**SIRC using an ATmega328**

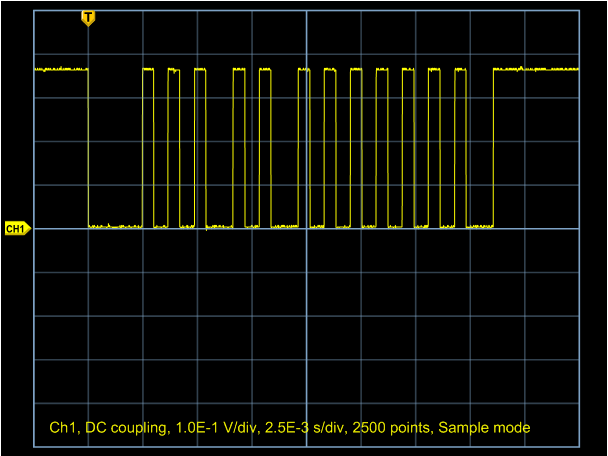
Page:1 [2](http://avrprogrammers.com/proj_sirc_2.php) [3](http://avrprogrammers.com/proj_sirc_3.php) [4](http://avrprogrammers.com/proj_sirc_4.php) [5](http://avrprogrammers.com/proj_sirc_5.php)

**Introduction**

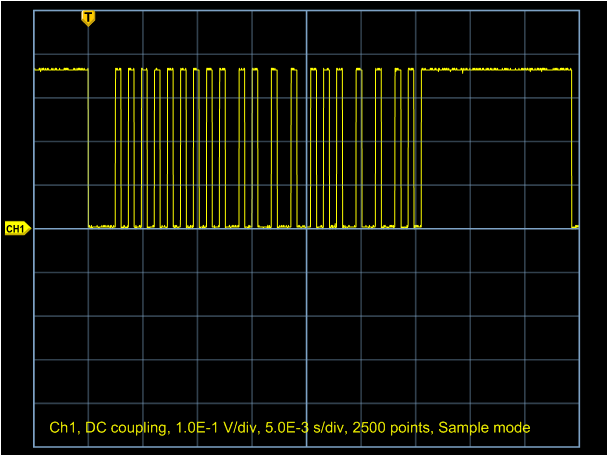
The purpose of this project is to demonstrate the use of an ATmega88, -168, or -328 as an SIRC IR remote control receiver. IR transmitters send a serial signal modulated onto a carrier that is around 40 kHz. The receiver module sees the 40 kHz signal and demodulates the serial signal from it. This prevents any random IR from interfering with the receiver. In this respect all IR remotes are the same. Beyond that, the frequency can vary from 30 to 56 kHz, and the serial signal can be pulse width modulated, pulse position modulated, or who knows what.

There is also an [ATtiny2313 SIRC version...](http://avrprogrammers.com/proj_sirc_attiny2313_1.php)

The screen captures below show the Sony 12-bit and 20-bit SIRC protocol as output by the TFMS-5400, at the input to the ATmega328. I don't have a 15-bit sample because I couldn't find any remote that offered it.



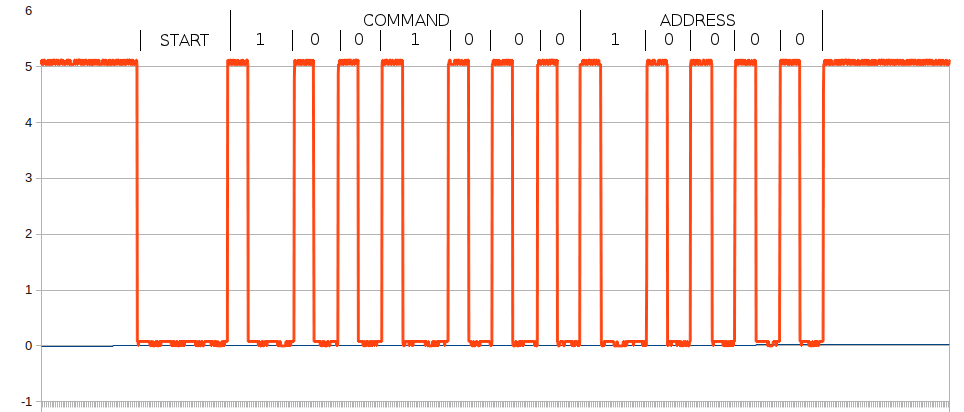
A frame from an SIRC-12 button on a Cox cable remote.



