Interfacing with other chips

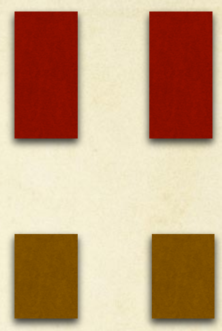
Examples of three LED driver chips

Why Add Other Chips?

- Lots of cool chips out there that add functionality beyond a basic Arduino
- One example – LEDs
  - From an Arduino you can drive 14 LEDs directly from the digital outs – what if you want more?
  - Use external LED-driver chip
  - Send data on which LEDs to turn on and of to that chip
  - Let it keep track of the LEDs while you do other things

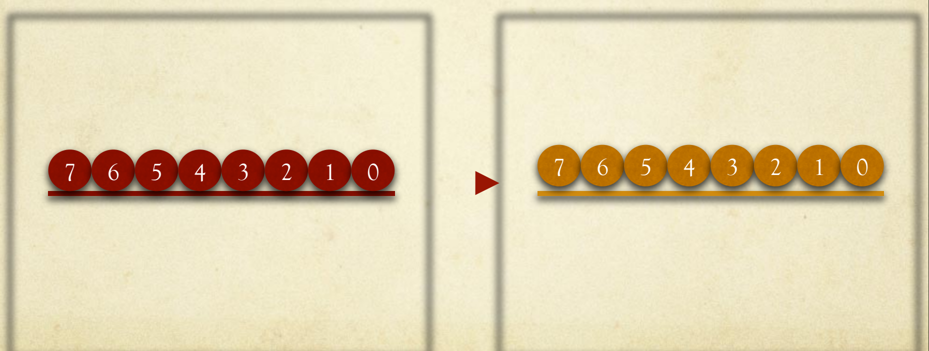
Communication Styles

- Parallel = multiple wires in parallel
- Serial = send data one at a time on one wire
  - In practice you usually need two wires: one for the data, and one to say when to look at the data (usually called Clock)
- So, serial communication takes more time, but uses fewer wires



