

10

COMMUNICATING WITH OTHER SURVIVORS



In Chapter 1, we discussed the pros and cons of teaming up with other humans when zombies walk the Earth.

Associating with other people can certainly be worthwhile: you can protect each other, share knowledge, pool resources, and so on. Of course, they can also take your stuff and put you between themselves and the oncoming zombies. If you decide to take the risk and reach out to your fellow life forms, then build the projects in this chapter.

First, we'll build a beacon to broadcast a voice signal that can be heard on an FM radio, so any survivors scanning the airwaves can hear your message, whether that's "Stay away!" or "Help, I'm trapped on the roof of a shopping mall!" After that, you'll also build a Morse code flasher that will blink out any message you care to translate into dots and dashes.

Of course, if you want to be the one scanning frequency bands, this chapter also explains how to hack a radio receiver to search for a signal. Then, you can lurk silently while you decide whether what's out there is worth broadcasting to (see Figure 10-1).



FIGURE 10-1: ZOMBIES LIKE THE RADIO TOO.

PROJECT 17: A RASPBERRY PI RADIO TRANSMITTER BEACON

The Raspberry Pi is a versatile device that can, given the right software, act as an FM radio transmitter. The only extra hardware you'll need is a length of wire to act as an antenna.

WHAT YOU WILL NEED

This is another Raspberry Pi project, so you will need to have a working Raspberry Pi system complete with keyboard, mouse, and screen as described in Chapter 5. Once the program that transmits the radio signal is up and running, you can turn off the screen to save power if you wish.

RADIO TRANSMITTER LEGALITY

IF YOU'RE READING THIS AFTER THE ZOMBIE APOCALYPSE, THERE WILL BE NO LEGAL PROBLEMS WITH BUILDING A TRANSMITTER BECAUSE THERE WON'T BE ANY GOVERNMENT TO ENFORCE THE REGULATIONS. IF, HOWEVER, YOU ARE BUILDING IN PREPARATION, THEN THE LEGALITY OF THE TRANSMITTER IN THIS PROJECT IS COVERED BY THE SAME LEGISLATION AS FM TRANSMITTERS DESIGNED TO BE CONNECTED TO AN MP3 PLAYER FOR CAR AUDIO.

THESE TRANSMITTERS ARE LEGAL IN THE UNITED STATES IF THE EFFECTIVE RANGE IS 200 FEET (60 M) OR LESS. IF YOU USE A FULL-LENGTH ANTENNA, THIS TRANSMITTER WILL HAVE A LONGER RANGE THAN THAT, SO TO STAY WITHIN THE LAW, USE A SMALL ANTENNA OF ABOUT 3 OR 4 INCHES (7 TO 10 CM).

REGULATION OF THE AIRWAVES IS NECESSARY SO THE FREQUENCIES USED BY EMERGENCY SERVICES STAY CLEAR, BUT THIS TRANSMITTER USES ONLY THE PUBLIC BROADCAST FM WAVE BAND. THE WORST THAT CAN HAPPEN IS ONE OF YOUR NEIGHBORS RECEIVES YOUR BROADCAST INSTEAD OF THEIR FAVORITE RADIO STATION.

To build this radio transmitter, you'll need the following parts:

ITEMS	NOTES	SOURCE
☐ Raspberry Pi	Raspberry Pi 2, Model B or B+	Adafruit (2358), Fry's (8258726)
☐ Jumper wire	Female-to-female jumper wire	Adafruit (826)
☐ Wire for the antenna	About 3 feet (1 m) of wire	

Any wire will do for the transmitter; just check your box of scavenged hookup wire for something that will fit into the end of the female-to-female jumper wire.

You could add the radio transmitter to your existing Raspberry Pi setup. However, for maximum transmission range, you'll want to put the transmitter somewhere high up, so I recommend getting a second Pi.

The length of the jumper wire doesn't matter; it just allows an easy connection between the Raspberry Pi GPIO pin and the antenna wire. The wire to use for the rest of the antenna should be the right size to poke into one end of the female-to-female jumper wire and stay there. You might need to put a kink in the antenna wire so that it stays in place.

CONSTRUCTION

To build your transmitter, all you need to do is plug one end of the jumper wire onto GPIO pin 4 of the Raspberry Pi (Figure 10-2), then plug the antenna wire into the other end of the jumper wire and fix the other end of the antenna to a high spot so that the antenna is pulled up vertically.

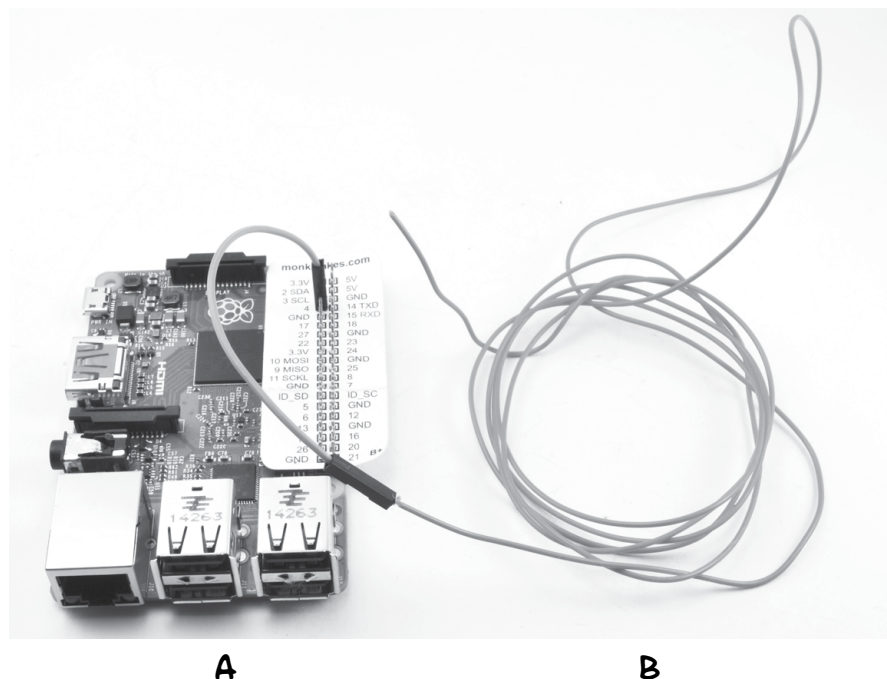


FIGURE 10-2: ATTACHING THE ANTENNA

You will get the longest transmission range if you place the whole Raspberry Pi up high. If you have a watchtower, this would be ideal.

It does not matter if the antenna wire is not very straight. You may find that some electrical tape wrapped around the junction of the antenna wire and the jumper wire will prevent the antenna from becoming detached. Once you've strengthened the antenna, you've built your radio transmitter beacon!

SOFTWARE

I wish I could claim credit for the wonderful piece of software you'll use in this project, but as it was developed by those clever folk at the Imperial College Robotics Society, I can't. You can find out all about their project at http://www.icrobotics.co.uk/wiki/index.php/Turning_the_Raspberry_Pi_Into_an_FM_Transmitter.