A frame from an SIRC-20 button on a Cox cable remote.

I chose the Sony SIRC protocol because it looked fairly simple. I use the TFMS-5400 40kHz device, which matches the Sony SIRC specification. More importantly, I have a bag full of them that I picked up on eBay some time back. There are other parts and other families which may be more appropriate for other protocols. [[TFMS-5xx0 datasheet](http://avrprogrammers.com/files/779-3-157692-FMS5330.pdf)] [[TSOP-17xx datasheet](http://avrprogrammers.com/files/TSOP1740.pdf)] The output of the TFMS-5400 is a logic level serial signal that matches whatever format the transmitter sends, which in this case is pulse width modulated. The device is 5V only, so the signal is 0-5V, but I have run it at 3.6V with good results, as can be seen by the captures above.

A frame always starts with a 2.4 mS logic zero. At the end of the 2.4 mS pulse, the first bit starts. A bit is a logic one for 0.6 mS, followed by a logic zero for either 0.6 mS for a zero bit, or 1.2 mS for a one bit. The frame consists of the start bit, 7 command bits, and 5 address bits for a 12-bit frame. The 20-bit frame consists of the 12-bit frame followed by 8 "extra" bits.



A frame from an SIRC-12 button on a GE remote.

I annotated the image above to show what you get from hitting the "0" key. The command is 9, and the address is 1. The data is sent LSB first. The largest command number possible is 127 and the largest address possible is 31. To decode the serial bit stream, we have to come up with a way of differentiating a one from a zero. The high part of the bits are all the same, so they don't help us identify the bit, but the low part is 0.6 mS for a zero and 1.2 mS, twice as large, for a one. That suggests timing the low part of the bit, and that's what we'll do. With timer 2 set to use the prescaler with the divide by 256 tap we get 16 uS per count. The three times we are interested in are 2400 uS (a count of 150), 1200 uS (a count of 75) and 600 uS (a count of 38). To simplify the interrupt routine a little, we assume that the bit is a zero. The timer is started at the falling edge of the signal, and the count is read at the rising edge. If the count is around 150, we know it is a start bit. If the count is around 75, we know it is a one bit.

To assemble the command and address fields, we rotate the bit value into the MSB of the command field, until we have counted the correct number of bits (7), then we switch to rotating bits into the address field until we hit the correct total number of bits (12). Anything else that comes is rotated into the "extra" field, which should be 8 bits, but we don't really care, because we don't use them. In a 15-bit frame, three bits will come in after the 12-bit frame and will fall in the extra field. At the end, if there have been 15 bits received, the extra bits are added as the most significant bits in the address field. That should satisfy the 15-bit protocol, but I haven't been able to test it. Counting the bits is only to determine which field they belong to - it is not used to determine when we are done. Timer 2, if we do not see another low transition, will timeout after 4.08 mS. This is the signal that the frame has ended.

Page:1 [2](http://avrprogrammers.com/proj_sirc_2.php) [3](http://avrprogrammers.com/proj_sirc_3.php) [4](http://avrprogrammers.com/proj_sirc_4.php) [5](http://avrprogrammers.com/proj_sirc_5.php)

**SIRC using an ATmega328**

Page: [1](http://avrprogrammers.com/proj_sirc_1.php) 2 [3](http://avrprogrammers.com/proj_sirc_3.php) [4](http://avrprogrammers.com/proj_sirc_4.php) [5](http://avrprogrammers.com/proj_sirc_5.php)

**Key Mapping**

The first step in in deciding which keys can be used is determining what, if anything, each key sends. Although this information appears early in the project, it is about the last thing you find out. The decoder has to be functioning before you can tell what is being sent. To get all of the codes, I put a Serial.print() in the timerISR() routine to print the address and command every time a button was pressed. It gets messy, because there are misreads - more on some keys than on others. I think the cheap remote may have contributed something to that problem. Some keys read correctly 100% of the time, while others misfire every third time. I didn't have that problem with the Cox cable remote, but it would polute the signal by echoing in a different protocol. Actually, the decision to change was ultimately made because the volume up/down signals would go three or four times, then the protocol would change to the cable box protocol for three or four echoes, then back. I couldn't get a smooth fade. Eventually, it would signal an error on the Aux, and switch from Aux to TV. Hopefully a bug in the controller.