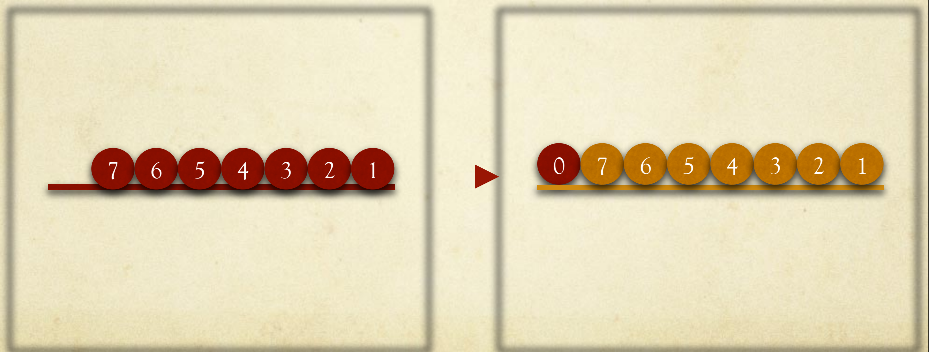
Shifting

- Shifting is the process of sending out a set of bits one at a time



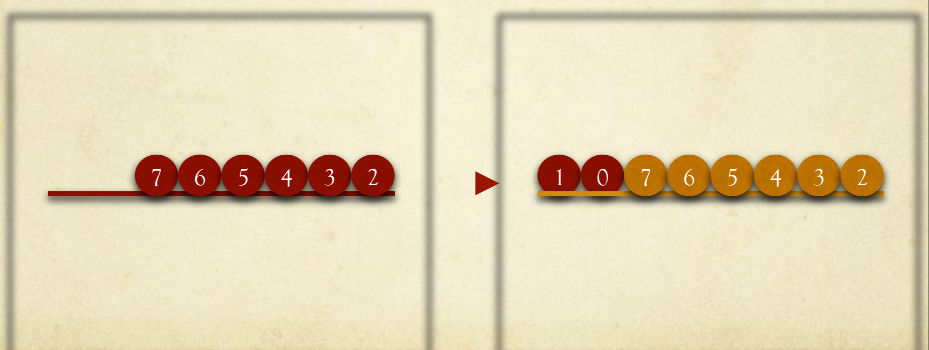
Shifting

- Shifting is the process of sending out a set of bits one at a time



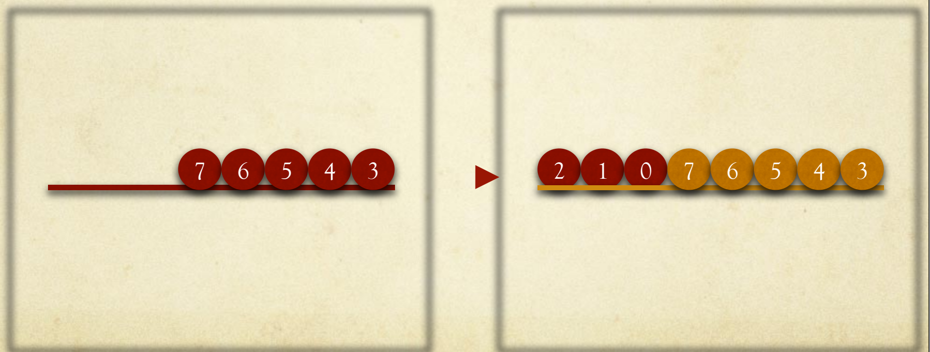
Shifting

- Shifting is the process of sending out a set of bits one at a time



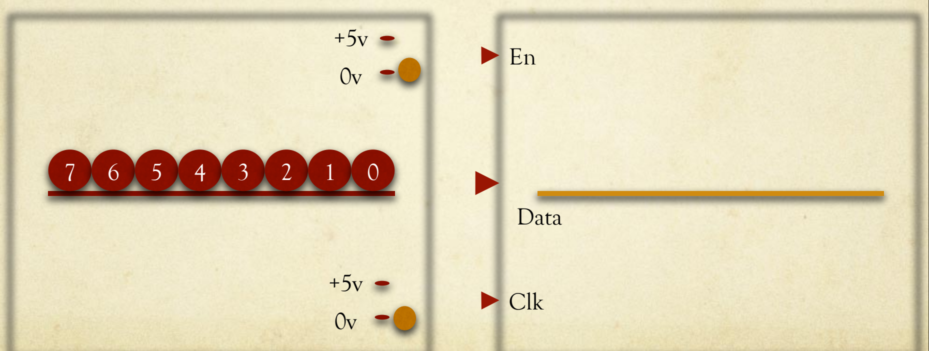
Shifting

- Shifting is the process of sending out a set of bits one at a time



Shifting

- There are a couple other control signals too…

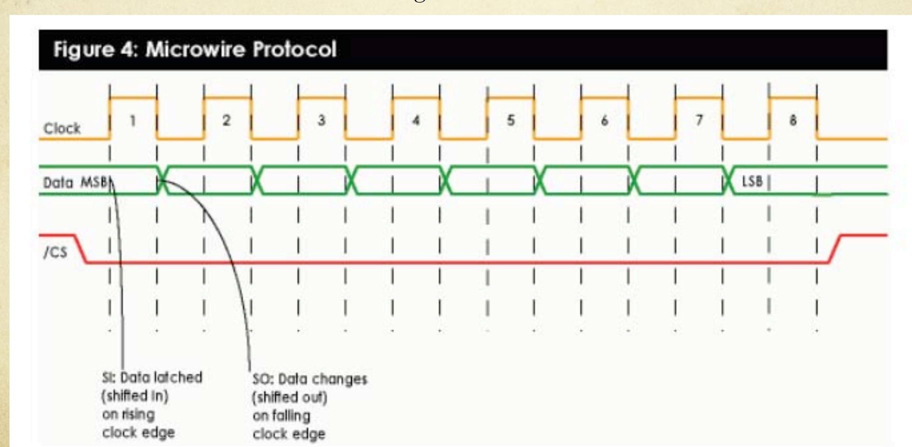


Overview

- There are a number of different protocols used for inter-chip communication (Arduino to external chip…)
  - Serial output – simplest protocol.
    - Also called SPI – Serial Peripheral Interface
    - CLK/Data/En, unidirectional
    - Example: STP08DP05 8-bit LED driver
  - SPI with more complex operation
    - Send data with SPI, both commands and data
    - Example: MAX 7219 8-digit LED display driver
  - I^2C/TWI – two-wire interface – more complex
    - CLK/Data - bidirectional
    - Example: Wii Nunchuck
  - Custom protocols – potentially complex
    - Example: TLC5940 16-bit PWM LED driver

Serial Output

- Two pins: Clk and Data
  - New data presented at Data pin on every clock
  - Looks like a shift register

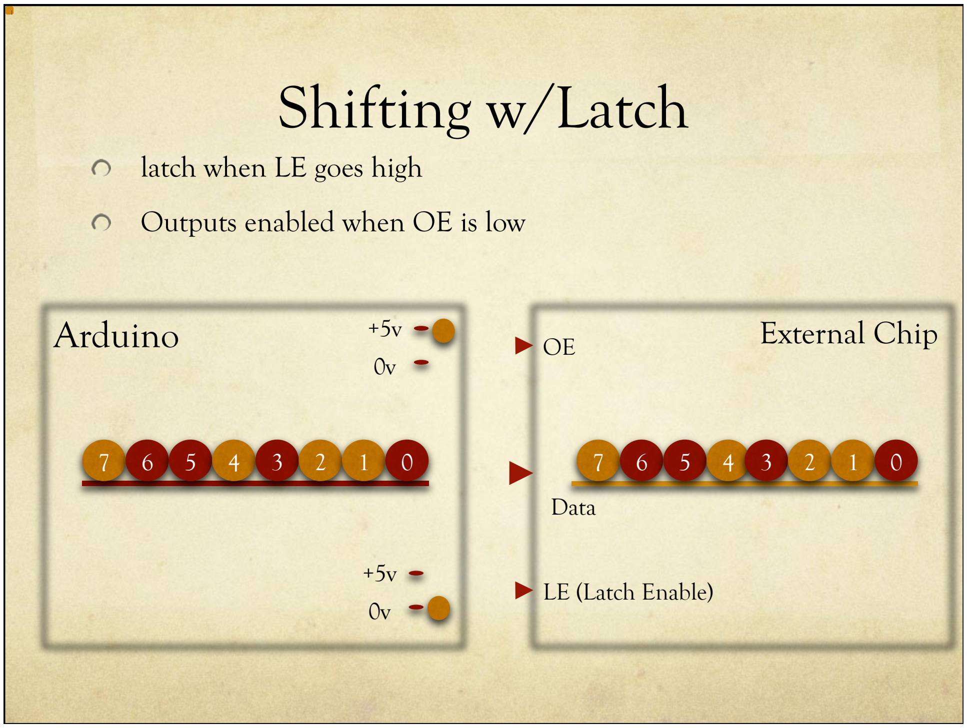


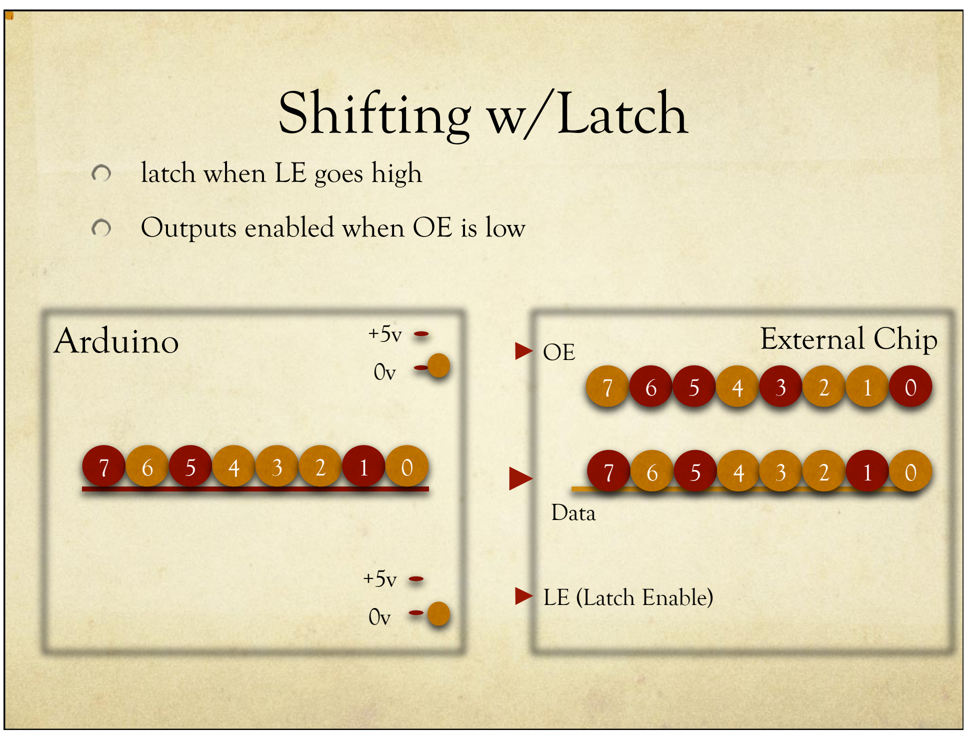
Example: Shift Register

- Simply connect LEDs to the outputs of the shift register
- The only problem is that the LED pattern changes while you're shifting it in...