The software uses a sound file to oscillate GPIO pin 4 in just the right way to generate an FM carrier wave and signal (see the box on frequency modulation).

To install the software, start an LXTerminal session on your Raspberry Pi and type the following commands:

```
$ mkdir pifm
$ cd pifm
$ wget http://www.icrobotics.co.uk/wiki/images/c/c3/Pifm.tar.gz
$ tar -xzf Pifm.tar.gz
```

These commands create a directory ready to install the software, download the software using the `wget` utility, and then uncompress the downloaded file into the newly created directory.

USING THE FM TRANSMITTER

To test out the FM transmitter, you need an FM receiver (see “Project 18: Arduino FM Radio Frequency Hopper” on page 188). You also need to find an unused frequency, or at least a frequency with only a faint signal. Of course, this won’t be a problem following the apocalypse, but it’s more of a challenge with the crowded preapocalypse airwaves. Use your FM receiver to find a quiet part of the spectrum and make a note of the frequency.

The software you installed includes a sound sample of the *Star Wars* theme for testing the transmitter before you record your own, more appropriate message—although the music is not completely inappropriate to accompany humanity’s great battle to save itself.

In the LXTerminal, issue the following command to play the tune over your transmitter:

```
$ sudo ./pifm sound.wav 103.0
```

In place of `103.0`, substitute the frequency that your radio receiver is tuned to.

RECORDING A MESSAGE

To record a message, you’ll need a laptop and some sound-recording or editing software. I recommend Audacity, which is available free for Windows, OS X, and Linux from <http://audacityteam.org/>.

Fiction and history both tell us that when law and order disintegrate, bad behavior often follows. So think long and hard about what you want to say in your message. Who knows what gun-toting, supply-stealing outlaws

are lurking around the corner? You'll probably want to direct new arrivals somewhere you can observe them before lowering your defenses, so bear this in mind when recording your broadcast.

The pifm software requires you to record your message with the sample rate set at 16 bit 44.1kHz and then export the message as a WAV file. In the software, change `sound.wav` to the name of your new sound file, say `my_message.wav`.

FREQUENCY MODULATION

FREQUENCY MODULATION, OR FM AS IT IS NEARLY ALWAYS CALLED, IS A WAY OF ENCODING A SIGNAL (IN THIS CASE A LOW-FREQUENCY SOUND SIGNAL) ON A MUCH HIGHER CARRIER FREQUENCY. THE SOUND SIGNAL NUDGES THE CARRIER FREQUENCY HIGHER OR LOWER THAN THE CARRIER FREQUENCY, DEPENDING ON THE LEVEL OF YOUR MESSAGE SIGNAL'S WAVEFORM.

FIGURE 10-3 SHOWS TWO CYCLES OF THE MESSAGE SIGNAL (SOLID LINE) SUPERIMPOSED ON THE MUCH HIGHER FREQUENCY CARRIER TO CREATE THE BROADCAST SIGNAL (DOTTED LINE), WHOSE FREQUENCY CHANGES AS YOUR MESSAGE SIGNAL CHANGES.

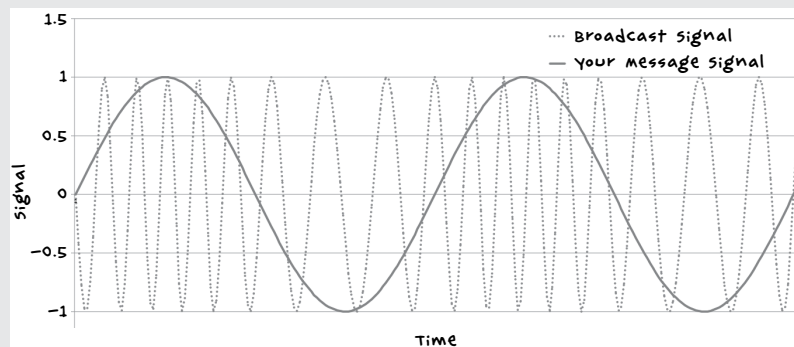


FIGURE 10-3: FREQUENCY MODULATION

WHEN THE SIGNAL IS AT ITS MAXIMUM, THE PEAKS OF THE DOTTED BROADCAST SIGNAL ARE CLOSEST TOGETHER. THAT MEANS THE FREQUENCY IS HIGHER THAN AVERAGE. AT THE BOTTOM OF THE WAVEFORM, WHEN THE SIGNAL HAS ITS MINIMUM VALUE, THE BROADCAST SIGNAL PEAKS ARE FARTHEST APART (THE FREQUENCY IS LOWER THAN AVERAGE).

IN THIS WAY, THE LOW-FREQUENCY SOUND WAVE IS ENCODED ONTO THE HIGH-FREQUENCY CARRIER WAVE. WHEN THIS SIGNAL GETS TO AN FM RADIO RECEIVER, THE CIRCUITRY IN THE RECEIVER EXTRACTS THE ORIGINAL LOW-FREQUENCY AUDIO FROM THE CARRIER SIGNAL.

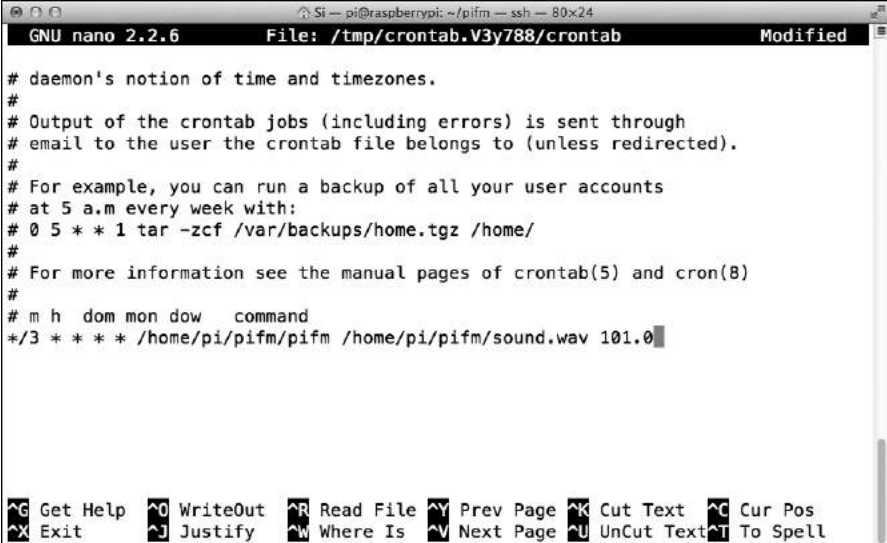
RUNNING THE TRANSMITTER AUTOMATICALLY

To maximize the chance of other survivors discovering your message, repeat this broadcast around the clock. You can configure the Raspberry Pi to do this for you automatically using a Linux tool called `crontab`. The `crontab` utility lets you schedule programs to run at certain times of day.