There must be a list of Sony remote codes somewhere, but I haven't been able to find one. With this little project, you will be able to make one easily enough. You just have to do the above trick for every Sony code number on a couple of universal remotes to get most of them. Luckily I needed to do that only twice for this SIRC project.

|  |  |  |
| --- | --- | --- |
| One Sony TV code | | |
| Addr | Cmd | Function |
| 1 | 37 | Input |
| 1 | 96 | Menu |
| 1 | 18 | Vol+ |
| 1 | 19 | Vol- |
| 1 | 20 | Mute |
| 1 | 58 | Info |
| 1 | 59 | Last |
| 1 | 116 | Up |
| 1 | 117 | Down |
| 1 | 51 | Right |
| 1 | 52 | Left |
| 1 | 0 | 1 |
| 1 | 1 | 2 |
| 1 | 2 | 3 |
| 1 | 3 | 4 |
| 1 | 4 | 5 |
| 1 | 5 | 6 |
| 1 | 6 | 7 |
| 1 | 7 | 8 |
| 1 | 8 | 9 |
| 1 | 9 | 0 |
| 1 | 17 | Ch - |
| 1 | 16 | Ch + |
| 1 | 101 | Ok |
| 1 | 14 | Guide |
| 1 | 20 | Mute |

An interesting thing about the address. Don't count on it always being the same. Any given button always sends the same address, but different buttons may send different addresses. There appear to be 5 different addresses in use on the Cox remote, and they seem to be grouped more-or-less by function. Other multi-function remotes seemed to do that also, but I didn't find a manufacturer's remote that behaved that way. To simplify things, though, I spent $10 on a GE universal remote. I set it up as a Sony TV, and all of the keys send the same address.

**The Design**

We should decide exactly what we want to do with the codes received. In the real world the codes would be mapped to functionality - some would pulse a line, some would set or reset a line, or some would change an analog voltage, or the code sent to an I2C potentiometer. Because the ATmega328 can pulse width modulate the voltage on an LED, I have chosen to simulate the volume control up/down function in that way. Other buttons, specifically 1 and 2, will turn LED's on while 4 and 5 will turn them off.

Not all remote control transmitters work the same. Some will resend the code up to three times in a short period of time for a single button push. Some will send the code once. All will repeat the code if the button is held down. The Sony protocol expects the message to be repeated every 45 mS. To prevent that from affecting the system, a command should be allowed to lock out further input until it has completed, if it so chooses.

The remote control runs entirely in the background. Commands will be presented to the main loop ("userville") and a flag indicates the arrival. The main loop will look for this flag and when set, will read the command. The address, and extra data are available as well, but the address will always be the address we set it to look for. It may be changed programmatically.

The code will be presented in the form of an Arduino library. An Arduino is not required, but the environment is, along with an ATmega88,-168, or -328. The code doesn't care which. You also don't need the Arduino bootloader in your chip, if you have an ISP, but I'll [leave that to you to figure out](http://www.arduino.cc/en/Hacking/Programmer).

Page: [1](http://avrprogrammers.com/proj_sirc_1.php) 2 [3](http://avrprogrammers.com/proj_sirc_3.php) [4](http://avrprogrammers.com/proj_sirc_4.php) [5](http://avrprogrammers.com/proj_sirc_5.php)

**Test Sketch**

This sketch demonstrates the library. Button 1 turns on an LED and button 4 turns it off. Button 2 turns on another LED and button 5 turns it off. The volume up and down fades an LED on and off.