Shifter with Output Latch

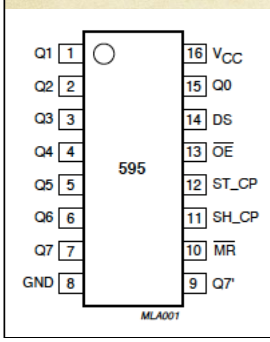
- One solution is to save the current outputs while you're shifting in the new ones
  - This is an "output latch"
  - Shift in new stuff "underneath" the bits that are being displayed
  - Then, all at once, swap the new bits for the old bits

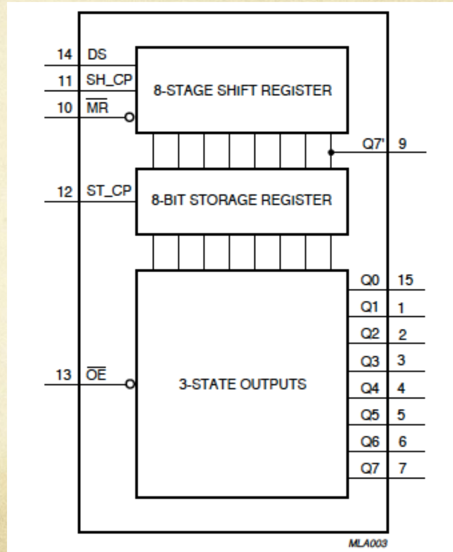

Shifting w/Latch
latch when LE goes high
Outputs enabled when OE is low
Arduino
+5v
0v
7 6 5 4 3 2 1 0
+5v
0v
OE
External Chip
7 6 5 4 3 2 1 0
Data
LE (Latch Enable)


Shifting w/Latch
latch when LE goes high
Outputs enabled when OE is low
Arduino
+5v
0v
7 6 5 4 3 2 1 0
+5v
0v
OE
External Chip
7 6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
Data
LE (Latch Enable)

Example: 74HC595

- This is a shift register with an output latch
  - You can save the previous values while shifting in new ones
- **BUT – need separate current-limiting resistor for each LED!**

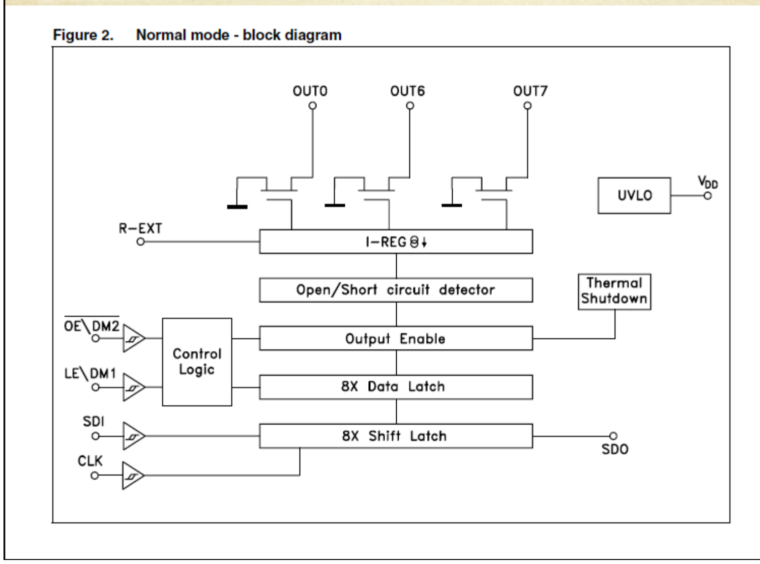




Example: STP08DP05

- Just like the 74HC595 – a shift register with a separate output latch
- ALSO – constant-current outputs for the LEDs
  - That means the outputs limit the current for you
  - You set the output current with a single resistor for all 8 outputs
  - Only one resistor for 8 LEDs!

Example: STP08DP05



SDI/CLK shifts data into the 8-bit shift-register

LE moves data to the "data latch" so that it can be seen on the output

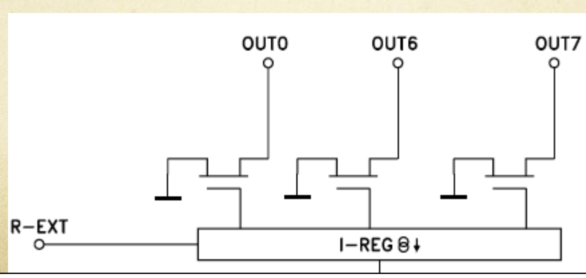
OE controls whether the data is enabled to drive the outputs

R-EXT sets the current for each output

Constant Current Source

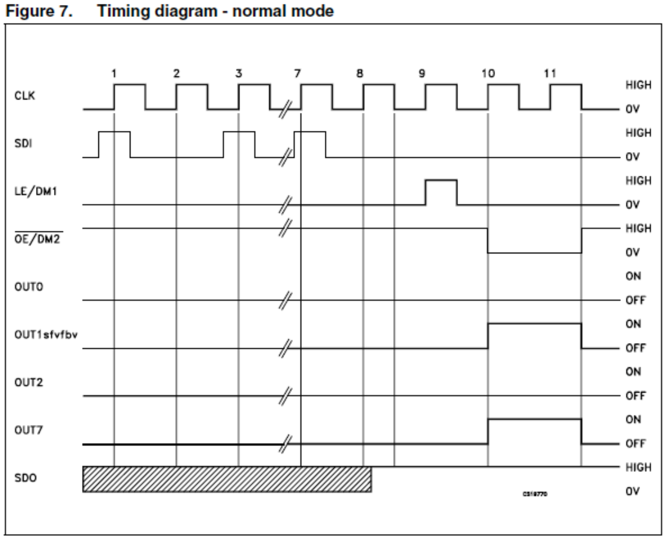
- Note that the constant current source only pulls to ground
  - So – LEDs connect to vdd…

+5v



Example: STP08DP05

Figure 7. Timing diagram - normal mode



Timing diagram shows shifting data in, one bit per clock

Data is transferred to output register on a high LE

Data shows up only when OE is low

This means you can dim all 8 LEDs using PWM on the OE signal

Arduino Code

- Arduino has a built-in function to shift data out for devices like this

Syntax

shiftOut(dataPin, clockPin, bitOrder, value)

Parameters

dataPin: the pin on which to output each bit (*int*)

clockPin: the pin to toggle once the **dataPin** has been set to the correct value (*int*)

bitOrder: which order to shift out the bits; either **MSBFIRST** or **LSBFIRST**.
(Most Significant Bit First, or, Least Significant Bit First)

value: the data to shift out. (*byte*)

Returns

None

Arduino Code

```
void shiftOut(uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder, byte val)
  {int i;
    for (i = 0; i < 8; i++) {
        if (bitOrder == LSBFIRST)
            digitalWrite(dataPin, !!(val & (1 << i)));
        else
            digitalWrite(dataPin, !!(val & (1 << (7 - i))));

        digitalWrite(clockPin, HIGH);
        digitalWrite(clockPin, LOW);
        }
  }
```

