Lesson 17 Control Tone with IR Remote

The Big Idea:

The ability to detect and decode Sony-protocol infrared messages, which was explored in Lesson 16, can be added to any Arduino™ sketch. This means that with the addition of an infrared sensor, an Arduino™ sketch can be made to respond to the buttons being pushed on a remote control.

This lesson combines this remote controllability with the techniques for generating musical notes explored in Lesson 10 to transform a television remote into a sort of musical instrument.

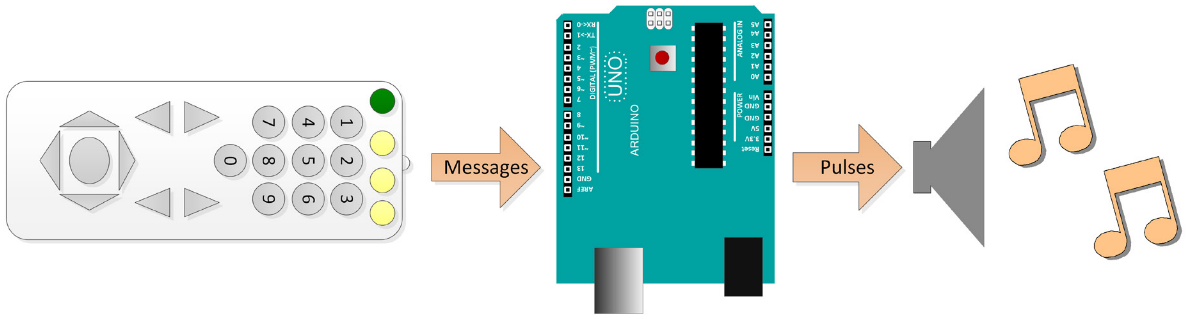


Figure 17-1. Create musical instrument using remote control, Arduino™ and speaker

Background:

Clearly a television remote control does not make a good musical instrument. But that's not really the purpose of this lesson. The next lesson, Lesson 18, adapts the infrared remote to the rolling robot built in Lesson 15. The purpose of this lesson is to introduce adding the ability to detect and decode infrared messages to a sketch intended to do something else, in this case play a musical note. The final sketch will play eight different notes in response to pressing the numbers 1 through 8 of the remote control.

This lesson also introduces the *array*, a programming tool for managing a group of values. In this case the array holds the values of the eight notes the remote can play. Arrays are extremely useful in programming and, therefore, very common.

Table 17-1. Vocabulary

Term	Definition
array	A set of values, in this case integers that represent different musical notes.
array declaration	The programming statement that creates and initializes an array.
array type	The type of data that a cell may contain. In this lesson the array is of type `int`, meaning each cell may store only an integer. But arrays can be of other types, including `boolean`, `double`, and `String`.
cell	An individual location within an array. A value is stored in a cell.
index	The address of a cell within an array. These are assigned in order, beginning with zero. The index of the first cell is always 0, the second is 1, the third is 2, and so on. A cell of length 5, for example, has the index numbers of 0, 1, 2, 3, and 4.
value	The contents of a cell.

Description:

The `Lesson17IRPlayTones.ino` *sketch*

The Arduino™ sketch in Lesson 16 is `Lesson16SonyIRProtocol.ino`. A reminder here: the approach and much of the code in this sketch is from a LadyAda tutorial, as is noted in the heading comments. Contained in this sketch is the method `printFull()`, which abstracts and prints the address and command components of the message.

The sketch for this lesson is `Lesson17IPlayTones.ino`. It is essentially a copy of the Lesson 16 sketch but with the following changes:

1. An array of eight integers is added to identify the eight different musical notes that are to be played in correspondence with the buttons 1 through 8 on the remote control. The actual values contained in each cell of this array are defined by the file `pitches.h`, which was introduced in Lesson 10. This file is also included in this sketch.
2. The `printFull()` method is renamed `playNote()` and its contents modified as follows:
    a. The command is evaluated to see if it is one of the buttons 1 through 8. If not, then any tone currently being played is stopped.

    b. If a button 1 through 8 is being pushed, then the note corresponding to the array positions of the button is played.

But this only works because of that array of integers.

Arrays

An array is a collection of like values. In that sense, it can be considered a list and be given a name. To be used, an array must first be created, then filled with the desired values, and finally accessed by the sketch.