Enter the following command into the LXTerminal:

```
$ sudo crontab -e
```

This will open a configuration file with the nano editor, as shown in Figure 10-4.



```
GNU nano 2.2.6 File: /tmp/crontab.V3y788/crontab Modified
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h dom mon dow   command
*/3 * * * * /home/pi/pifm/pifm /home/pi/pifm/sound.wav 101.0

^G Get Help ^O WriteOut ^R Read File ^V Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^N Next Page ^U UnCut Text ^T To Spell
```

FIGURE 10-4: SCHEDULING YOUR BROADCASTS

Scroll down to the end of the file and add the following line:

```
*/3 * * * * /home/pi/pifm/pifm /home/pi/pifm/sound.wav 101.0
```

The first part of the line (`*/3`) schedules the transmission to run every 3 minutes, 24 hours a day, 7 days a week. If you use a different sound file or frequency, you need to replace `sound.wav` with your filename and enter your chosen frequency. If your message is longer than 3 minutes, change `*/3` to the number of minutes you need it to be.

You only need to do this configuration once; the settings will stick even if the Pi is rebooted.

PROJECT 18: ARDUINO FM RADIO FREQUENCY HOPPER

After the zombie apocalypse strikes, your chances of survival will be increased by group living—that is, assuming no bite victims come inside and turn into zombies. Always be sure that everyone gets checked for zombie-infected wounds before you grant entry!

You'll inevitably need to sleep or go on supply runs, and without someone to watch your back you'll be vulnerable. (Not to mention the slow descent into insanity you'll suffer from lack of human contact—and you thought zombies were crazy.) Therefore, you'll likely benefit from having a few companions around. Other groups of survivors may already be trying to make contact by broadcasting their own radio messages, as we now are. In fact, another group might have bought or salvaged this book and made the FM transmitter of Project 17. To find them, you just need to be able to pick up their transmission.

This project (Figure 10-5) takes a cheap FM receiver and hacks it so that it automatically scans the FM band for the next station. If someone has started transmitting on FM, creating a station instead of the hiss of empty airwaves, you will hear their broadcast. An Arduino simulates the pressing of the tune button on the radio receiver.



FIGURE 10-5: FM RADIO FREQUENCY HOPPER

WHAT YOU WILL NEED

To make this project, you will need the following parts:

<i>ITEMS</i>	<i>NOTES</i>	<i>SOURCE</i>
☐ Arduino	Arduino Uno R3	Adafruit, Fry's (7224833), Sparkfun
☐ FM radio	Simple low-cost FM headphone radio	Dollar store (or equivalently named establishment in your country's currency)
☐ Powered speaker		Electronics store
☐ Audio lead (aux lead)	To connect the radio to the powered speaker	
☐ Red LEDs	2 red LEDs	Adafruit (297)
☐ Barrel jack plug	DC power jack with flying leads, 12V cigarette lighter adapter, or 5V USB adaptor and lead	Adafruit (80), eBay
☐ Right-angle header pins	12-way right-angle header pins	eBay

We are using right-angle pins rather than straight header pins as right-angle pins make it a little easier to solder wires and component leads to this project.

Look for an FM radio that has a Tune button that moves from one station to the next and a Reset button that starts from the beginning of the FM wave band. The radio I used cost less than \$2, including in-ear headphones.

The Arduino and speakers both require power. Although I have suggested using the barrel jack, you could just as easily use the USB port to power the Arduino. By now, you should be used to figuring out the most convenient way to power low-voltage devices from a 12V battery.

CONSTRUCTION

This project assumes the radio uses an SC1088 integrated circuit. This extremely low-cost chip is used in most very cheap radios, which seem to use the reference design specified in the datasheet for the chip. (Just search for "SC1088 datasheet" online; you should turn up a PDF in the first few results.) The wiring diagram is shown in Figure 10-6. It shows the Arduino being powered from the DC jack, but it could equally well be powered by the USB port.

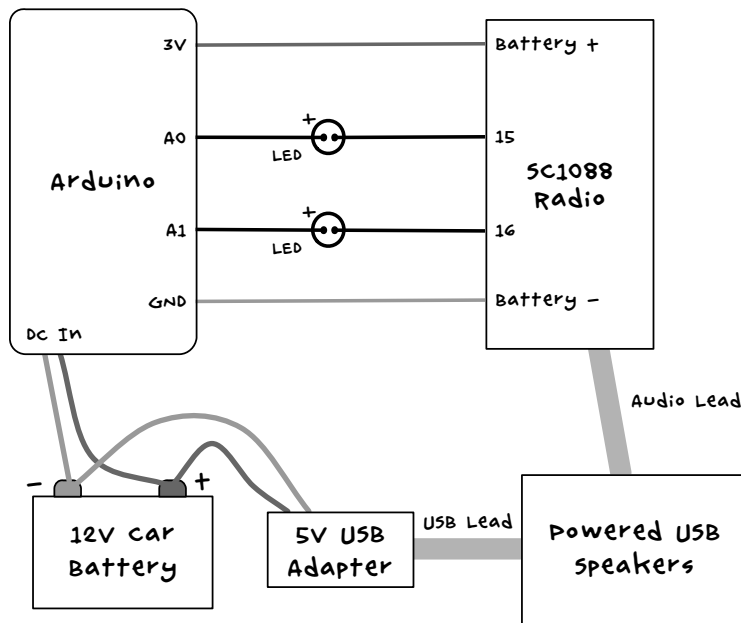


FIGURE 10-6: RADIO SCANNER WIRING DIAGRAM. THE NUMBERS 15 AND 16 ON THE SC1088 RADIO INDICATE PIN NUMBERS OF THE CHIP.

The “tune” and “reset” pins of the SC1088 IC are designed to be connected to momentary pushbuttons that short these pins to the chip’s 3V supply rail. You can see this configuration in the datasheet’s reference schematic. When pushbuttons are not shorting the input pins to the supply rail, they are pulled down to ground by variable resistances that are set inside the chip. We can emulate the functionality of the pushbutton by connecting these pins to ~3V when we want to simulate a button push, and by leaving the pin *floating* (not being driven high or low) when we want to simulate a button waiting to be pressed. To make the pin float, we can set the Arduino pin that is driving it to an input. When acting as an input, an I/O pin is said to be *high impedance*, meaning that the pin looks like an open circuit to anything that is attached to it.

To convert the 5V of the Arduino output pins to 3V, we place red LEDs between the Arduino pin and the SC1088. These drop the 5V to about 3.3V, the same level as supplied to the chip. The LEDs will also glow very slightly when activated, letting you know when the project is in operation.

STEP 1: DISASSEMBLE THE RADIO

First, take the radio apart. How to do this will depend on how your radio is put together. For mine, I just undid two screws and the whole thing came apart. Figure 10-7a shows the radio in its original state and 10-7b after removal of the case.