Arduino Code (74HC595)

```
int latchPin = 8;  //Pin connected to ST_CP of 74HC595
int clockPin = 12;  //Pin connected to SH_CP of 74HC595
int dataPin = 11; //Pin connected to DS of 74HC595

void setup() {  //set pins to output because they are addressed in the main loop  pinMode
(latchPin, OUTPUT);
pinMode(clockPin, OUTPUT);
pinMode(dataPin, OUTPUT);}

void loop() {  //count up routine
    for (int j = 0; j < 256; j++) {

        //ground latchPin and hold low for as long as you are transmitting
        digitalWrite(latchPin, LOW);
        shiftOut(dataPin, clockPin, LSBFIRST, j); // shift out the value of j

        //return the latch pin high to signal chip that it
        //no longer needs to listen for information
        digitalWrite(latchPin, HIGH);

    delay(1000);  }}
```

Arduino Code (STP08DP05)

```
int latchPin = 8;  //Pin connected to LE of STP08DP05
int clockPin = 12;  //Pin connected to CLK of STP08DP05
int dataPin = 11; //Pin connected to SDI of STP08DP05
Int OEPin = 10; //Pin connected to OEbar of STP08DP05

void setup() {  //set pins to output because they are addressed in the main loop
pinMode(latchPin, OUTPUT);
pinMode(clockPin, OUTPUT);
pinMode(dataPin, OUTPUT);
pinMode(OEPin, OUTPUT);}

void loop() {  //count up routine
    for (int j = 0; j < 256; j++) {
        //ground latchPin and hold low for as long as you are transmitting, OE pin is high…
        digitalWrite(latchPin, LOW);    digitalWrite(OEPin, HIGH);
        shiftOut(dataPin, clockPin, LSBFIRST, j);

        //return the latch pin high transfer data to output latch, OE low to light the LEDs
        digitalWrite(latchPin, HIGH);  digitalWrite(OEPin, LOW);
        delay(1000);  }}
```

Chaining Multiple Chips

Figure 2. Normal mode - block diagram

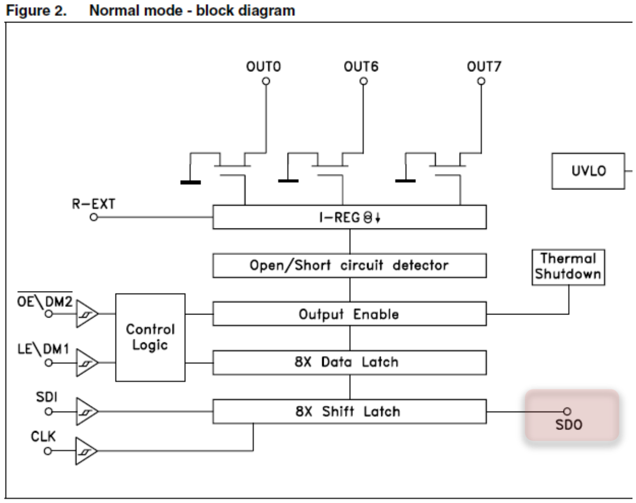
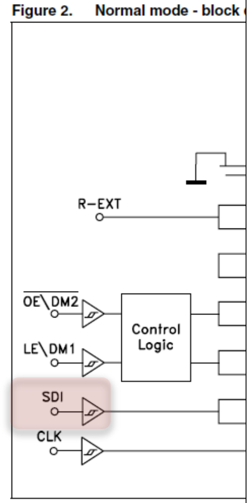


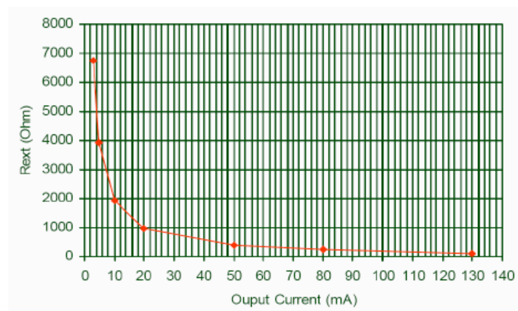
Figure 2. Normal mode - block



Choosing a Resistor

- I chose a 2k ohm resistor for around 10ma

Figure 11. Output current-R_{EXT} resistor



T_A = 25 °C, Vdrop = 0.3 V; 1.2 V, Iset = 3 mA; 5 mA; 10 mA; 20 mA; 50 mA; 80 mA, Max

Table 10. Output current-R_{EXT} resistor

Output current (mA)	3	5	10	20	50	80	130
Rext (Ω)	6740	3930	1913	963	386	241	124

Maximum output current capabilities setting was 130 mA applyingan Rext = 124 Ω

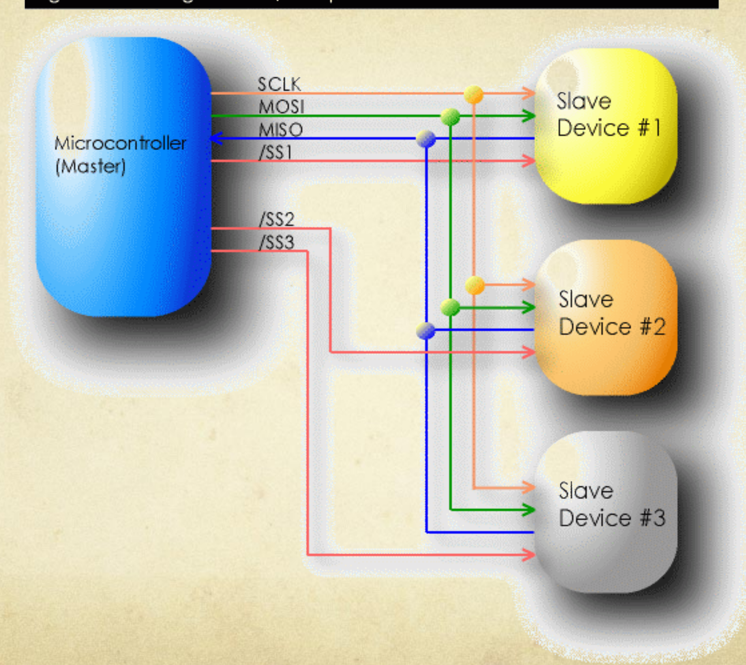
STP08DP05 Summary

- Easy chip to use
  - Use ShiftOut(…) to shift data to the chip
  - Can chain many together to drive lots of LEDs
- Just four wires from Arduino to external chip drives 8 LEDs
  - Clk and Data used to shiftOut() the data
  - LE goes high to capture the data
  - OE goes low to make the data appear (or for PWM)
- Constant-current drivers so only one resistor per chip
  - Simple on or off for each LED

