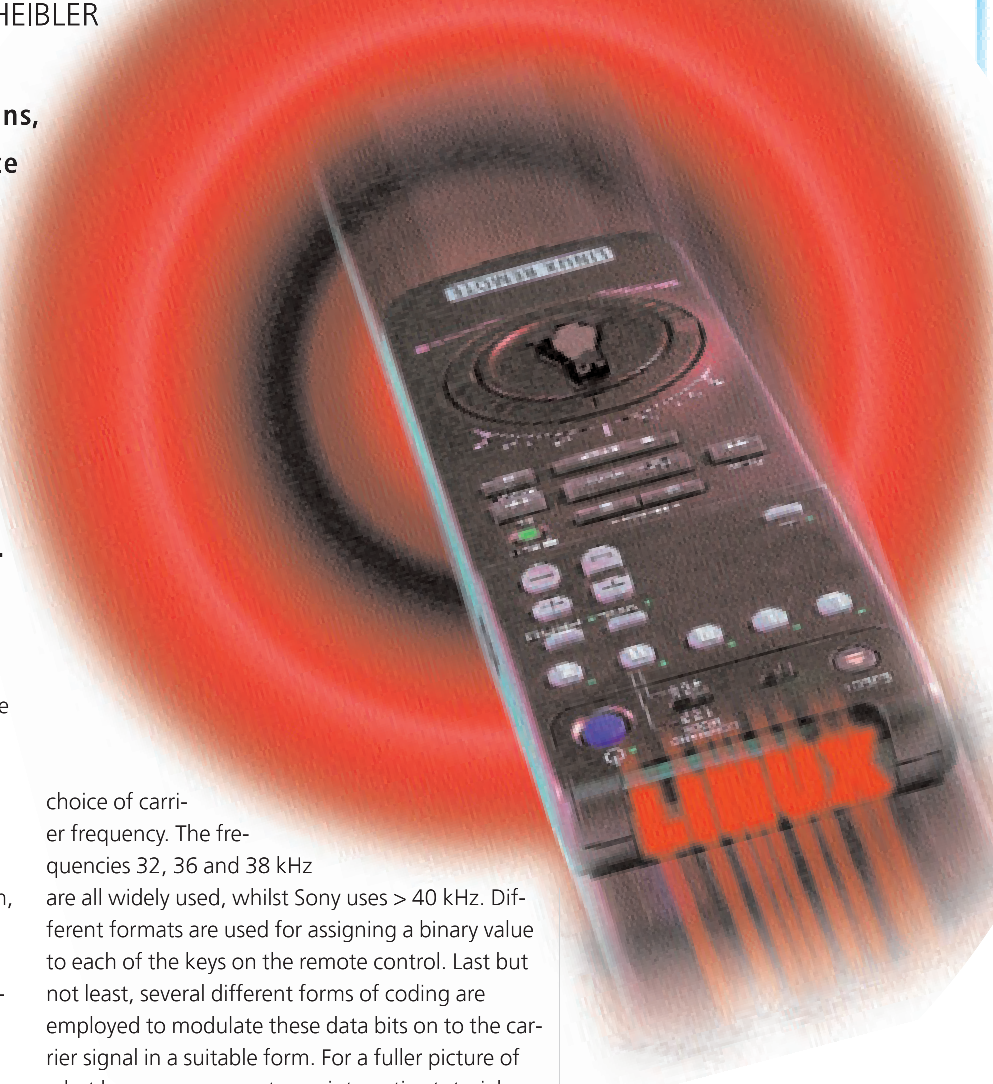
Linux Infra-red Remote Control

LINUX TAKES CONTROL

KARSTEN SCHEIBLER

Video recorders, televisions, hi-fi systems and satellite receivers: just about every consumer entertainment device now comes with its very own infra-red remote control. Your Linux computer, too, can be controlled from the comfort of your armchair. To find out how, read on...