Creating an array of integers

Suppose there is need for a list of numbers three long. The list needs a name. That name is a variable. As with any variable it has an *array type*, in this case an integer array, and a name. The act of creating it is called *array declaration*. The C statement in Example 17-1 declares a variable of type `integer array`. The resulting array has room for three integers.

Example 17-1. Declaring an array

```
int myNumbers[3];  // create int array of length 3
```

Setting the values of each cell in the array

Important

Each location in an array is called a *cell*. The array in Example 17-1 has three cells. Cells are numbered beginning with the number 0. The number of the last cell in an array is always the length of the array minus 1.

The programming statements in Example 16-2 set the value of the first cell to -27, the second cell to 45, and the third cell to 14.

Example 17-2. Setting the value of a cell

```
myNumbers[0] = -27;      // first cell is now -27
myNumbers[1] = 45;       // second cell is now 45
myNumbers[2] = 14;       // third cell is now 14
```

An array can be created and initialized in a single step, as shown in Example 17-3.

Example 17-3. Initializing an array

```
int myNumbers[] = { -27, 45, 14};  // same result
```

Accessing the values of each cell in the array

Suppose there is need for assigning the contents of the third cell to another variable. The contents of the third cell is the name of the array with the index in square brackets. Example 17-4 assigns the value 14 to the variable `aNumber`.

Example 17-4. Assigning the value to a variable

```
int aNumber;    // declare int variable
aNumber = myNumbers[2]; // the value 14 is assigned to aNumber
```

One advantage of the cells in an array each having an index number is that a `for` loop can be used to access the values in order. The following prints the values of the array to the serial port.

Example 17-5. Printing values of the array to the serial port

```
...
for(int i = 0; i < 3; i++){
     Serial.print("Index " + i);
     Serial.println(" has the value of: " +
          myNumbers[i]);
}
```

If this code snippet were run, the following would appear on the computer monitor:

```
Index 0 has the value of: -27
Index 1 has the value of: 45
Index 2 has the value of: 14
```

In this lesson, the array is named `myNotes`, and it is declared and initialized as shown in Example 17-6.

Example 17-6. Declaring and initializing an array

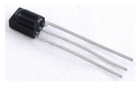
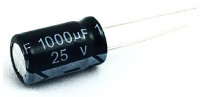
```
...
int myNotes[] = {NOTE_C5, NOTE_D5, NOTE_E5, NOTE_F5,
                 NOTE_G5, NOTE_A5, NOTE_B5, NOTE_C6};
```

The values of each of the cells, in turn, are integers. `NOTE_C5`, for example, is declared in `pitches.h` as the integer value 523. If a speaker were connected to pin 5 of an Arduino,™ the C-language statement shown in Example 17-7 would result in the note C5 being heard.

Example 17-7. Playing first note in an array

```
...
     tone( 5, myNotes[0]);  // plays C5 tone
```

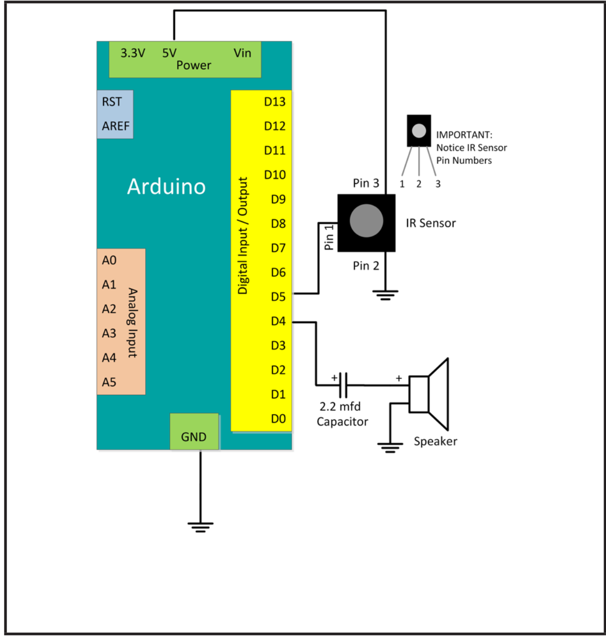
Materials:

Quan-tity	Part	Image	Notes	Catalog Number
1	Arduino™ Uno		Single-board computer. This board is delicate and should be handled with care. When you are not using it, keep it in a box or plastic bag.	3102
1	USB Cable		This is a standard USB adapter cable with a flat connector on one end and a square connector on the other.	2301
1	Computer with at least one USB port and access to the Arduino™ website, http://www.arduino.cc.	---	The operating system of this computer must be Windows, Macintosh OS/X, or Linux.	---
1	Infrared sensor		3-pin, 38kHz.	1302
1	2.2 microfarad capacitor		Actual value is not critical. Any capacitor of values between 1.0 and 10.0 microfarads will serve.	0205
1	Speaker		Magnetic speaker 4, 8, or 16 ohm.	3119

Procedure:

Important	The sketches in this lesson require a file named `pitches.h`. This file can be found on the Arduino™ website at http://arduino.cc/en/Tutorial/tone.