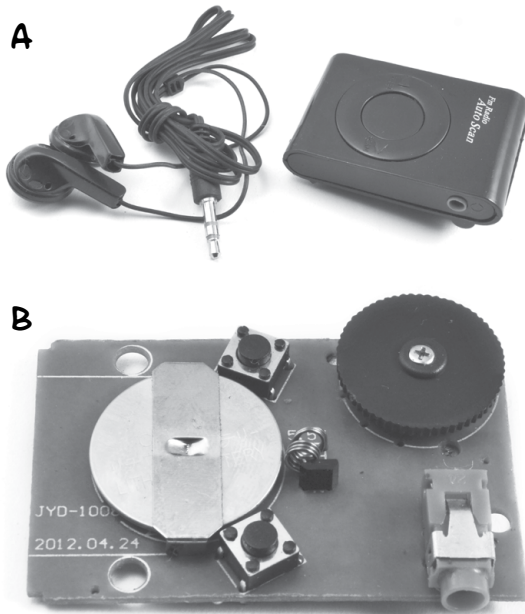


FIGURE 10-7: TAKING THE RADIO APART

Take the button cell battery out because we are going to use the Arduino to supply power to the radio.

STEP 2: IDENTIFY THE CONNECTION POINTS

Now we need to identify the points where we need to attach wires and LED leads. Figure 10-8 shows the underside of the radio's circuit board.

Start by identifying the location of the Scan and Reset switches. The pins for these will form a rectangle. The pins are connected in pairs, so both of the solder points labeled *A* are actually connected, as are the pair of points labeled *B*.

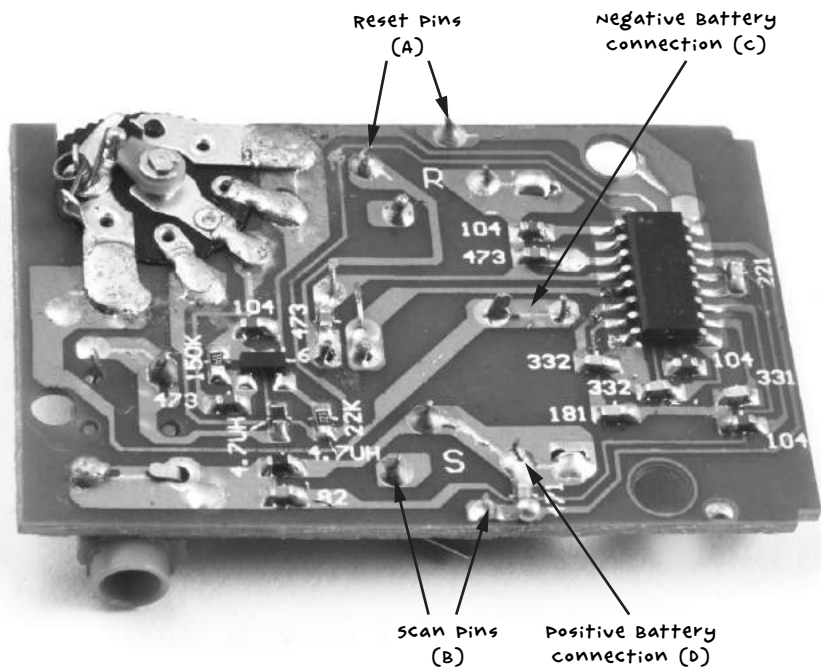


FIGURE 10-8: THE RADIO PCB

The A connections are for the Reset button. If you follow the track on the PCB, you will see that one of the A pins connects to pin 16 of the SC1088 (IC pins are numbered 1 to 16 counterclockwise, with a little dot on the IC package next to pin 1).

Following the track from B, you can see that one pin connects to pin 15 of the SC1088. This is the connection that we will use to scan for the next station.

If you're finding it hard to see where the tracks run, use your multimeter set to continuity mode to identify the pins. Press one probe to the IC pin you want to find a connection for (15 or 16) and then try the different likely connections on the switches with the other probe until the buzzer on the multimeter sounds.

Next, find the two connections needed to power the radio from the Arduino, which correspond to the battery holder connections on the PCB. The 3V batteries the radio takes have a negative central connection (C) and positive connections to the outside frame of the battery holder (D).

STEP 3: ATTACH THE HEADER STRIP

I have suggested a right-angle header strip here, because it's easier to solder the wires to, but regular header pins work almost as well. Break off a length of 12 pins and attach them to the Arduino pins 3.3V through to A5 (Figure 10-9). One pin will sit between the two header sockets, unconnected to anything.

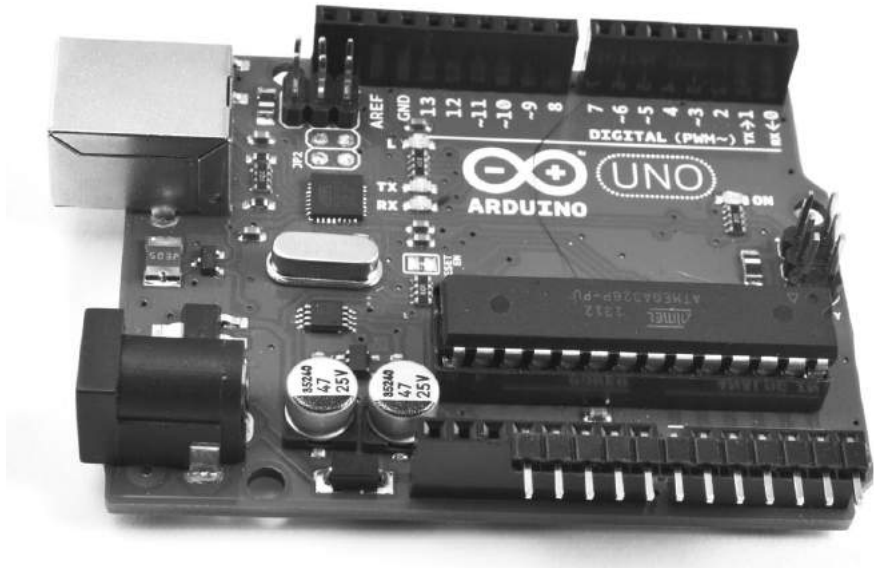


FIGURE 10-9: THE ARDUINO HEADER PINS

STEP 4: LINK THE RADIO TO THE ARDUINO

Figure 10-10 shows the radio connected to the Arduino. Use short wires to connect the 3.3V Arduino pin to the positive battery connection, point D, that you identified earlier. Connect an Arduino GND connection (it doesn't matter which one) to point C, the negative battery connection. Connect the positive (longer) lead of one LED to Arduino pin A0 and the negative lead of that same LED to point B. Do the same with another LED to Arduino pin A1 and point A on the radio PCB.