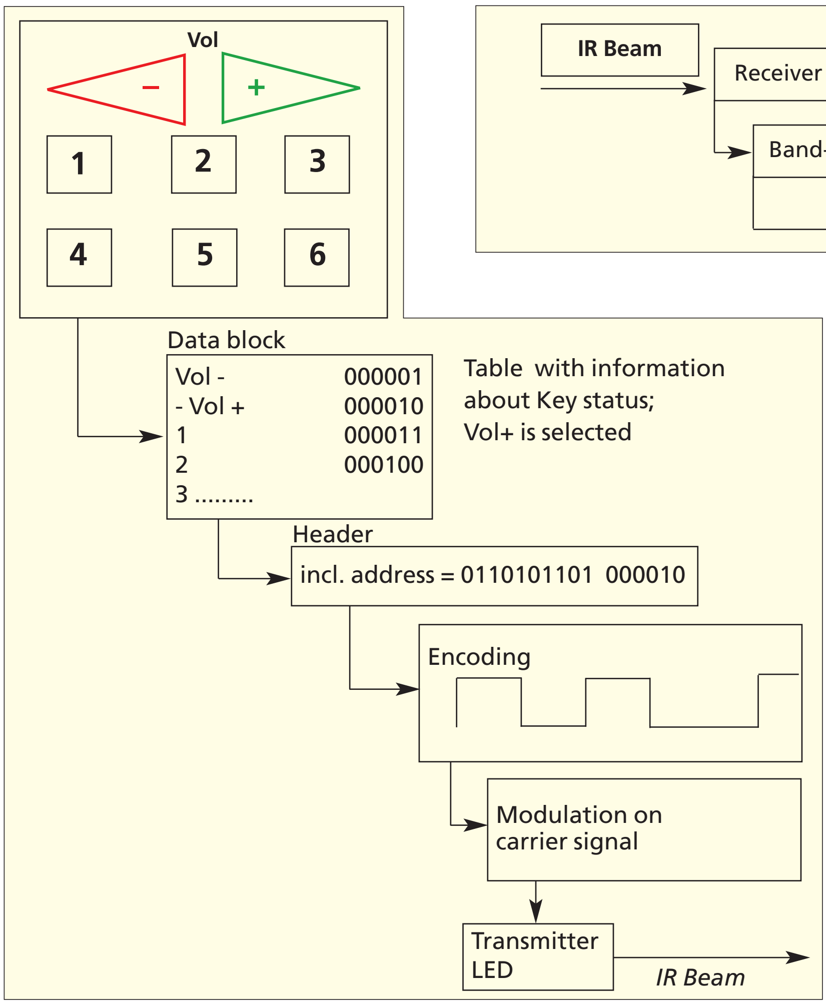


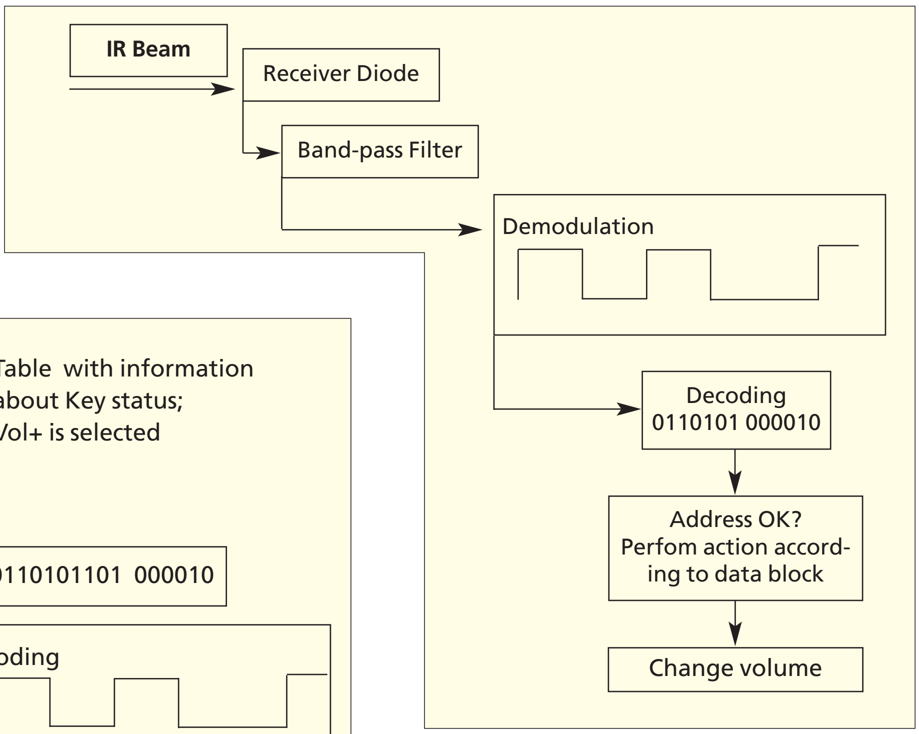
Infra-red is the widely-used standard for the remote controls supplied with consumer devices. Only a few up-market products use radio and thus do not have to rely on an unobstructed line of sight to the target device. Home computers, too, can be controlled remotely with the appropriate equipment.

Before we examine the subject in greater depth, let's first take a look at the fundamentals of infra-red remote controls. In essence, the transmission process can be described as follows (and is illustrated in Fig. 1). When a key on the remote control handset is pressed, the value corresponding to this key is determined. This value, expressed as a binary number, is modulated upon a carrier signal (normally in the range 30 to 40 kHz) and leaves the handset via the infra-red transmitting diode.

In practice, however, different manufacturers have pursued separate paths, for example in the choice of carrier frequency. The frequencies 32, 36 and 38 kHz are all widely used, whilst Sony uses > 40 kHz. Different formats are used for assigning a binary value to each of the keys on the remote control. Last but not least, several different forms of coding are employed to modulate these data bits on to the carrier signal in a suitable form. For a fuller picture of what happens you can try an interactive tutorial on the Web where, taking a Sony remote control as an example, you can marvel at the signals generated.