#include

const int LED1 = 9;

const int LED2 = 12;

const int LED3 = 13;

const int cmdBtn1 = 0;

const int cmdBtn2 = 1;

const int cmdBtn4 = 3;

const int cmdBtn5 = 4;

const int cmdVolUp = 18;

const int cmdVolDn = 19;

const int adrBtn = 1;

int level1 = 1;

void setup() {

pinMode(LED1, OUTPUT);

pinMode(LED2, OUTPUT);

pinMode(LED3, OUTPUT);

}

void loop() {

if ( Sirc.commandReady ) {

Sirc.commandBusy = true;

if ( Sirc.address == adrBtn ) {

if ( Sirc.command == cmdBtn1 ) {

digitalWrite( LED2, HIGH );

} else if ( Sirc.command == cmdBtn4 ) {

digitalWrite( LED2, LOW );

} else if ( Sirc.command == cmdBtn2 ) {

digitalWrite( LED3, HIGH );

} else if ( Sirc.command == cmdBtn5 ) {

digitalWrite( LED3, LOW );

} else if ( Sirc.command == cmdVolUp ) {

if (( level1 += 5 ) > 254 ) {

level1 = 254;

}

} else if ( Sirc.command == cmdVolDn ) {

if (( level1 -= 5 ) < 1 ) {

level1 = 1;

}

}

analogWrite( LED1, level1 );

}

Sirc.command = 0;

Sirc.commandReady = false;

Sirc.commandBusy = false;

}

}

Page: [1](http://avrprogrammers.com/proj_sirc_1.php) [2](http://avrprogrammers.com/proj_sirc_2.php) 3 [4](http://avrprogrammers.com/proj_sirc_4.php) [5](http://avrprogrammers.com/proj_sirc_5.php)

**Library**

This is the code that receives and decodes the IR from the universal remote.

To read the pulses, we set an I/O pin up as an input with the ability to interrupt on an edge. A timer is set on the first, and all subsequent falling edges. The pulse width is measured by the count of the timer when the line goes back high. When we start the timer, we also flip the interrupt to trigger on the rising edge. We let the timer continue to run, though. If the timer, which is set to expire in around 4 mS, ever does expire, it will signal the end of the received frame. During this time, we receive bits, rotate them into the appropriate bucket, depending on which bit we are on, and we count the bits so we know where we are in the frame. Remember, the timer will fire when the end of frame is reached, so we don't need to track the total bits received for that purpose. We just receive bits until the timer fires.

#ifndef \_\_SIRC\_H\_\_

#define \_\_SIRC\_H\_\_

#ifndef cbi

#define cbi(sfr, bit) (\_SFR\_BYTE(sfr) &= ~\_BV(bit))

#endif

#ifndef sbi

#define sbi(sfr, bit) (\_SFR\_BYTE(sfr) |= \_BV(bit))

#endif

class sirc

{

public:

sirc( );

unsigned char address;

unsigned char command;

bool commandReady;

bool commandBusy;

bool commandLength;

unsigned char programmedAddress;

unsigned char error;

unsigned char extra;

void risingEdgeISR();

void fallingEdgeISR();

void timerISR();

private:

bool \_inbit;

unsigned char \_bit\_time;

unsigned char \_bit\_count;

unsigned char \_command;

unsigned char \_last\_command;

unsigned char \_first\_command;

unsigned char \_address;

unsigned char \_bit\_value;

unsigned char \_extra;

unsigned char \_pin;

};

extern sirc Sirc;

#endif

#include

#include

#include

#include

sirc Sirc;

ISR( TIMER2\_OVF\_vect )

{

Sirc.timerISR( );

}

void rising\_edge\_routine( )

{

Sirc.risingEdgeISR( );

}

void falling\_edge\_routine( )

{

Sirc.fallingEdgeISR( );

}

sirc::sirc( void ) {

\_pin = 2;

programmedAddress = 1;

command = 0;

address = 0;

\_bit\_count = 0;

extra = 0;

\_inbit = false;

TCCR2A = 0;

sbi( TCCR2B, CS22 );

sbi( TCCR2B, CS21 );

cbi( TCCR2B, CS20 );

attachInterrupt( 0, falling\_edge\_routine, FALLING );

sei( );

pinMode( 2, INPUT );

digitalWrite( 2, HIGH );

}

void sirc::fallingEdgeISR( )

{

TCNT2 = 1;

attachInterrupt( 0, rising\_edge\_routine, RISING );

sbi( TIMSK2, TOIE2 );

\_inbit = true;

}

void sirc::risingEdgeISR( )

{

\_bit\_time = TCNT2;

attachInterrupt( 0, falling\_edge\_routine, FALLING );

\_inbit = false;