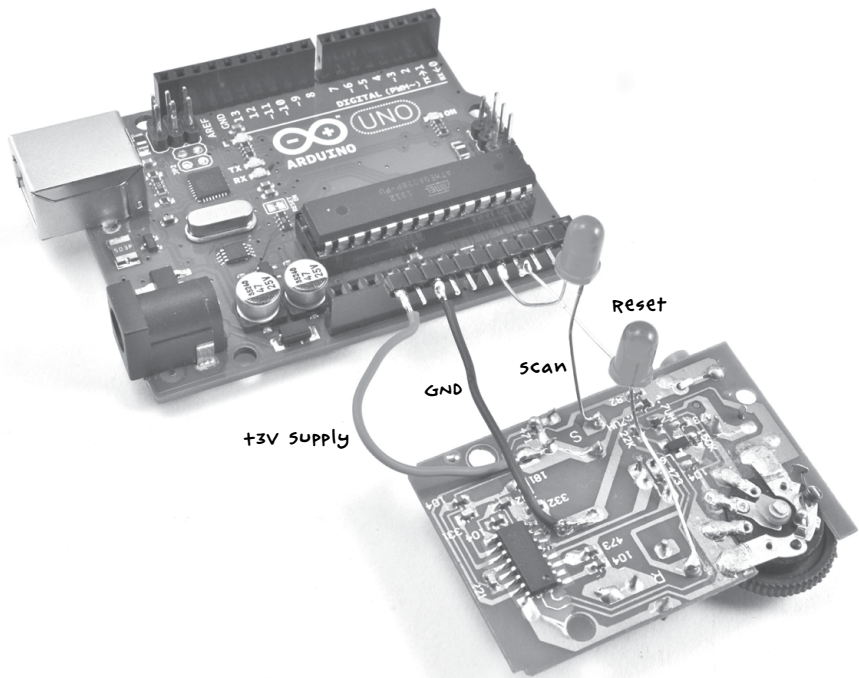


FIGURE 10-10: THE ARDUINO CONNECTED TO THE RADIO

STEP 5: CONNECT EVERYTHING TOGETHER

Finally, plug the powered speakers into the radio's audio jack. You can test this using the headphones first. The radio uses headphones or an audio lead as an antenna, so you may get better results with a longer lead of a few feet than with a very short lead.

SOFTWARE

All the source code for this book is available from <http://www.nostarch.com/zombies/>. See Appendix C for instructions on installing the Arduino sketch.

The Arduino sketch for this project is called *Project_18_Scanner*, and I'll walk you through it now.

The sketch starts by defining several constants:

```
const int scanPin = A0;
const int resetPin = A1; const int pulseLength = 1000;
const int period = 5000;
const int numStations = 5;
```

The `scanPin` and `resetPin` constants define the two Arduino pins we'll use, and `pulseLength` defines the length of the simulated button press. The scan buttons needs to be pressed for a full 1,000 milliseconds (1 second) for the radio to scan for the next station rather than simply move the frequency up a step, though this can vary depending on your radio.

The constant `period` tells the Arduino an amount of time, in milliseconds, to pause so you have time to register whether you are hearing a transmission or just white noise.

Next, we define a single global variable:

```
int count = 0;
```

This variable, called `count`, is used to keep track of the number of scans to make before resetting to the start of the FM band again.

The `setup` function initializes both pins as inputs (although as we shall see, this sketch is unusual in that it changes the pin mode of the pins after their first initialization).

```
void setup()
{
  pinMode(scanPin, INPUT);
  pinMode(resetPin, INPUT);
}
```

The `loop` function is where we actually scan for frequencies:

```
void loop()
{
  delay(period);
  pinMode(scanPin, OUTPUT);
  digitalWrite(scanPin, HIGH);
  delay(pulseLength);
  pinMode(scanPin, INPUT);
  count ++;
  if (count == numStations)
  {
    count = 0;
    pinMode(resetPin, OUTPUT);
    digitalWrite(resetPin, HIGH);
    delay(pulseLength);
    pinMode(resetPin, INPUT);
  }
}
```

First of all, the loop delays by the time specified in `period`. The function then sends a pulse to the scan pin to begin scanning. When the pulse has finished, the pin is set back as an input.

The count variable then increments, and when it has reached the maximum specified in `numStations`, a pulse is sent to the reset pin to start scanning from the beginning of the FM band again. During testing, setting `numStations` to 5 will allow you to check whether the project is working and finding different stations. However, after a zombie apocalypse, the airwaves should be pretty empty, so you may want to reduce this number to just 1, as any signal you happen across is bound to be transmitted by survivors (or perhaps smart zombies). If you discover any automated transmissions you want to ignore, like a distress beacon from your former boss or the murmurings of zombies inexplicably learning the rudiments of human language, change `numStations` to a value of one more than the number of stations you want to ignore.

USING THE RADIO SCANNER

When you first turn everything on, you should hear static. After five seconds or so, the scan LED will glow very dimly, and the radio will scan for its first station. After five more seconds, it will move on to the next station, and so on, until you identify a human friend. Remember: safety in numbers—not hordes.

PROJECT 19: ARDUINO MORSE CODE BEACON

Morse code is a 19th-century invention that allows you to send messages using a series of long or short pulses of light or sound. Each letter of the alphabet is made up of dots and dashes, where a dot is a short pulse and a dash is a long pulse (three times longer than a dot). For example, the letter *z* is represented as this:

z
--..

And the word *zombie* would be this:

zombie
--.. --- -- -... .. .

Morse code uses shorter sequences of dashes and dots for the more commonly used letters, so *e*, as the most common letter used in the English language, is just a single dot. If you are interested, you can search online for the complete Morse code, though the software in this project will translate your message into Morse code for you. Take a look at the code for a table of Morse codes.

This Arduino-based project uses 12V LED lamps, like those you used back in “Project 3: LED Lighting” on page 49, to flash a message to any other survivors in visual range. It’s especially effective at night. Figure 10-11 shows the finished project.

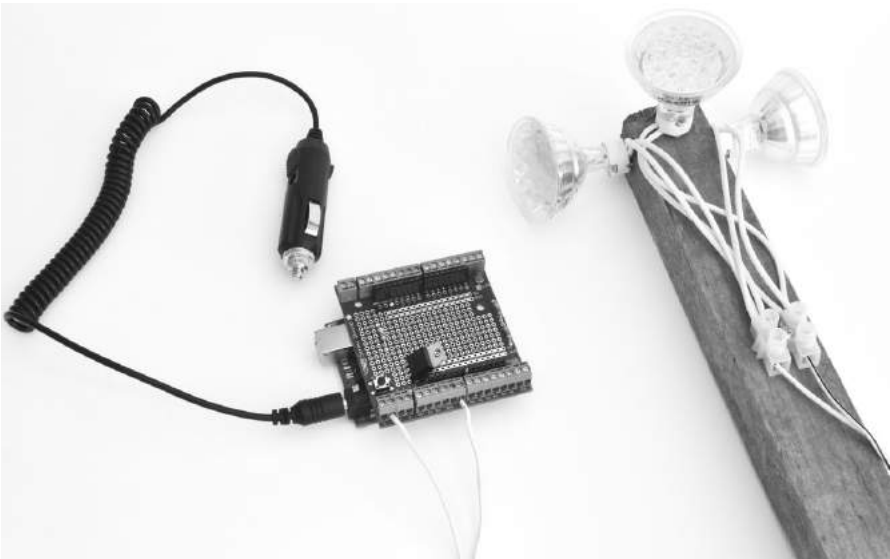


FIGURE 10-11: A MORSE CODE BEACON

WHAT YOU WILL NEED

To make this project, you will need the following parts:

ITEMS	NOTES	SOURCE
☐ Arduino	Arduino UNO R3	Adafruit, Fry's (7224833), SparkFun
☐ screwshield		Adafruit (196)
☐ 1 kΩ resistor		Mouser (293-1k-RC)
☐ MOSFET	FQP33N10 MOSFET	Adafruit (355)
☐ MR16 LED lamps	12V 3W	Hardware store
☐ MR16 lamp sockets	sockets with trailing leads	Hardware store
☐ Terminal block	2-way terminal block	Home Depot, Lowe's, Menards
☐ 9V Arduino battery lead	DC power jack with flying leads or 12V cigarette lighter adapter	DC power supply
☐ Wire	Bell cable (or other cable)	

It is best to use a fresh Arduino and screwshield for this project, both because it will be situated away from your main setup and because your screwshield from previous projects is probably pretty full by now. This project will be powered by its own solar power supply and battery (refer to “Project 1: Solar Recharging” on page 26).

I used three LED lights, but if you want more lamps, just add more in parallel. The transistor used to switch the lights is capable of switching up to 20W of lighting but only with a heatsink, so your combined wattage should be kept below 10W. If you made “Project 3: LED Lighting” on page 49, I would just use the same LEDs.

CONSTRUCTION

The layout for the screwshield and wiring schematic are shown in Figure 10-12.

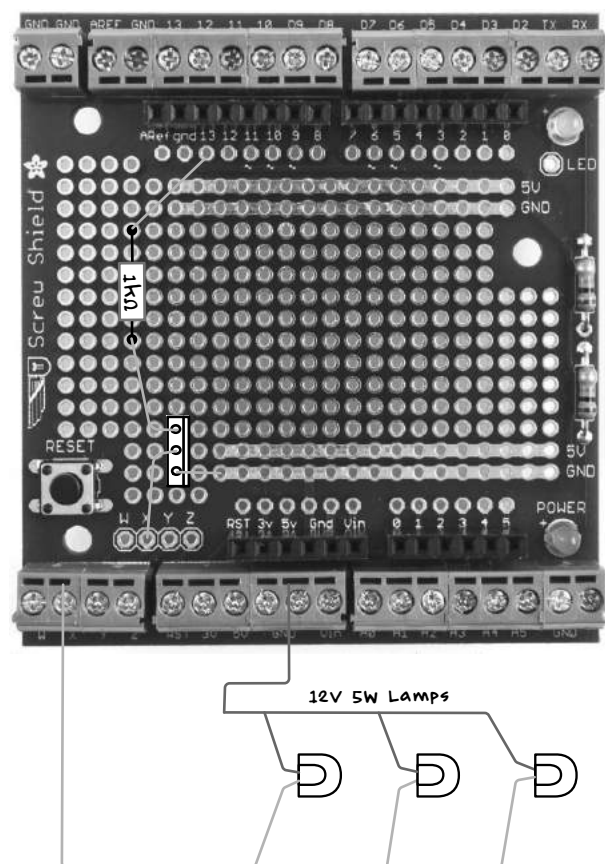


FIGURE 10-12: SCREWSHIELD LAYOUT AND WIRING SCHEMATIC FOR THE MORSE CODE BEACON

STEP 1: ASSEMBLE THE SCREWSHIELD

Assemble the screwshield following the instructions in “Assembling a Screwshield” on page 259.

STEP 2: SOLDER THE COMPONENTS ONTO THE SCREWSHIELD

You only need to solder two components for this project: a resistor and metal oxide semiconductor field effect transistor (MOSFET). MOSFETs are great for switching fairly high-power loads quickly.

Solder the resistor and transistor in place according to the circuit schematic. When soldering the transistor, make sure you place it so that the metal tab faces to the right (Figure 10-12). When the components are soldered into place, the assembly should look like Figure 10-13.

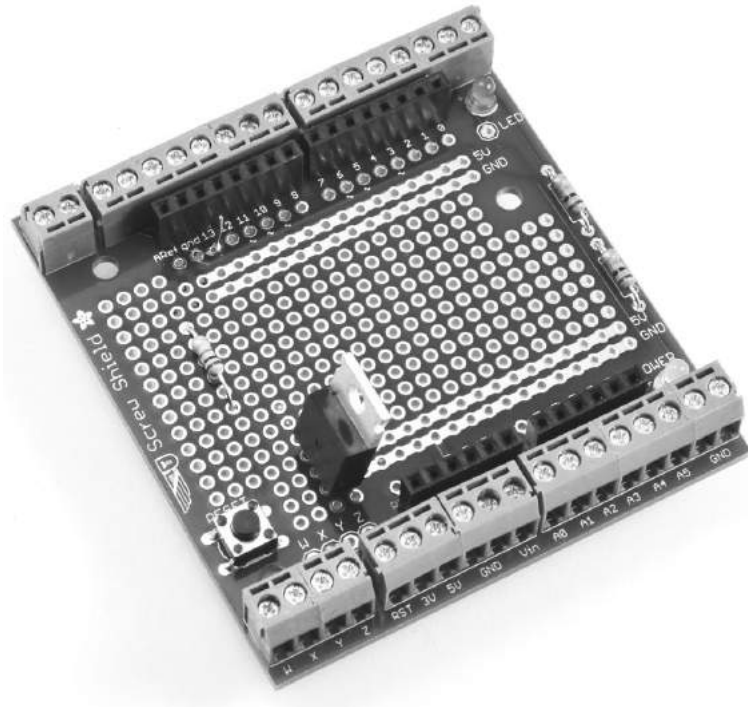


FIGURE 10-13: THE TOP OF THE SCREWSHIELD

STEP 3: WIRE THE UNDERSIDE OF THE SCREWSHIELD

Once the components are secured in place, use their excess leads to make the connections on the underside (Figure 10-14). Before soldering the resistor lead that connects to pin 13 on the Arduino, add some insulation to avoid causing short circuits with the 5V and GND tracks it crosses over.