Having arrived at the receiver end, the infra-red beam must first pass through a filter that masks out all interfering light frequencies. After that it falls on


Vol
–
+
1
2
3
4
5
6
Data block
Vol - 000001
- Vol + 000010
1 000011
2 000100
3 ....
Table with information about Key status; Vol+ is selected
Header
incl. address = 0110101101 000010
Encoding
Modulation on carrier signal
Transmitter LED
IR Beam



[left]
Fig. 1a: The remote control generates a code for the key that was pressed using a device-specific address and a binary code for the key

[right]
Fig. 1b: The receiver demodulates the signal, determines the original key code and initiates the corresponding action.

Types of encoding

Let's take a closer look at the transmission process. The remote control's keypad is connected to a logic circuit that continuously checks for depressed or released keys. If a change takes place, the transmission chip picks out the bit code for the key concerned and very often adds on a further bit that indicates whether the key has been pressed or released. These bits might be described as the data part of the bit code. Also added to the code is the so-called address part. This is nothing more than a fixed bit sequence in the remote control and base unit. This code differs, even for different devices from the same manufacturer. This ensures that only the intended device responds to the command from the remote control.

The resultant overall bit code is usually between 14 and 32 bits in length, the length of the data and address parts differing from manufacturer to manufacturer. Converted into a stream of high and low signal levels, the result is a self-synchronising serial data stream. These level states – also referred to in this context as mark (high) and space (low) – may not directly correspond to the ones and zeros of the binary bit pattern. The encoding that is used always utilises both levels for each bit. It is the encoded binary value that modulates the low-frequency carrier frequency (in theory this is nothing more than an AND operation between the data stream and the carrier generator) which is finally emitted via the infrared transmission diode.

Figure 2 shows some details of the three most frequently used encoding methods. The first two methods each vary the time during which the level of the data stream is high (Pulse Width Modulation) or low (Pulse Interval Modulation) to differentiate between the bit states 0 and 1. However, the third method is the most commonly used. It represents the two bit states through different signal edge changes (BiPhase coded). To help the receiver identify the start of a new bit code, some models of remote control also send a long pulse and a space before the encoded data itself is transmitted.

the infra-red receiving diode. On the output of this device the modulated low-frequency carrier signal will appear. To guarantee this, a band-pass filter is connected in series with the receiving diode, which more or less ensures that only the frequency of the carrier signal passes through. Other infra-red disturbing influences can also be effectively filtered out in this way, such as solar radiation. Finally, decoding of the received bit-code takes place, so that at last the desired action may be carried out such as changing the volume level.

Computer control

If you want to use a remote control on your PC, there are a number of options. Various TV cards offer possibilities. Many come supplied with both a remote control and a matching infra-red receiver (for example, the Hauppauge WinTV/Radio.) This makes matters really simple and saves you having to build yourself an appropriate receiver. Apart from this, you can exploit the TV functionality of cards that have the Brooktree 848/878 chipset. Note that none of these TV cards come supplied with suitable Linux drivers for the remote control. There are open-source drivers for the various models, but it must be admitted that some of them need some further development.

One further remark concerning the remote control of TV cards. A short while ago Hauppauge cards were bundled with a remote control for connection to the serial port, the product in question being the Anir Multimedia Magic from Animax. The infra-red receiver of this remote control can be driven in almost the same way as the serial receiver just described and can also be used under Linux.

Those with no TV card at their disposal and those who prefer to take out the soldering iron can always make use of the serial interface (Fig. 3) by connecting an infra-red receiver to it. You can build a suitable receiver yourself with very little effort. In this case we make use of the fact that the serial port's UART triggers an interrupt whenever the DCD signal line changes level. A special driver is needed to respond to this. As a third choice the parallel port can be used with a similar self-built receiver, although greater effort is involved. Circuit diagrams for both types of receivers can be found on the LIRC home page.

Some infra-red receiver designs that are connected to the serial interface don't require a special driver since control is done through */dev/ttySx*. The number of components required in the receiver is only marginally greater compared to that of the basic DCD receiver. However, the design uses a PIC16xxx microcontroller. This is an obstacle for many would-be builders as they will not have the necessary equipment to program the device.

Using a dedicated infra-red receiver isn't the only option. Some of the IrDA ports on PCs can be coaxed into transmitting and receiving the signals of an infra-red remote control. Due to the lack of hardware I have not yet been able to acquire any personal experience with this. Further information about it can be found on the LIRC home page, but in the end there is no substitute for actually trying it.

Another possibility is IRman. Available as a commercial product for 33 US dollars, this device is also connected to the serial port and uses a PIC as decoder. A series of drivers are available from the manufacturer for specific (Windows) programes. If you are interested in finding out how to control the device you may take a look at the Perl code from Cajun.