1. Using a bread-board, wire the parts as shown in Figure 17-2.



Caution

This is a schematic diagram, not a pictorial diagram. This means the physical locations of the various pins may be different from those shown here.

The IR sensor is an example. This diagram implies the center pin is pin 1. But it is not. The pins on the IR sensor are numbered 1, 2, 3, left to right when the round button is facing the observer.

Figure 17-2. Schematic diagram of Arduino™ Uno wired to IR sensor and speaker

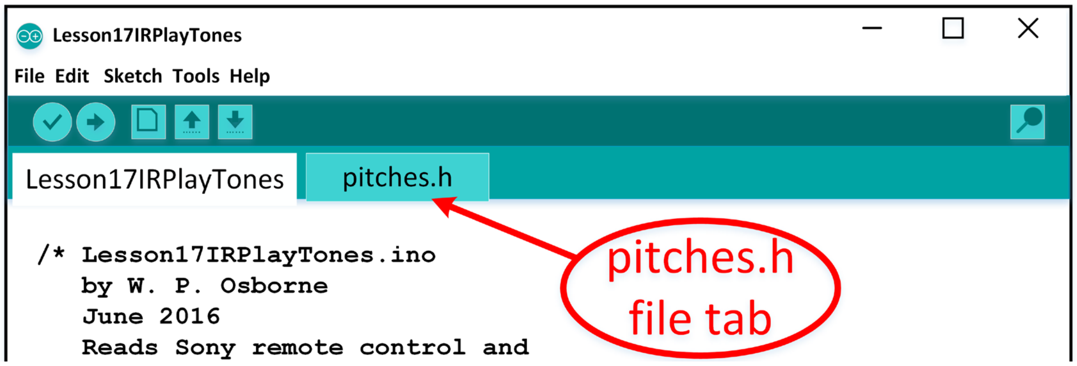
2. Make a copy of the `Lesson16IRSonyProtocol` sketch to use as a base for this lesson. Rename it `Lesson17IRPlayTones.ino`. The steps are:
   a. From within the Arduino™ IDE open the `Lesson16IRSonyProtocol` sketch you used for preparing the remote control command code spreadsheet.

   b. From the Arduino™ File menu select Save As. Then, save the sketch as `Lesson17IRSonyDecodeToTone`. This is the sketch you will modify, leaving the `Lesson16IRSonyProtocol.ino` sketch unchanged.

Before making any changes to `Lesson17IRSonyDecodeToTone` run it to make sure it works exactly the same as `Lesson16IRSonyProtocol`.

3. If necessary, add the file `pitches.h` to this sketch. The file may be found on the Arduino™ website at http://arduino.cc/en/Tutorial/tone. Add it to the sketch by:
   a. placing a copy of the file in the Lesson17IRSonyDecodeToTone folder.

   b. closing and then reopening the Arduino™ IDE to the `Lesson17IRSonyDecodeToTone.ino` sketch. The `pitches.h` file should appear in the IDE with its own tab as shown in Figure 17-3.



```
/* Lesson17IRPlayTones.ino
   by W. P. Osborne
   June 2016
   Reads Sony remote control and
```

Figure 17-3. Portion of Arduino™ IDE showing tab for `pitches.h` *file*

c. adding two lines to the very top of the `Lesson17IRSonyDecodeToTone` sketch, just after the introductory comments and before DEFINITIONS AND GLOBAL VARIABLES, as shown in Snippet 17-1.

Snippet 17-1. Modification of `Lesson17IRSonyDecodeToTone` *sketch to include* `pitches.h`*. (Added lines are highlighted.)*

```
...
// INCLUDE FILES
#include "pitches.h"

/////////////// DEFINITIONS AND GLOBAL VARIABLES
//////////////
// DEFINE the allowed input pins. These are the pins to which
// pin 1 of the IR Receiver may be connected
#define IRInputPin0 1
#define IRInputPin1 2
#define IRInputPin2 4
#define IRInputPin3 8
#define IRInputPin4 16
#define IRInputPin5 32
#define IRInputPin6 64
#define IRInputPin7 128
```

In the next few steps we will be testing the ability of this circuit and sketch to play and stop a tone by direction from a Sony-protocol remote control.