SPI Interface

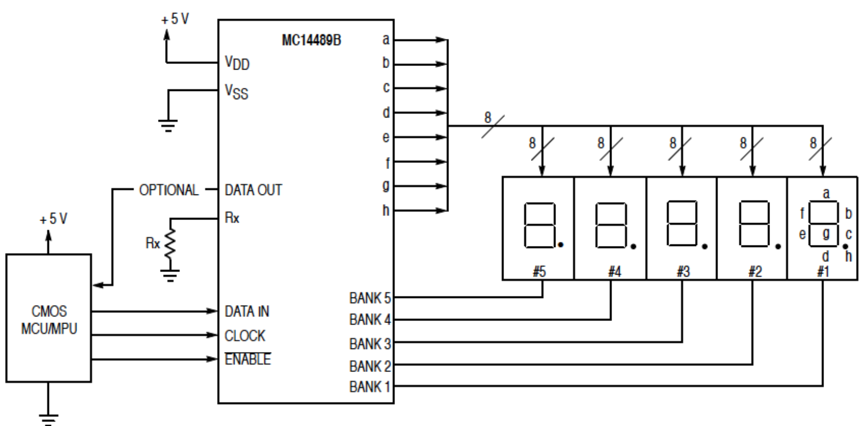
- Serial Peripheral Interface
  - Generalized version of previous example
  - "official" version has bidirectional data – you can read back data from the other device at the same time as you're sending
  - But, you can ignore that and use the same ShiftOut function if you like

Figure 3: SPI - Single Master, Multiple Slaves



Example: MC14489

- Designed to drive 5-digit 7-segment display
  - Cycles through each digit automatically
  - Could also drive 20 individual LEDs

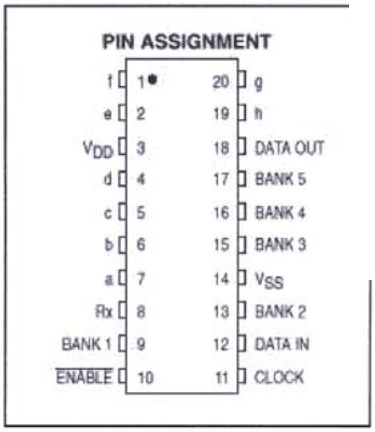


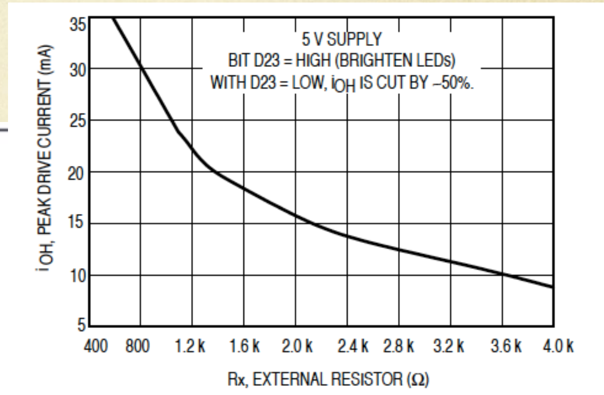
Example: MC14489

- Send in four bits per digit
- Three decoding modes
  - Hex
  - Special
  - No Decode

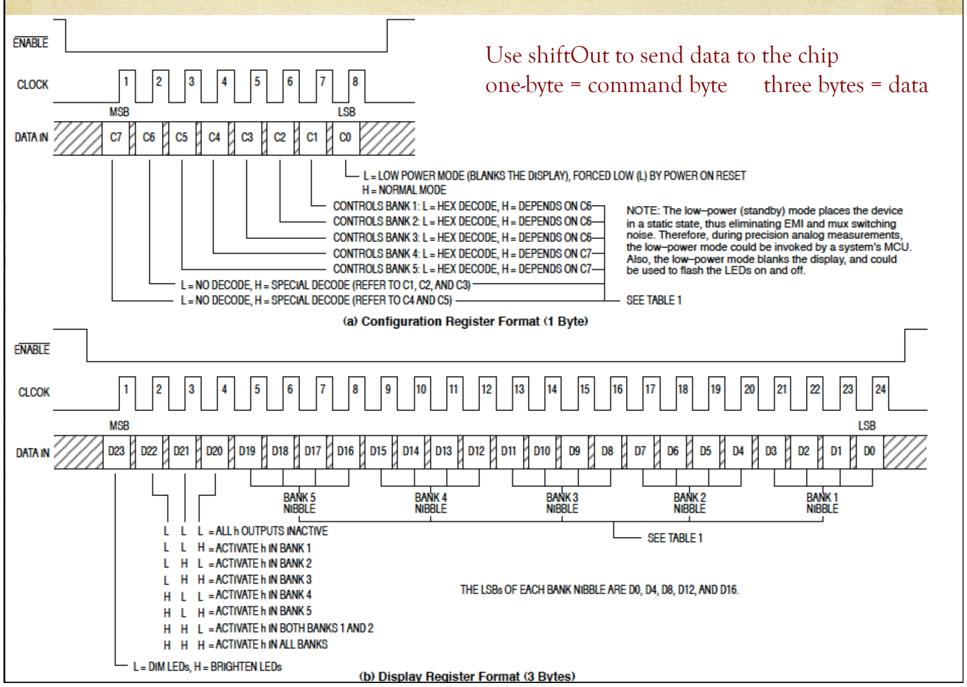
Bank Nibble Value		7-Segment Display Characters		Lamp Conditions: No Decode (1) (Invoked via Bits C1 to C7)			
Hexadecimal	Binary MSB LSB	Hex Decode (Invoked via Bits C1 to C5)	Special Decode (Invoked via Bits C1 to C7)	d	c	b	a
$0	L L L L	0					
$1	L L L H	1	c				on
$2	L L H L	2	H			on	
$3	L L H H	3	h			on	on
$4	L H L L	4	J		on		
$5	L H L H	5 (2)	L		on		on
$6	L H H L	6	n		on	on	
$7	L H H H	7	o		on	on	on
$8	H L L L	8 (3)	P	on			
$9	H L L H	9 (4)	r	on			on
$A	H L H L	A	U	on		on	
$B	H L H H	b	u	on		on	on
$C	H H L L	C	y	on	on		
$D	H H L H	d	-	on	on		on
$E	H H H L	E	=	on	on	on	
$F	H H H H	F	o	on	on	on	on

MC 14489





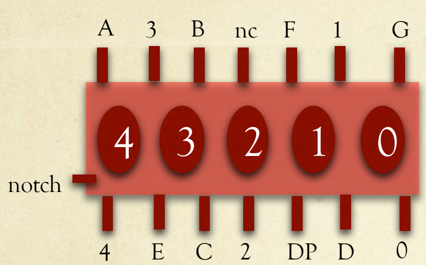
Use shiftOut to send data to the chip
one-byte = command byte three bytes = data