\_bit\_value = 0;

if (( \_bit\_time > 70 ) && ( \_bit\_time < 82 )) {

\_bit\_value = 128;

}

if (( \_bit\_time > 150 ) && ( \_bit\_time < 160 )) {

\_bit\_count = 0;

return;

}

if ( \_bit\_count < 7 )

\_command = (\_command >> 1) + \_bit\_value;

if ( \_bit\_count < 12 )

\_address = (\_address >> 1) + \_bit\_value;

if ( \_bit\_count < 20 )

\_extra = (\_extra >> 1) + \_bit\_value;

++\_bit\_count;

}

void sirc::timerISR( )

{

cbi( TIMSK2, TOIE2 );

\_command >>= 1;

\_address >>= 3;

if ( \_bit\_count == 15 )

{

\_address &= \_extra;

}

if ( \_address == programmedAddress )

{

if (( \_first\_command ^ \_last\_command ^ \_command ) == \_command )

{

if ( !commandBusy )

{

address = \_address;

command = \_command;

commandReady = true;

commandLength = \_bit\_count;

}

}

\_last\_command = \_first\_command;

\_first\_command = \_command;

}

}

The timer ISR keeps track of the two previous commands, and only passes the command along if it matches those. Three consecutive matches is a good indication that you are getting the correct code. The commands are sometimes only partially received, causing the command to flip from one to another rapidly. This triple check makes sure that doesn't happen.

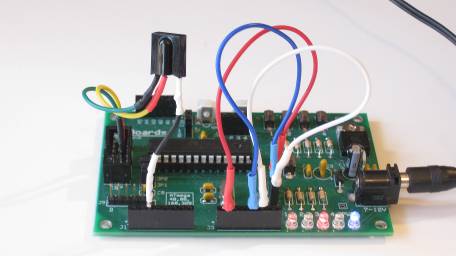
If the address matches, and the command passes the triple check, and the user has not put commands on hold, the new command is presented to the user and a flag is set telling him. It is a very good idea not to have one button toggle the state of something important, because the number of messages sent varies depending on how long the button is pushed. You may get one, or you may get 15. You must design your application to handle that possibility. For instance, on a mute functino, you should perform the action, then lock out commands for a second or so to prevent the function from being repeated (reversed) a few mS after being done correctly.

Page: [1](http://avrprogrammers.com/proj_sirc_1.php) [2](http://avrprogrammers.com/proj_sirc_2.php) [3](http://avrprogrammers.com/proj_sirc_3.php) 4 [5](http://avrprogrammers.com/proj_sirc_5.php)

**Downloads**

This is the [Arduino library](http://avrprogrammers.com/files/SircLib.zip) that receives and decodes the IR from the universal remote.

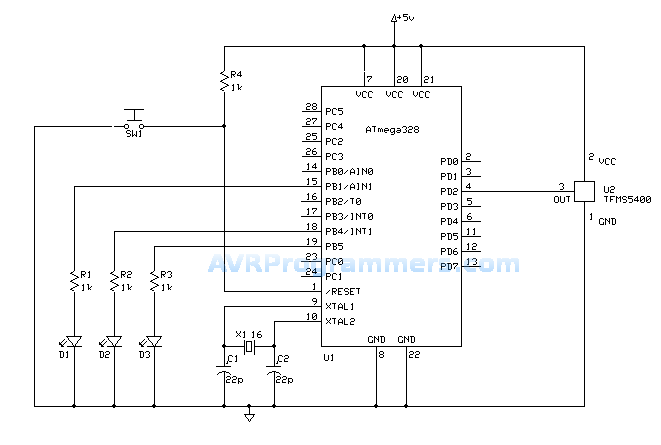
This is the [Arduino sketch](http://avrprogrammers.com/files/SircSketch.zip) that uses the library.



SIRC receiver on the AVR development board.

**Hardware**

For this design I used the Dev28a AVR development board with an ATmega328P, and then an ATmega168P. The only additional hardware was the TFMS-5400 IR receiver chip. Because I use the timer directly, the library won't work with other MCUs, like the ATmega32 or the ATmega644. It could be written to work with any of the MCUs, but that was beyond the scope of this project, and the depth of my forethought.



SIRC receiver schematic diagram.