4. Locate the `printFull()` method in the `Lesson17IRSonyDecodeToTone` sketch. It will be near the bottom of the sketch and can be identified by the comments just above that say, "Prints the components of the detected message. This method, as written, sends the command

and address of a received message to the Serial Monitor. We are going to replace this method with a new one of the same name that plays a tone instead. We do this by:

a. Replacing the comments above the method so that they look like Snippet 17-2:

Snippet 17-2. Modifications to `printFull()` comments

```
//////////PRINTFULL//////////
// isolates the address and the command from the message, then
// plays a tone
```

b. Replacing the programming statements within the method so that they play a note when the number 5 is pressed on the remote and stops playing it when any other number is pressed.

The resulting `printFull()` method should look as shown in Snippet 17-3, Modifications to `printFull()` to play tone. Delete the code that does the printing and replace it with code that will play an *A* note when button number 5 is pushed and stop playing whenever any other button is pushed.

Snippet 17-3. Modifications to `printFull()` to play tone

```
//////////////// PRINTFULL //////////////
// isolates the address and the command from the message,
// then plays a tone
void printFull(uint16_t n)
{
  // break into command and address
  uint8_t command = n & 127;
  uint16_t address = (n >> 7);

  if(command == 4){
    tone(4, NOTE_A4);
  }
  else {
    noTone(4);
  }
}
```

5. Upload the program and test. A tone should start playing when button number 5 is pressed and stop when any other button is pressed.

Notice how simple these changes are. Now, one by one, the pieces necessary to make playing a simple melody possible will be added.

6. So far, these changes have "hard coded" the sound pin into the `printFull()` method. This is not good practice. A constant that identifies the pin should be created at the top of the program, and the name of that constant should replace the number 4 inside the `printFull()` method. To do this:

   a. At the top of the sketch and, within that, at the bottom of the declarations of constants, add the highlighted statements, as shown in Snippet 17-4.

Snippet 17-4. Adding the definition of `soundPin`
(Added statements are highlighted.)

```
// Variable to hold the collection of incoming low pulses
// that contain the address and message.
// For those who don't remember, uint means unsigned integer.
// 16 means sixteen bits. This variable is used to contain
// the 12 bits of the incoming message.
uint16_t currMessage = 0;

// TONE related variables and definitions
#define SOUND_PIN 4
```

   b. Inside the `printFull` method, replace the number 4 with the constant `soundPin`.

Snippet 17-5. Replacing a variable with the constant `soundPin`

```
//////////////// PRINTFULL //////////////
// isolates the address and the command from the message,
// then plays a tone
void printFull(uint16_t n)
{
  // break into command and address
  uint8_t command = n & 127;
  uint16_t address = (n >> 7);

  if(command == 4){
    tone(SOUND_PIN, NOTE_A4);
  }
  else {
    noTone(SOUND_PIN);
  }
}
```

Again, test the program to make sure it still works.

7. Recall that the intent is to play different notes by pressing different buttons on the remote. For this, an array of integers is added to the sketch. The array is eight cells long, and each cell contains the frequency for a musical note. These frequencies are defined in `pitches.h`.

   a. Create an array variable to hold the eight notes. Call it `myNotes`. The declaration should go just below the declaration of `soundPin`; see Snippet 17-6.

Snippet 17-6.

```
// TONE related variables and definitions
#define SOUND_PIN 4


//Array of notes
int myNotes[] = {NOTE_C5, NOTE_D5, NOTE_E5, NOTE_F5, NOTE_G5
                 NOTE_A5, NOTE_B5, NOTE_C6};
```

   b. The indexes of this array are 0, 1, 2, 3, 4, 5, 6, 7. As with any array, the index of the first cell is 0, and the last is the cell length minus one. By happy coincidence, the commands generated by the number buttons on a Sony-protocol remote also begin with 0 and are always one less than the number of that button, as shown in Table 17-2.

Table 17-2. Commands sent by remote number button

Button Number	1	2	3	4	5	6	7	8
Command Sent	0	1	2	3	4	5	6	7

Because the commands of the buttons exactly match the indexes of the array, to play different notes the only modification needed to `PrintFull()` is to replace `NOTE_A4` with the `myNotes` cell corresponding to the button command. This is shown in Snippet 17-7.