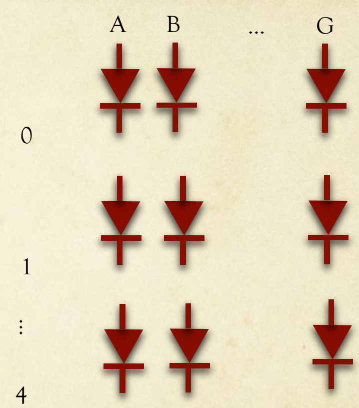


MC14489 Sumary

- Another convenient way to drive a bunch of LEDs
  - 5-digits of 7-segment numbers
  - or 20 individual LEDs
  - LEDs should be "common cathode" type
    - Anodes are the segments
    - Cathodes are the digits
    - Chip does the cycling between digits for you
- Single resistor sets current for all LEDs
- SPI interface (Clk, DataIn, Enable (active-low))
  - Slightly funky interface – you send 1 or 3 bytes and the chip figures out what you meant
  - Different numbers of bytes for chips connected in series

Aside: Vintage 7-seg displays

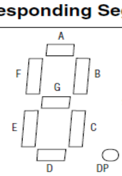




Common-Cathode LEDs

Vf = 1.6v

Table 6. No-Decode Mode Data Bits and Corresponding Segment Lines



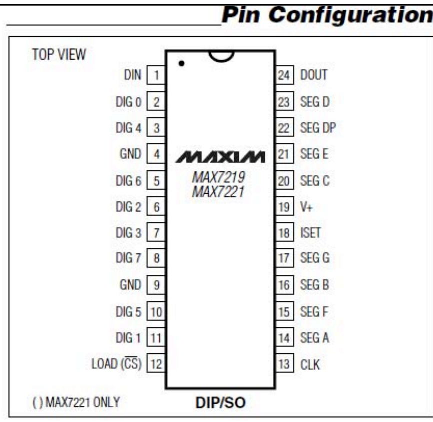
STANDARD 7-SEGMENT LED

	REGISTER DATA							
	D7	D6	D5	D4	D3	D2	D1	D0
Corresponding Segment Line	DP	A	B	C	D	E	F	G

Example: MAX 7219

- Display driver for 8-digits of 7-segment numbers
  - Can also be used for 8x8 array of LEDS
    - (i.e. 64 individual LEDs)
- Drives common-cathode LED digits or LED matrix
  - Cycles between each of 8 digits (or matrix rows) fast enough so they all look ON
- SPI interface
  - Slightly complicated command/data interface
  - Send address of internal register followed by data
  - Each SPI communication is 16 bits
  - Luckily, there's an Arduino library for the chip

Pin Configuration



Typical Application Circuit

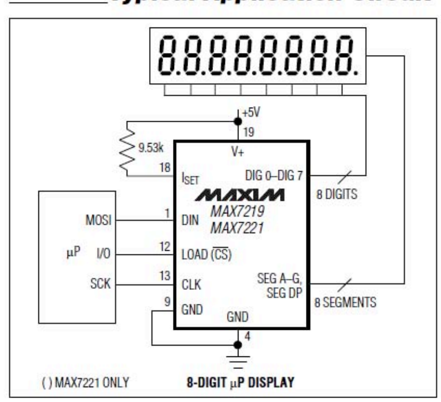
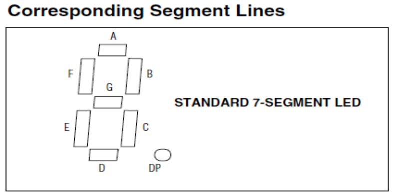
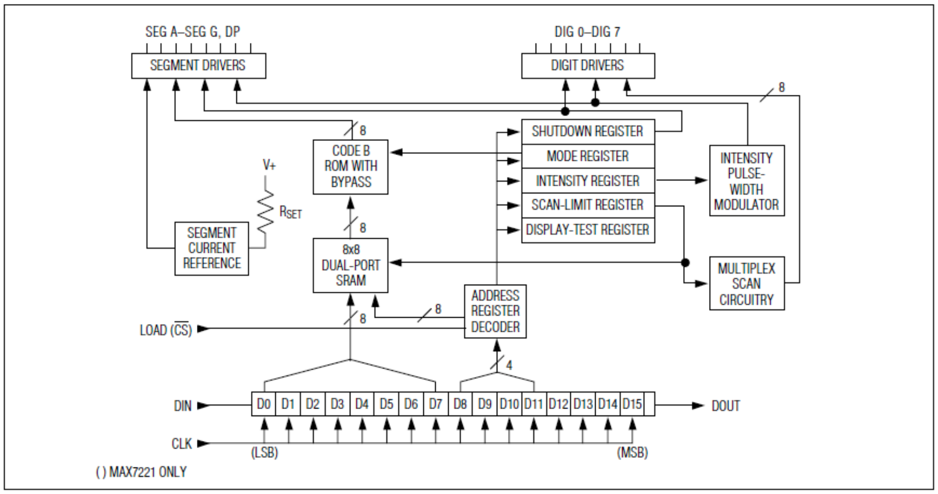


Table 6. No-Decode Mode Data Bits and Corresponding Segment Lines



	REGISTER DATA							
	D7	D6	D5	D4	D3	D2	D1	D0
Corresponding Segment Line	DP	A	B	C	D	E	F	G

Functional Diagram



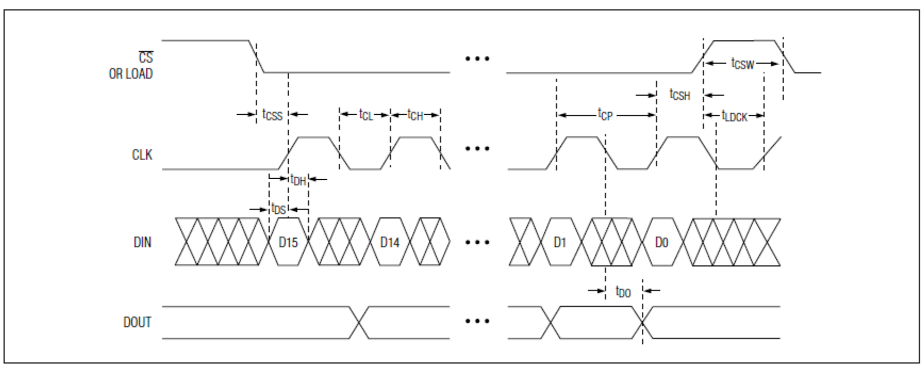


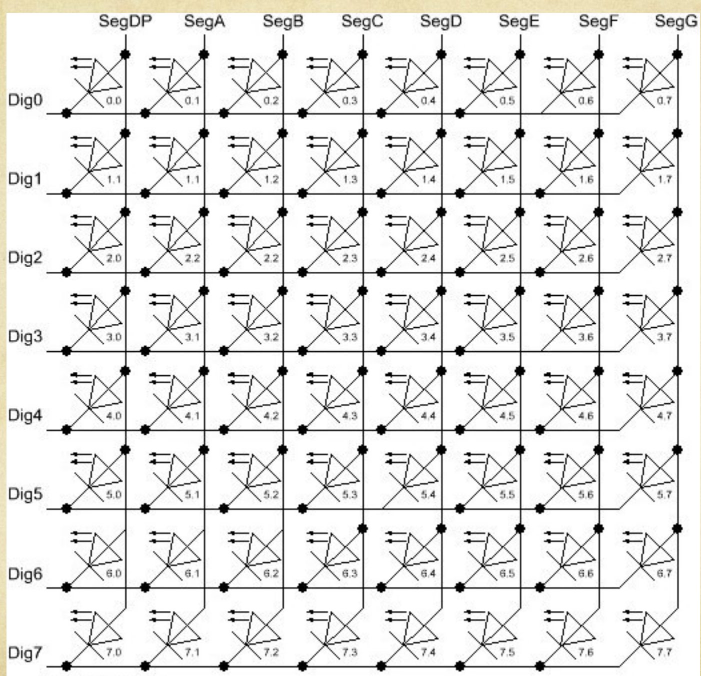
Figure 1. Timing Diagram

Table 1. Serial-Data Format (16 Bits)