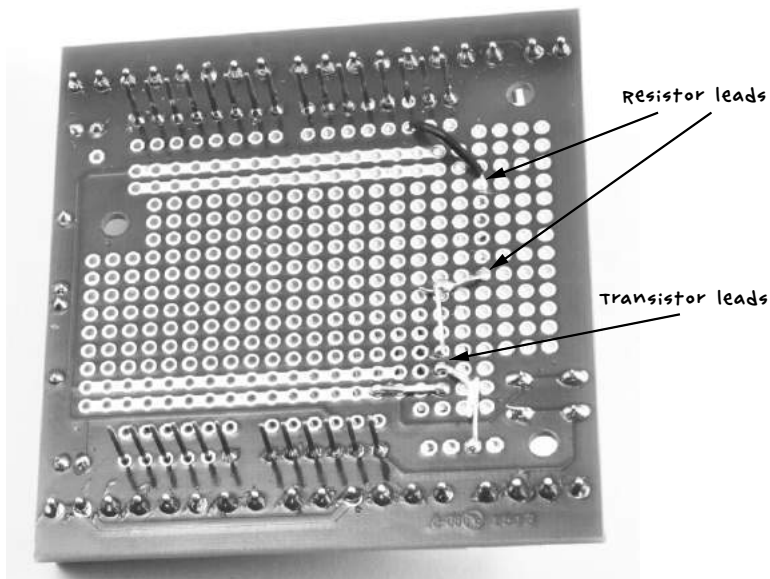


FIGURE 10-14: THE UNDERSIDE OF THE SCREWSHIELD

STEP 4: CONNECT THE LAMPS

If you want to keep this simple, you can just use a single LED lamp. For a wider range of visibility, however, connect a few LED lamps and point them in different directions (Figure 10-15).



FIGURE 10-15: THE LAMP ASSEMBLY

In Figure 10-15, I've fixed three lamp sockets to a bit of wood and connected all three 12V LED lamps to the terminal block. Lamps of this type usually include a circuit that allows the wires to be connected any way around, but if your modules have a polarity marked on them with a + and –, you need to make sure all the + connections are connected to one terminal of the terminal block and the – connections go to the other. The lamp holders will have holes allowing them to be attached to the wood with screws.

STEP 5: FINAL WIRING

Use some bell cable or other wire to connect the lamp assembly to the X and V_{in} terminals on the screwshield. Stranded wire is best, as it's less liable to break. Make this wire as long as you need it (but above 50 ft, or 15 m, there might be some reduction in brightness): you may want to site the lamp assembly high up outside, to make it easier for people to see your message, while leaving the Arduino in the safety of your bunker. Remember to waterproof the lamp assembly—sealing it in a transparent plastic bag will do the trick.

To connect power to the Arduino, use either a cigarette lighter adapter or a custom lead using alligator clips and a barrel jack plug with flying leads to connect the Arduino to a 12V solar power supply or battery. Note that this project requires 12V for the lamps, so you cannot use a 5V USB lead to power the Arduino.

SOFTWARE

All the source code for this book is available via <http://www.nostarch.com/zombies/>. See Appendix C for instructions on installing the Arduino sketch. The Arduino sketch for this project is called *Project_19_Morse_Beacon*.

The sketch uses the Arduino's built-in *EEPROM* library. The Morse code message is stored in EEPROM memory every time a change is made, meaning that the beacon can remember the message even if power to the Arduino is interrupted. The sketch also makes use of a library from the Arduino community called *EEPROMAnything*, which makes saving to and reading from EEPROM easier. The code for *EEPROMAnything* is included in the download for this project, so there is nothing to download separately.

First, we load both the official Arduino EEPROM library and *EEPROMAnything*:

```
#include <EEPROM.h>
#include "EEPROMAnything.h"
```

A number of constants are used to control the project:

```
const int ledPin = 13;
const int dotDelay = 100; // milliseconds
const int gapBetweenRepeats = 10; // seconds
const int maxMessageLen = 255;
```

The pin that controls the LEDs is specified in `ledPin`. The constant `dotDelay` defines in milliseconds the duration of a dot flash. Dashes are always three times the duration of a dot.

The constant `gapBetweenRepeats` specifies in seconds the time that will elapse between each repetition of the message, and `maxMessageLen` specifies the maximum length, in letters rather than dots and dashes, of the message. A maximum size is specified because in Arduino code, you have to declare the size of arrays.

Two global variables are used:

```
char message[maxMessageLen];
long lastFlashTime = 0;
```

The message variable will contain the text of the message to be flashed, and `lastFlashTime` keeps track of when the message was last flashed, to allow a break between the repeats.

Two global char arrays are used to contain the dot and dash sequences for Morse code. The program will only flash characters that it knows how to send, that is letters, digits, or a space character. All other characters in the message are ignored.

```
char* letters[] = {
    ".-", "-...", "-.-.", "-..", ".", "-.-.", "--.", "....", "..", // A-I
    "....", "-.-", ".-..", "--", "-.", "---", "-.-.", "--.-", ".-.", // J-R
    "...", "-", "...", "...", "---", "-.-", "--.-", "--.." // S-Z
};

char* numbers[] = {"-----", ".----", "-....", "...--", "....-", ".....",
    "-....", "--...", "---..", "----."};
```

The setup function sets the `ledPin` as an output and then starts serial communication at `Serial.begin`:

```
void setup()
{
    pinMode(ledPin, OUTPUT);
    Serial.begin(9600);
    Serial.println("Ready");
    EEPROM_readAnything(0, message);
}
```

```

if (! isalnum(message[0]))
{
    strcpy(message, "SOS");
}
flashMessage();
}

```

Serial communication is used to set a new message, either using the serial monitor of the Arduino IDE or, as you will see in “Using the Morse Beacon” on page 205, a terminal program running on a Raspberry Pi.

Every time the message is changed, it is saved in EEPROM, so during the setup process, the sketch reads any stored message from EEPROM. If no message has been set, the if statement in setup sets the default message to “SOS.” Finally, at flashmessage, the setup function flashes the message for the first time.

The loop function first checks whether a new message has been sent over the serial connection:

```

void loop()
{
    if (Serial.available())      // Is there anything to be read from USB?
    {
        int n = Serial.readBytesUntil('\n', message, maxMessageLen-1);
        message[n] = '\0';
        EEPROM_writeAnything(0, message);
        Serial.println(message);
        flashMessage();
    }
    if (millis() > lastFlashTime + gapBetweenRepeats * 1000L)
    {
        flashMessage();
    }
}

```

Any new message is read into the message character array until the new-line character (\n) is read. The null character '\0' is added to the end of the message. This is the Arduino’s way of indicating the end of a string of characters. Once the whole message has been read through, it is saved into EEPROM (EEPROM_writeAnything), and then the new message begins flashing immediately.

The remainder of the loop function checks whether enough time has passed before it can repeat the message. This could be done more simply using delay, but we would be unable to interrupt the loop if a new message arrived during the delay.

The `flashMessage` function is the most complex function in the sketch.

```
void flashMessage()
{
  Serial.print("Sending: ");
  Serial.println(message);
  int i = 0;
  while (message[i] != '\0' && i < maxMessageLen)
  {
    if (Serial.available()) return; // new message
    char ch = message[i];
    i++;
    if (ch >= 'a' && ch <= 'z')
    {
      flashSequence(letters[ch - 'a']);
    }
    else if (ch >= 'A' && ch <= 'Z')
    {
      flashSequence(letters[ch - 'A']);
    }
    else if (ch >= '0' && ch <= '9')
    {
      flashSequence(numbers[ch - '0']);
    }
    else if (ch == ' ')
    {
      delay(dotDelay * 4); // gap between words
    }
  }
  lastFlashTime = millis();
}
```

The `flashMessage` function starts by echoing the message it is about to send to reassure you that it is sending what you want it to. It then loops over every character in the message. Before each character, it uses `Serial.available` to check for a new message. If a new message has come in, the function stops sending its message in order to receive the new message from your computer or Raspberry Pi; then it begins sending the new message instead.