One option for building your own receiver is to use the SFH506 series integrated circuits from Siemens (now Infineon). These are chips containing an infra-red receiving diode, band-pass filter and demodulator (Fig. 4). Because of the integrated

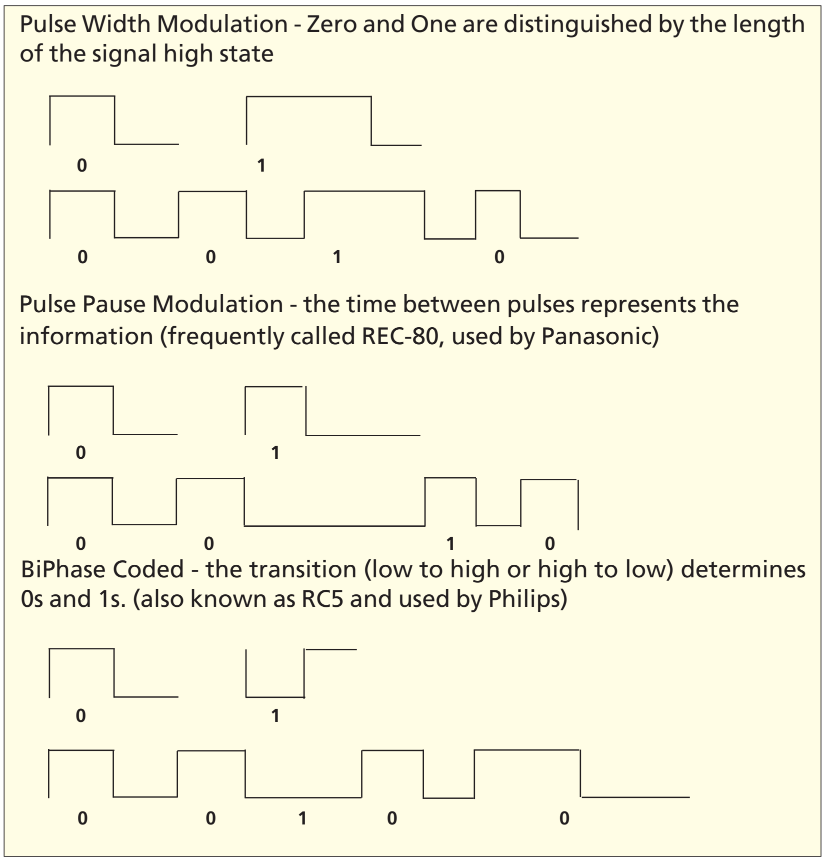
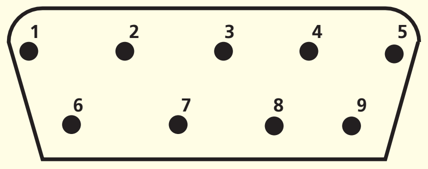

Pulse Width Modulation - Zero and One are distinguished by the length of the signal high state
0
1
0
0
1
0
Pulse Pause Modulation - the time between pulses represents the information (frequently called REC-80, used by Panasonic)
0
1
0
0
1
0
BiPhase Coded - the transition (low to high or high to low) determines 0s and 1s. (also known as RC5 and used by Philips)
0
1
0
0
1
0
0

Fig. 2: The three commonly used types of modulation, Pulse Width Modulation (Sony), Pulse Interval Modulation (REC-80, Panasonic) and BiPhase Coding (RC5, Philips).



Pin #1	DCD	Data Carrier Detect
Pin #2	RD/RX/RXD	Receive Data
Pin #3	TD/TX/TXD	Transmitted Data
Pin #4	DTR	Data Terminal Ready
Pin #5	GND	Signal Ground
Pin #6	DSR	Data Set Ready
Pin #7	RTS	Request to Send
Pin #8	CTS	Clear to Send
Pin #9	RI	Ring Indicator

Fig. 3: The serial interface also provides the infra-red receiver with the supply voltage via RTS and DTR.

functionality there are a number of types: SFH506-30, SFH506-36, SFH506-38 which correspond to the carrier frequencies 30, 36, 38 KHz. It may be difficult to get hold of an SFH506 now since Siemens has stopped producing them.

There is a similar integrated circuit though with the same functional scope: the TSOP1736/38/40 [4]. Whichever one is used, SFH506 or TSOP17xx, the operation is identical. You apply a supply voltage to two of the three connector pins and obtain the serial data stream on the third pin when an infra-red signal with the correct carrier frequency has been received.

This data stream can now be passed to DCD so that using the UART interrupts the duration of the pulse and space can be determined. Once the durations and level states plus the encoding method are known it is possible to deduce the bit code that was sent.