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
X	X	X	X	ADDRESS				MSB			DATA				LSB

Table 2. Register Address Map

REGISTER	ADDRESS					HEX CODE
	D15–D12	D11	D10	D9	D8	
No-Op	X	0	0	0	0	0xX0
Digit 0	X	0	0	0	1	0xX1
Digit 1	X	0	0	1	0	0xX2
Digit 2	X	0	0	1	1	0xX3
Digit 3	X	0	1	0	0	0xX4
Digit 4	X	0	1	0	1	0xX5
Digit 5	X	0	1	1	0	0xX6
Digit 6	X	0	1	1	1	0xX7
Digit 7	X	1	0	0	0	0xX8
Decode Mode	X	1	0	0	1	0xX9
Intensity	X	1	0	1	0	0xXA
Scan Limit	X	1	0	1	1	0xXB
Shutdown	X	1	1	0	0	0xXC
Display Test	X	1	1	1	1	0xXF



Led Matrix 8x8

Common-Cathode LED array

MAX 7219

- On the one hand – just like MC14489
- On the other hand, more complex internal structure
  - Each SPI transfer needs to be 16 bits – address/data
- Two Arduino libraries available
  - Matrix – built-in to Arduino environment
  - LedControl – download from Playground – more complex control

Matrix Library

Matrix
Class for manipulating LED matrix displays connected to the Wiring I/O board.

write()
Write data to the display.

clear()
Clears the display screen.

setBrightness()
Set the brighness of the screen.

Matrix Library

- Matrix object is defined with Clk, Data, and Latch pins

Examples

```
#include <Binary.h>
#include <Sprite.h>
#include <Matrix.h>

Matrix myMatrix = Matrix(0, 2, 1);

void setup()
{
}

void loop()
{
  myMatrix.clear(); // clear display

  delay(1000);

  // turn some pixels on
  myMatrix.write(1, 5, HIGH);
  myMatrix.write(2, 2, HIGH);
  myMatrix.write(2, 6, HIGH);
  myMatrix.write(3, 6, HIGH);
  myMatrix.write(4, 6, HIGH);
  myMatrix.write(5, 2, HIGH);
  myMatrix.write(5, 6, HIGH);
  myMatrix.write(6, 5, HIGH);

  delay(1000);
}
```

LedControl Library

- Support for more than one MAX 7219
- Support for numbers and letters on 7-segment displays
- Support for rows and columns in an 8x8 matrix

LedControl Library

```
/* We start by including the library */
#include "LedControl.h"

/*  Make a new instance of an LedControl object
  * Params :
  * int dataPin    The pin on the Arduino where data gets shifted out (Din on MAX)
  * int clockPin   The pin for the clock (CLK on MAX)
  * int csPin      The pin for enabling the device (LD/CS on MAX)
  * int numDevices The maximum number of devices that can be controlled
  */
LedControl lc1=LedControl(12,11,10,1);
```

LedControl Library

```
void clearDisplay(int addr);
void setLed(int addr, int row, int col, boolean state);
void setRow(int addr, int row, byte value);
void setColumn(int addr, int col, byte value);
void setDigit(int addr, int digit, byte value, boolean dp);
void setChar(int addr, int digit, char value, boolean dp);

/*
 * Display a character on a 7-Segment display.
 * There are only a few characters that make sense here :
 * '0','1','2','3','4','5','6','7','8','9','0',
 * 'A','b','c','d','E','F','H','L','P',
 * '.','-','_',' '
 */
```

LedControl Library

```
//include this file so we can write down a byte in binary encoding
#include <binary.h>

//now setting the leds in the sixth column on the first device is easy
lc.setColumn(0,5,B00001111);

//now setting the leds from the third row on the first device is easy
lc.setRow(0,2,B10110000);

//switch on the led in the 3'rd row 8'th column
//and remember that indices start at 0!
lc.setLed(0,2,7,true);
//Led at row 0 second from left too
lc.setLed(0,0,1,false);
```

MAX 7219 – Setting Resistor

- This resistor goes to Vdd, NOT GND!
  - Sets current for each segment (LED)

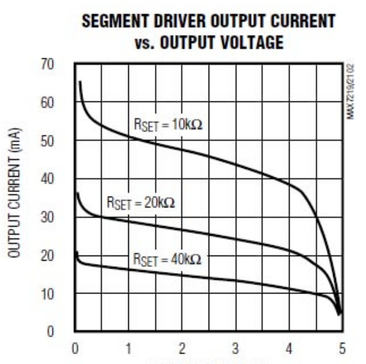
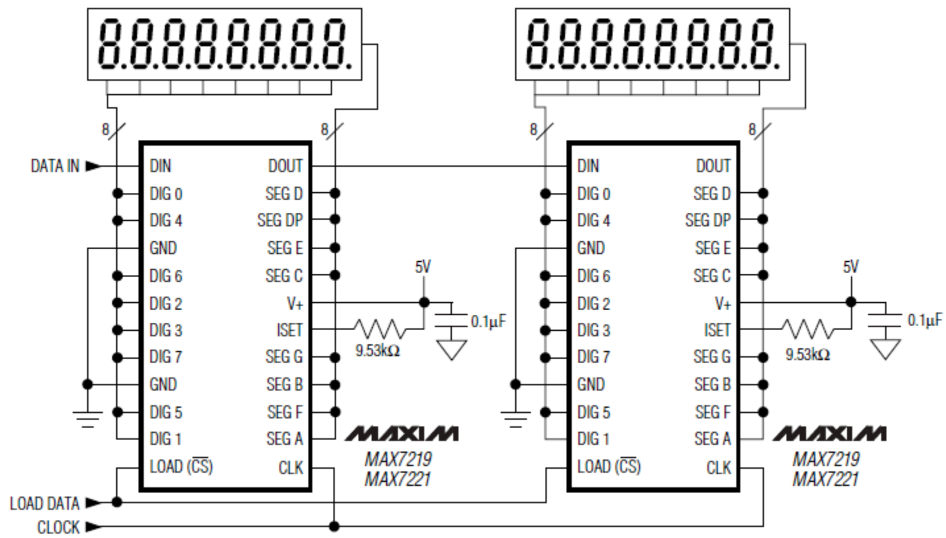


Table 11. R_{SET} vs. Segment Current and LED Forward Voltage

I_{SEG} (mA)	V_{LED} (V)				
	1.5	**2.0**	**2.5**	**3.0**	**3.5**
40	12.2	11.8	11.0	10.6	9.69
30	17.8	17.1	15.8	15.0	14.0
20	29.8	28.0	25.9	24.5	22.6
10	66.7	63.7	59.3	55.4	51.2

These values are in kOhms!!!