The `flashMessage` function determines whether the character is an uppercase letter, a lowercase letter, a number, or the space character and then takes the appropriate action.

If the character is a lowercase letter, the index position of the sequence of dots and dashes held in the `letters` array is provided as a parameter to the `flashSequence` function, which then flashes those dots and dashes. The other options are handled in the same way.

Finally, when the whole message has been sent, the `lastFlashTime` variable is set to the current time so the loop function can work out when it is time to start flashing the message again.

The work of flashing the sequence of dots and dashes for a particular character is handled by the `flashSequence` function:

```
void flashSequence(char* sequence)
{
    int i = 0;
    while (sequence[i] != NULL)
    {
        flashDotOrDash(sequence[i]);
        i++;
    }
    delay(dotDelay * 3);    // gap between letters
}
```

This loops over each dot or dash, calling `flashDotOrDash`:

```
void flashDotOrDash(char dotOrDash)
{
    digitalWrite(ledPin, HIGH);
    if (dotOrDash == '.')
    {
        delay(dotDelay);
    }
    else // must be a -
    {
        delay(dotDelay * 3);
    }
    digitalWrite(ledPin, LOW);
    delay(dotDelay); // gap between flashes
}
```

The `flashDotOrDash` function uses the appropriate delay period to flash a dot or dash.

USING THE MORSE BEACON

Upload the sketch to your Arduino and power up the project. The default message should start to flash. If it doesn't, go back and check over all your wiring. To change the message, attach your Arduino to your computer, open the serial monitor on the Arduino IDE, and type in a new message (Figure 10-16).

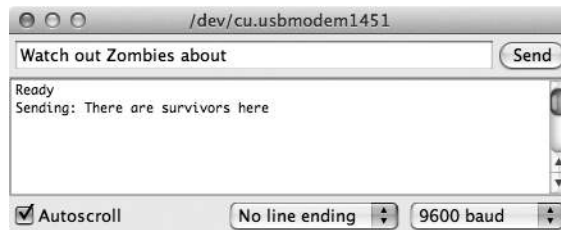


FIGURE 10-16: CHANGING THE MESSAGE USING THE SERIAL MONITOR

Here, the current message, “There are survivors here,” should change to “Watch out zombies about” when the Send button is pressed.

If you prefer to use your Raspberry Pi to change the message, install the terminal program `screen` (your Raspberry Pi will need an Internet connection):

```
$ sudo apt-get install screen
```

Once `screen` is installed, connect the USB lead between your Raspberry Pi and the Arduino and then enter the following command on your Raspberry Pi:

```
$ screen /dev/ttyACM0 9600
```

At this point, anything you type should be sent to the Arduino, and any messages coming from the Arduino should be displayed. Figure 10-17 shows the message being changed using `screen`. Note that the message will not appear on the screen as you type it but only after you press ENTER.

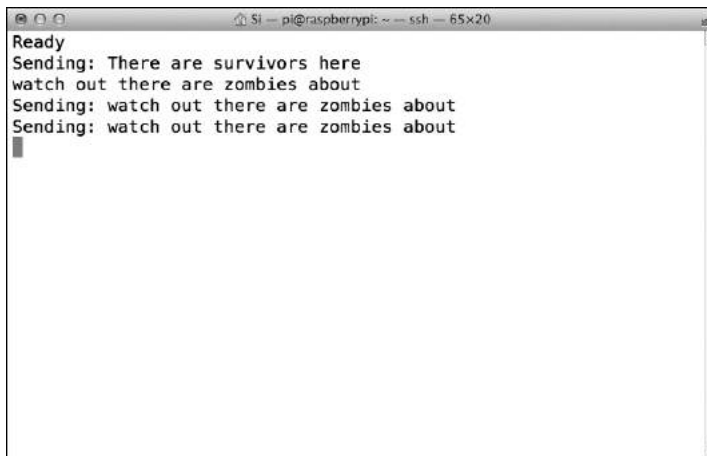


FIGURE 10-17: CHANGING THE MESSAGE USING THE `screen` COMMAND

Once the message has been changed, the Arduino will remember it, so you can unplug the Arduino to get ready for installation. Unplugging the Arduino will quit the `screen` command by closing the serial connection to the Raspberry Pi.

Now just attach your project to your desired location, preferably one with 360-degree visibility, and start blinking your message. Figure 10-18 shows the project fixed to my zombie-proof shed.

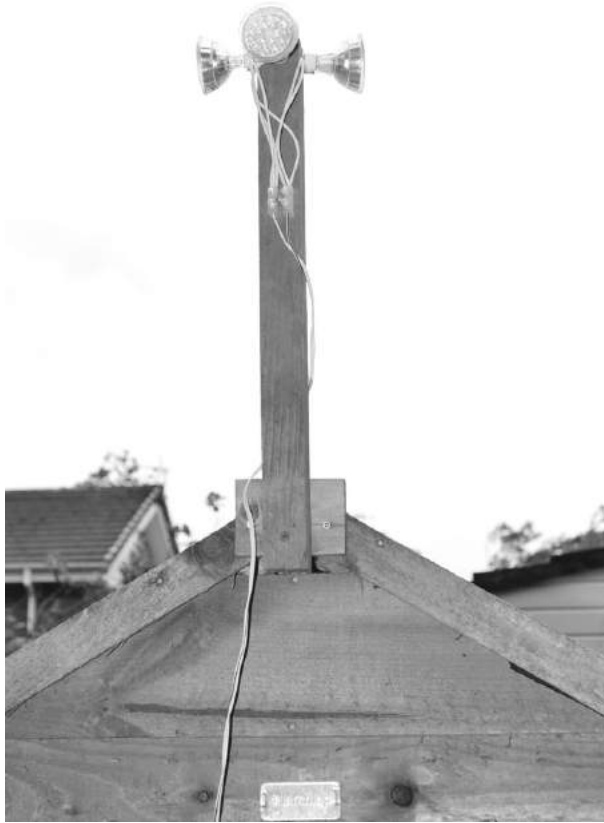


FIGURE 10-18: INSTALLING THE MORSE BEACON

If you want to conserve power, only use your beacon at night, when it is most likely to be spotted. But beware: popular culture gives us mixed messages on whether zombies are attracted to flashing lights. You may want to reinforce your stronghold before sending out messages, just in case.

In Chapter 11, we will continue with the theme of communication. For the final project of this book, we'll build a pair of haptic communication devices that will allow you and a fellow survivor to communicate silently, without alerting zombies to your presence.