Snippet 17-7. Modification of `PrintFull()` to play note from `myNotes` array

```
//////////////// PRINTFULL //////////////
// isolates the address and the command from the message,
// then prints them to the serial port.
void printFull(uint16_t n) //prints first 12 bits (starting
from MSB)
{
  // break into command and address
  uint8_t command = n & 127;
  uint16_t address = (n >> 7);
```

```
    if(command >= 0 && command < 8){
   view ban, myNotes[command]);
  }
  else {
    noTone(SOUND_PIN);
  }
}
///////////////// END PRINTFULL ////////////////////////
```

The Arduino™ should now play a different note for each of the first eight buttons of the Sony-protocol remote control.

Exercises:

Exercise 17-1. Play a simple melody

Show working numbers by playing a simple melody.

Exercise 17-2. Shift scale, play sharp or flat

Put some of the other remote control buttons to work for shifting scale or playing sharps or flats, or some other music-related purposes.

Complete listing 17.1. `Lesson17IRSonyDecodeToTone.ino`

This sketch may be downloaded from LearnCSE.com.

```
// Lesson17IRSonyDecodeToTone.ino
// by W. P. Osborne
// 6/30/15
// Reads Sony remote control and
// displays the encoded 12 bits. The first seven bits are the
// "command", the remaining 5 bits are the device being
// addressed.
//
// Based on code from LadyAda tutorial as edited,cut,
// enhanced, and generally changed by Luke Duncan,
// Preetum Nakkarin, and W. P. Osborne.

// PIND is an Arduino™ command that gets a byte containing
// the input status of bytes 0 through 7. In our case we want
// the bit corresponding to pin 2.
// NOTE: If you are seeking input from ports 8 - 15 use
// PINDB. For analong pins 0 - 5 use PINC
//
// Modified from IRSonyDecodeToDecimal to match the flow
// charts used for the lesson. By W. P. Osborne.


// INCLUDE FILES
#include "pitches.h"

/////////////// DEFINITIONS AND GLOBAL VARIABLES
//////////////
// DEFINE the allowed input pins. These are the pins to which
// pin 1 of the IR Receiver may be connected
#define IRInputPin0 1
#define IRInputPin1 2
#define IRInputPin2 4
#define IRInputPin3 8
#define IRInputPin4 16
#define IRInputPin5 32
#define IRInputPin6 64
#define IRInputPin7 128
```

```
// Assign the pin to be used for the IR sensor
#define IRpin_PIN  PIND
#define IRpin      IRInputPin5

// Sony protocol has 12 data bits in its message.
#define NUMBITS 12

// RESOLUTION is how finely we can measure the length of a
// pulse. Any measurement is a multiple of RES
#define RES 50  // resolution

// A low pulse of length LOWUS is a binary zero
// A low pulse of length HIGHUS is a binary one
#define LOWUS 600
#define HIGHUS 1200

// Maximum amount of time we'll wait for a pulse.
#define MAXPULSE 5000 // longest pulse we'll accept

// Maximum amount of time we'll wait to see if there
// are any pulses coming in. Used to find the end
// of the rest period between messages.
#define MAXREST 20000 // longest we'll wait for a pulse

// Define variables to be used to tell the wait() method
// if it is timing a high pulse or a low pulse.
#define HIGHP true
#define LOWP false

// Variable to hold the collection of incoming low pulses
// that contain the address and message.
// For those who don't remember, uint means unsigned integer.
// 16 means sixteen bits. This variable is used to contain
// the 12 bits of the incoming message.
uint16_t currMessage = 0;

// TONE related variables and definitions
#define soundPin 4

// Array of notes
int myNotes[] = {NOTE_C5, NOTE_D5, NOTE_E5, NOTE_F5, NOTE_G5,
                 NOTE_A5, NOTE_B5, NOTE_C6};
```

```
/////////////// END OF DEFINITIONS AND GLOBAL VARIABLES
///////////

/////////////// SETUP (Light yellow on flow chart) ///////////
// All we set up is the serial port so we can see the
// commands and addresses
void setup()
{
  Serial.begin(9600);
  Serial.println("Ready to decode IR!");
}
/////////////// END OF SETUP METHOD //////////////////


///////////////// LOOP (Dark yellow on flow diagram) /////////
// Remember, loop is run over and over and over and over ////
and....
void loop()
{
  if(readIR())
  {
    printFull(currMessage);
  }
}
////////////////// END OF LOOP METHOD /////////////////////


////////////////// READIR (Blue on flow chart) //////////////