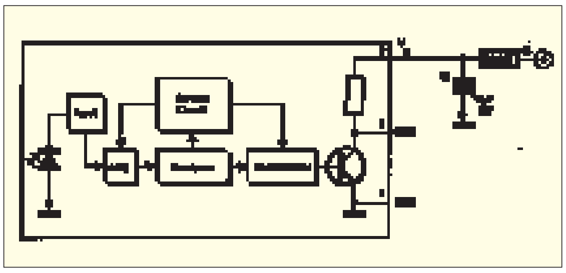
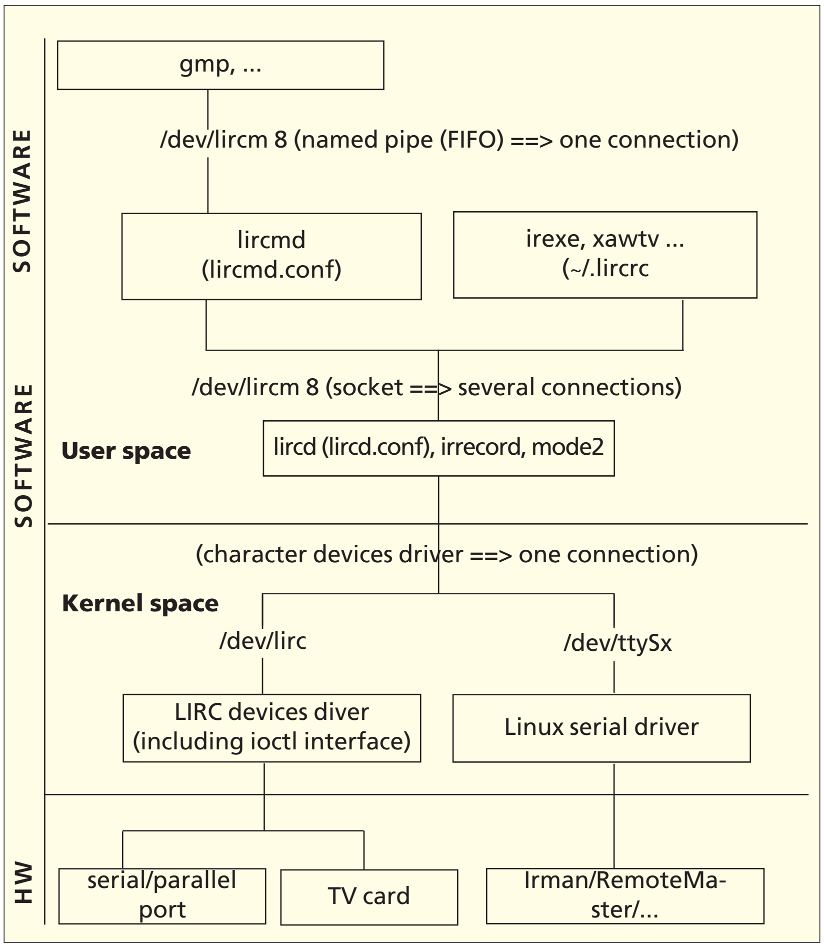


Fig. 4: Besides an optical filter (black housing) commercially available receivers such as the SFH506 or TSOP 17XX contain a band-pass filter, gain controller and the output stage.


SOFTWARE
gmp, ...
/dev/lircm 8 (named pipe (FIFO) ==> one connection)
lircmd
(lircmd.conf)
irexe, xawtv ...
(~/.lircrc
SOFTWARE
/dev/lircm 8 (socket ==> several connections)
User space
lircd (lircd.conf), irrecord, mode2
(character devices driver ==> one connection)
Kernel space
/dev/lirc
/dev/ttySx
LIRC devices diver
(including ioctl interface)
Linux serial driver
HW
serial/parallel
port
TV card
Irman/RemoteMaster/...
LIRC is neatly constructed as a layer model. Only a few programs will need exclusive access to the hardware devices; a number of clients can connect simultaneously to the *lircd* socket.

Readers who are familiar with the serial interface will already know that a supply voltage can be obtained from this itself, which you can use to power the integrated circuit (serial mice do this for example.) This can be achieved by connecting RTS which provides a voltage level of 8 to 12 volts. This voltage can be reduced to the required 5 volts using resistors or, better, a voltage regulator. You need to be careful to keep the load on the interface low, even though a few milliamps are all that are needed. The serial interfaces of notebooks can cause problems as the voltages they provide may be too low to use.

Of course, the hardware is only half the story. Software is also required to enable the received signals to be decoded. For the Linux user, this is where LIRC can help. LIRC stands for Linux Infra-red Remote Control and provides a way to decode and implement actions as a result of infra-red signals received from remote controls. LIRC also provides functionality for the transmission of infra-red signals, though only at the driver level. Further software for this can be found in the *xrc* packages which you will find on the LIRC home page.

In earlier versions of LIRC only the serial type of receiver was supported. Since then there are now also kernel modules for several TV cards as well as the IrDA port. In the archive *lirc-0.6.0.tar.gz* all these drivers can be found in directory *drivers/*.

Using *./configure; make; make install*, the configuration and installation are carried out in the usual way. After entering *./configure* a dialog-based shell script should appear. Here you select the infra-red receiver type and where necessary adjust a few compile-time settings. The subsequent *make* should then generate the binaries. By running invoking *make install* as *root* you can now copy the files to the usual places under */usr/local/*. The entry:

```
alias char-major-61 lirc_driver.o
```

in */etc/conf.modules* ensures that the LIRC driver is linked automatically into the kernel through *kerneld/kmod* if necessary (»driver« here represents the actual driver name, for example <«serial« for *lirc_serial*).

The majority of infra-red receivers require their own driver in the form of a kernel module. (Fig. 5). Here the driver takes over direct communication with the hardware and hands over the received data to a daemon. This attempts to assign these signals again to a key on the remote control.
A further service program from the LIRC package allows other programs to be started, depending on the particular key that was pressed. I will go into that in detail later.

Let's take a closer look at a kernel driver for LIRC, taking *lirc_serial* as an example. Normally, this driver is loaded as a kernel module, hence the first function invoked is *init_module()*. There is, of course, a corresponding function that runs the kernel on removing the module, *cleanup_module()*.

First of all the code checks whether the I/O region of the serial interface is still available at all. If so, this is initialised. The *init_port()* function implements this for us.

In the next step, the *lirc_serial* module registers itself in the kernel as a character device via the call-up *register_chrdev()*. Here the *lirc_fops* structure contains the pointers to the individual functions that can be carried out on the device *(open, close, read, write, ioctl)*. Once all of this has been successfully completed, the driver can start its real work.

The *irq_handler()* plays a central role in all of this. It responds to interrupts triggered by a change in level on the DCD pin of the serial interface. Each time the interrupt handler is called it establishes whether it is dealing with a pulse or space and determines the time in microseconds since the last call. It combines these two values into an integer (a pulse/space-timestamp) and places it in a ring buffer which the function *lirc_read()* is able to read out. By this means the data passes from the kernel into the user space.

The TV card drivers use a different format. They are not supplied with time differences by the hardware but instead receive the bit codes ready for decoding. This means that *lircd* no longer has to deal with this conversion. With an *ioctl()* call, *lircd* ascertains the particular format that the driver delivers.

For more information about kernel module programming refer to the Linux Kernel Module Programming Guide.

As already mentioned, LIRC can also transmit infra-red signals. This also takes place via the serial interface, or more precisely, via DTR, with the aid of the *lirc_serial* driver. For this you have to compile it with *#define LIRC_SERIAL_TRANSMITTER*. As a rule though that is set up via *./configure*. For transmission we require three things: a carrier frequency, the serial data stream that contains the encoded bit code and an infra-red transmitting diode.

There are two possibilities here. First, you could output the modulated carrier frequency via DTR straight away and would need only to pass this signal to an infra-red transmitting diode. However, at 38 kHz it is difficult to get the exact timing. Second, you could output the serial data stream via DTR and have it modulated by hardware connected to it. The first option can be achieved by setting *#define LIRC_SERIAL_SOFTCARRIER* in *lirc_serial* (which is likewise settable via *./configure*). The transceiver example integrated circuits on the LIRC home page use this method.

If you look more closely at the source code of *lirc_serial* in the LIRC package (Listing 1), you will notice that only a warning is output if the I/O region is already allocated. This takes into account the fact that many users have compiled the generic serial driver into the kernel. It makes more sense to also compile the generic driver as a module and put *lirc_serial.o* before *serial.o* for »modproben«.

Listing 1: Some snippets of source code

```
int init_module(void)
{
  int result;

  if ((result = init_port()) < 0)
    return result;
  if (register_chrdev(major, LIRC_DRIVER_NAM↩
E, &lirc_fops) < 0)
    printk(KERN_ERR  LIRC_DRIVER_NAME {
           ": register_chrdev failed\n");
  release_region(port, 8);
  return -EIO;
  }
  return 0;
}

/*
Sections of the IRQ handler for lirc_serial
*/

void irq_handler(int i, void *blah, struct pt↩
_regs *regs)
{
  /* ... */
  do{
    counter++;
    status=sinp(UART_MSR);
    if((status&UART_MSR_DDCD) && sense!=-1)
    {
      /* get current time */
      do_gettimeofday(&tv);
      dcd=(status & UART_MSR_DCD) ? 1:0;
      deltv=tv.tv_sec-lasttv.tv_sec;

      /* ... */

      data=(lirc_t) (deltv*1000000+
           tv.tv_usec-
           lasttv.tv_usec);
             frbwrite(dcd^sense ? data : (data↩
|PULSE_BIT));
             lasttv=tv;
             wake_up_interruptible(&lirc_wait↩
_in);
      }
  } while(!(sinp(UART_IIR) & UART_IIR_NO_INT↩
)); /* still pending
? */
}
```

This inconspicuous while loop was not implemented in older versions of the driver, which had the effect that under unfavourable conditions the driver no longer executed its *irq_handler()*. This could only be overcome by removing and reloading the kernel module.

As shown in listing 2, the Hauppauge TV card driver reads the received data. Note that the format of the data is different from that supplied by *lirc_serial*. *lircd* uses *ioctl()* to determine the data format in question.

The interface between kernel driver and user space applications is the device file */dev/lirc*, which *make install* creates with *mknod /dev/lirc c 61 0*. The receiver types that are driven via */dev/ttySx* constitute an exception here. The program *mode2* from the directory *tools/* of the LIRC package is a simple example of how one communicates with a LIRC driver: open */dev/lirc* and read out integer values. The values read are to be interpreted differently depending on the particular type of driver: they are either pulse/space timestamps (serial drivers) or bit codes (TV card) – *mode2* recognises only the first type and continuously outputs all received pulses/spaces.