// Attempts to read a message. First, it looks for a rest
// period.
// Remember, the IR Sensor output is HIGH when there is no
// IR detected.
// If it goes LOW sooner than the length of a rest period
then
// then we have a message. A Sony remote sends messages in
// groups of three. The rest period is the time between two
// messages.
//
// If a candidate rest period is found then IRRead looks to
// see if the input then drops long enough to be a start
// pulse, but not too long.
//
```

```
// Finally, if the rest period and start pulse are detected,
// the 12 data bits are collected and put into an unsigned
// integer for later decoding.
//
// Notice how readIR() makes extensive use of the wait()
// method.
boolean readIR()
{
  currMessage = 0;

  if( wait(HIGHP, MAXREST) == -1)
  {//timeout ( before sees a low pulse)
    return false;
  }

  if(wait(LOWP, MAXPULSE) == -1) //start pulse
  {
    return false; //timeout
  }
  //start decoding message

  for(int i = 0;i<NUMBITS;i++)
  {
    if(wait(HIGHP, MAXPULSE) == -1)
     {return false;}
    int data = wait(LOWP, MAXPULSE);
    if(data == -1)
    { return false; } //timeout

    // Here the pulse is put in its
    // appropriate bit position.
    if(data * 2 > LOWUS + HIGHUS)
    {
      currMessage = currMessage | (1 << i);
    }
    else
    {

      //It's a zero, do nothing
    }
  }
  return true;
}
///////////// END IRREAD Method //////////////////////
////////////// WAIT (Green on flow diagram)
////////////////////////
// In the following the (IRpin_PIN & (1 << IRpin)) test is a
// bit tricky. We named setthe value of IRpin to 2 because
// that is the pin to which the output of the sensor is
// attached.
// But, that pin in IRpin_PIN is actually in the third
// position.
// So, to do the test IRpin value must be shifted to the left
// one bit. The (1 << IRpin) results in a "mask", that limits
// testing to the desired pin(s). In this case, it is Pin 2.
//
// A visual example may help:
//     Assume the only HIGH is on Pin 2, to which the output
//     of the IR sensor is connected.
//     The bit pattern of IRpin_PIN is:  00000100
//
//     But, the bit pattern of IRpin is: 00000010 because
//     this is decimal 2.
//     To make the "AND" test meaningful the IRpin pattern
//     must be shifted left 1 bit.
//     This is done with the (1 << IRpin) command.
//     Then, the pattern is:
//                                        00000100
//
// This trick works with pins 1, 2, and 4 but none of the
// others.
// Pin 3, for example, resolves to the decimal value of 8.
// If the input were connected to Pin 3 the test would be:
//     (IRpin_PIN & 8)








// A better way to do this may be as follows:
//      Pin to be tested  Mask in binary  Mask in DEC
//      0                  00000001        1
//      1                  00000010        2
//      2                   00000100       4
//      3                   00001000       8
//      4                   00010000       16
//      5                   00100000       32
//      6                   01000000       64
//      7                   10000000      128

//wait until the pulse of type TYPE ends. returns -1
// if timedout, otherwise returns length of pulse
int wait(boolean type, int timeout)
{
  int num = 0;  // we wait in increments of RES in
microseconds.
  while( ( type == HIGHP && (IRpin_PIN & IRpin) )
          ||
         ( type == LOWP && !(IRpin_PIN & IRpin ) ) )
  {
    // pin is still HIGH
    delayMicroseconds(RES);
    num++;
    // if the pulse is too long we 'timed out'
    if(num*RES >= timeout)
    {
      return -1;
    }
  }
  return num*RES;  // return the duration of the pulse
}
//////////////// END WAIT METHOD /////////////////////////


////////// PRINTFULL /////////////////////////
// Plays the tone selected by the button pushed
// on the Sony-compatible remote control
void printFull(uint16_t n) //prints first 12 bits (starting
from MSB)
{

  // break into command and address
  uint8_t command = n & 127;
  uint16_t address = (n >> 7);
  if(command >= 0 && command < 8){
    noTone(soundPin);
    delay(60);
    tone(soundPin, myNotes[command]);
  }
  else
    noTone(soundPin);
}
///////////////// END PRINTFULL ////////////////////////
```