The LIRC daemon *lircd* is employed at this point to convert the received signals into a simple useable form. This daemon returns the names of the keys. For this, though, it also needs a configuration file containing the parameters of the remote control (this file is called *lircd.conf*, and it is expected in */usr/local/etc/*). If no suitable file can be found in directory *remotes/* you can create one yourself with *irrecord* (which is found in the *daemons/* directory normally installed below */usr/local/bin/*). The only parameter that you have to enter here is a filename into which the configuration file is to be saved.

Since */dev/lirc* normally has file mode 644 (which allows users only read access to the device) you must either start *irrecord* as root or change the file mode. Should *irrecord* complain with the message »Something went wrong« and not be able to generate a configuration file you can also specify the additional switch - - *force* on the command line to obtain a configuration file in RAW mode. This should work in any event.

It is worth taking a closer look at the file generated. First of all, it can be seen that all the data for the remote control is encapsulated in a „begin remote … end remote" block. This allows trouble-free placement together in one file of several such blocks that are to be used when analysing the received signals. In addition, the particulars of the remote control's timing parameters are very informative.

Since we now have a file in */usr/local/etc/* containing the relevant timing parameters of the remote control used, there is nothing to prevent a start of *lircd* (*daemons/, /usr/local/sbin/*). With the program *irw* (*tools/, /usr/local/bin/*) you can now check whether *lircd* is going about its work correctly. As soon as you press a couple of keys on the remote control, something similar to that shown in listing 3 should be displayed on the terminal (in this case with the previously mentioned Anir Multimedia Magic and the corresponding configuration file). There are always four parameters per line: 1 - bit code, 2 - repetition counter (if the key remains depressed for an extended period), 3 - name of the key, 4 - name of the remote control.

Listing 2: Read out of the received data

```
static __u16 read_raw_keypress(
struct lirc_haup_status *remote)
{ /* ... */
  /* Starting bus */
  i2c_start(t->bus);
  /* Resetting bus */
  i2c_sendbyte(t->bus, ir_read, 0);
  /* Read 1st byte: Toggle byte (192 or 224) */
  b1 = i2c_readbyte(t->bus, 0);
  /* Read 2nd byte: Key pressed by user */
  b2 = i2c_readbyte(t->bus, 0);
  /* Read 3rd byte: Firmware version */
  b3 = i2c_readbyte(t->bus, 1);
  /* Stopping bus */
  i2c_stop(t->bus);
  /* ... */
}
```

Listing 3: Parameters of Anir MultiMedia Magic

```
00000000005ba4f0 00 CD_UP ANIMAX
0000000000de21f0 00 RADIO_DOWN ANIMAX
0000000000de21f0 01 RADIO_DOWN ANIMAX
00000000005ea1f0 00 RADIO_UP ANIMAX
0000000000dc23f0 00 TV_DOWN ANIMAX
```

Hauppauge TV cards

Some manual intervention is necessary to coax the LIRC driver lirc_haup to work with Hauppauge TV cards (WinTV/Radio). A Red Hat 6.1 installation with kernel 2.2.15 and the package lirc-0.6.0 served as the base system for a test.

To start with, the kernel has to be compiled with support for BT848 (Character Devices->Video For Linux) and MSP3400 (Sound->Additional low level sound drivers) in order to be able to use the TV card at all. It should be compiled as a module, otherwise the addresses of the I2C functions are not exported and as a result are not available to kernel modules. In that case, loading the kernel module lirc_haup would fail.

As can be seen in the source code excerpt from read_raw_keypress, these functions are referenced. Before you compile the kernel, however, the file /usr/src/linux/include/linux/i2c.h still has to be patched: in line 99 you must replace the #if 0 with if 1. A patch file is available for this in the LIRC package under drivers/lirc_haup/patches/. If they are not already present you must also insert the following lines in /etc/conf.modules to allow the dynamic linking of the modules into the kernel through kmod:

```
alias char-major-61 lirc_haup.o
alias char-major-81-0 bttv.o
pre-install bttv modprobe msp3400.o
```

Once the kernel is more or less ready you can get on with the configuration of LIRC. Here you select "Hauppauge TV Card (old I2C Layer)" as the driver. Under contrib/ you can copy a start script for /etc/rc.d/init.d/ and set the corresponding S- and K-links in order to start lircd each time the system is booted. So far so good. After this step we found LIRC worked wonderfully but the TV tuner would apparently no longer respond. This problem occurred only if the BT848 support (bttv.o) was installed as a module. As soon as it was permanently compiled into the kernel the TV tuner worked faultlessly. The disadvantage was that LIRC no longer worked due to the missing I2C symbols. However, with a small modification in /usr/src/linux/drivers/char/i2c.c we were able to get this problem under control. Somewhere near to line 430 is the beginning of the EXPORT_SYMBOL block. This is bounded by an #ifdef MODULE … #endif that also includes the functions init_module() and cleanup_module(). If, using cut and paste, you move the line containing #ifdef MODULE to be ahead of these two functions, the I2C symbols can then be accessed from other kernel modules, although the bttv driver itself is not present as a module.

One further hint: when problems are experienced with the TV card drivers it is generally always worth checking out the latest CVS source tree, since there are usually more up to date versions of the driver to be found there.

Communication between the daemon and *irw* takes place via the socket */dev/lircd*. This allows a number of programs to connect to *lircd* simultaneously. The LIRC mouse daemon *lircmd* is one of them. As the name suggests, with this you can achieve the functions of a mouse via a remote control. The associated configuration file *lircmd.conf* is expected too, which like that for *lircd* should be placed in */usr/local/etc/*. A glance at the already existing configuration files in directory *remotes/* should give an idea of what the file should look like. A few points on this:

- PROTOCOL IMPS/2 ensures that the IMPS2 protocol runs via */dev/lircm*. If this directive is omitted, the Mouse Systems protocol is used.
- ACCELERATE controls the speed with which the mouse pointer moves.
- ACTIVATE defines the specific key via which the mouse functionality can be activated
- All MOVE_ instructions assign the directions.
- All BUTTON_ directives assign the mouse buttons
- Two parameters are always specified with ACTIVATE, MOVE_, BUTTON_, the first being the name of the remote control (»*« for all remote controls), the second is the key abbreviation that *lircd* supplies.

Listing 4: Simple example for *xmms*

```
# Output text each time that 1 is pressed

begin
        prog = irexec
        button = 1
        config = echo "You've pressed 1"

end

# Start xmms in the background and switch int➘
o xmms mode

begin
        prog = irexec
        button = RADIO
        config = xmms &
        mode = xmms
end

# All "begin ... end" blocks within "begin xm➘
ms ... end xmms"
# are taken into consideration only if we are➘
in mode = xmms.
# This makes sense if you intend switching be➘
tween a number of
# keypad layouts.
# Hint: if the mode is given the same name as➘
the programme then
# this is selected automatically.

begin xmms
        begin
                prog = xmms
                button = play
                config = PLAY
        end
end xmms
```

- Example: *gpm -m /dev/lircm -t msc*, for the case where the Mouse Systems protocol is used.

Once the kernel driver and daemon have processed the received raw data to the extent that the key abbreviation can be obtained directly from *lircd*, it is time to respond. For example, *irexec* does precisely this by starting various programs. As to be expected, the details as to what is to be started by which key abbreviation, are determined by a configuration file. This file is expected to be in *$HOME/.lircrc*. A few points on this too:

- The file is split into „begin … end“ blocks for each key abbreviation
- Within these blocks »prog = irexec« must appear first of all. This allows several programs to be configured via.lircrc. An entry with »prog = xawtv«, for example, would be ignored by irexec
- »button = BUTTON ABBREVIATION« for the relevant button that is to be responded to
- »config = Program« whatever is to be executed
- »repeat = 0« no response will be made to button repetition
- With »mode« a different »begin mode – end mode« block in the configuration file can be activated

To simplify communication with *lircd* and for easy read-in and processing of the file *.lircrc*, since lirc-0.6.0 there has been a library (*liblirc_client*) which is also used by *irexec*. Further information regarding the functional scope and linking of *liblirc_client* in standalone programs can be found in *doc/* in the LIRC package.

A program that also uses the *liblirc_client* and ought to be of particular interest to TV card owners is called *xawtv*. With *xawtv* remote viewing is possible both under X and with the frame-buffer device. Through the support of *liblirc_client* you can sit comfortably in the armchair and zip things with the remote control. For simplicity's sake *xawtv* includes its own *.lircrc* file with (*contrib/dot.lircrc.*). The key abbreviations that are to be found in the configuration file of the Hauppauge remote control are used there. Thus anyone who has assigned other key abbreviations needs to adjust them here.

The XMMS plug-in will also probably be found useful since with this *xmms* can be operated as conveniently as *xawtv*. LIRC and xmms should already be installed prior to compiling the plug-in, otherwise the necessary header files will not be found. An example *.lircrc* file is also enclosed and can be appended to one that already exists (if need be adapt the key abbreviations to suit your own remote control). In contrast to *xawtv* with the *xmms* plug-in *irexec* must run in the background as well. Use is made of "mode" in the configuration file and *xmms* itself is started only through *irexec*. This can be circumvented by commenting out the *irexec* entry along with the two lines "begin xmms" and "end xmms". ■

Info

LIRC home page:
http://mars.wiwi.uni-halle.de/lirc/

Linux Kernel Module Programming Guide:
http://www.fokus.gmd.de/linux/LDP/lkmpg-1.1.0.html/

Remote Control Tutorial:
http://cgl.bu.edu/GC/shammi/ir/

TSOP17XX data sheets:
http://www.vishay.de/datasheets/optoelectronics/photomodules/TSOP17…html

A self-built universal remote control:
http://www.geocities.com/SiliconValley/Sector/3863/uir/index.html

Animax:
http://www.animax.no/

xatw home page:
http://me.in-berlin.de/~kraxel/xawtv.html

IRman:
http://www.evation.com/