Multiple MAX chips



Multiple MAX Chips

- There is an important difference between the way the setRow() and the setColumn() methods update the Leds:
  - setRow() only needs to send a single int-value to the MAX72XX in order to update all 8 Leds in a row.
  - setColumn() uses the setLed()-method internally to update the Leds. The library has to send 8 ints to the driver, so there is a performance penalty when using setColumn().
  - You won't notice that visually when using only 1 or 2 cascaded Led-boards, but if you have a long queue of devices (6..8) which all have to be updated at the same time, that could lead to some delay that is actually visible.

MAX 7219 Summary

- Drives more LEDs than the STP08DP05 or MC14489
- Similar to MC14489, but for 8 digits or 64 LEDs
  - Designed for common-cathode LED arrays
    - Set the anodes to true and false
    - Pull down the cathodes in sequence
  - Uses time-multiplexing to drive them all
  - Also supports 7-segment displays
  - Slightly more complex interface

Atmel SPI Support

- The Atmel ATMega328 chip supports hardware-controlled SPI
  - Could be faster than ShiftOut function
  - Uses built-in SPI register on ATMega328
    - Set up the SPI functionality by setting bits in a control register
    - Write data to the SPI output register (MOSI) which causes the transfer to happen
    - A bit gets set in the control register when it's done

Atmel SPI Support

Figure 18-1. SPI Block Diagram[1]

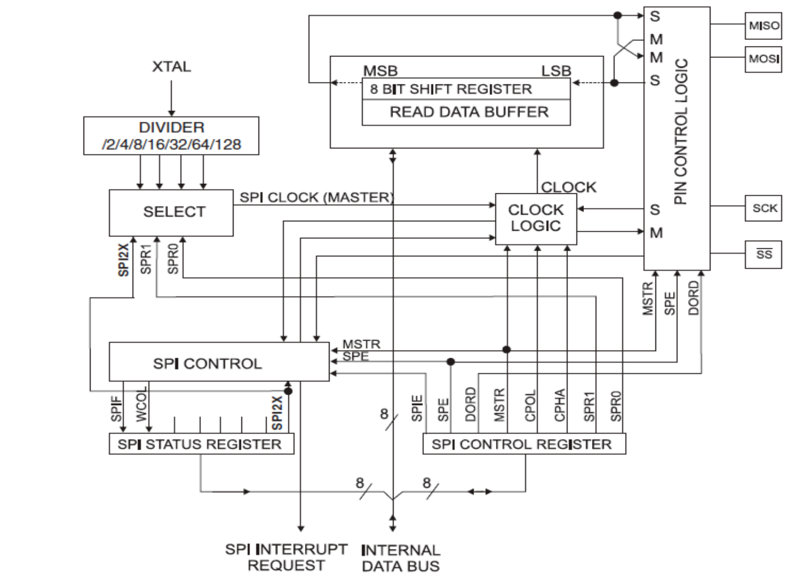
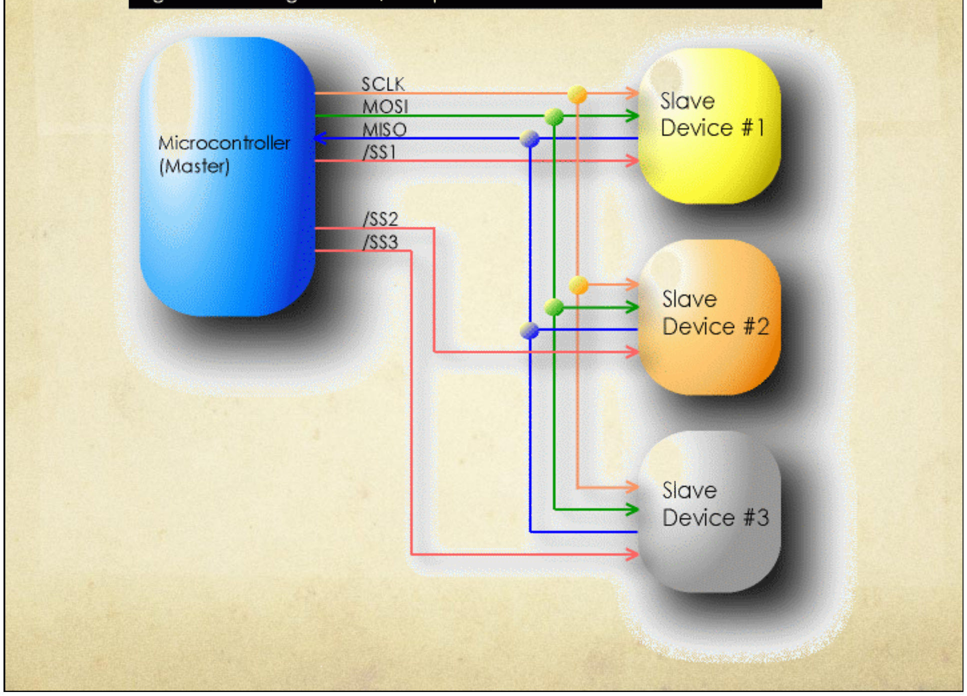


Figure 3: SPI - Single Master, Multiple Slaves



SPI library setup

Spi Library

This library provides functions for transferring information using the Serial Peripheral Interface (SPI). The SPI interface is automatically initialized when the Spi library is included in a sketch. It sets the following digital I/O pins:

```
pin 13  SCK     SPI clock
pin 12  MISO    SPI master in, slave out
pin 11  MOSI    SPI master out, slave in
pin 10  SS      SPI slave select
```

The default SPI configuation is as follows:

```
SPI Master enabled
MSB of the data byte transmitted first
SPI mode 0 (CPOL = 0, CPHA = 0)
SPI clock frequency = system clock / 4
```

mode(byte config)

Sets the SPI configuration register. Only required if the default configuration described above must be modified. The SPE (SPI enabled) and MSTR (SPI master) bits are always set. If there are multiple SPI devices on the bus which require different SPI configurations, this function can be called before accessing each different device type to set the appropriate configuration.

```
Example:
Spi.mode((1<<CPOL) | (1 << CPHA)); // set SPI mode 3
or
Spi.mode((<<SPR0));                // set SPI clock to system clock / 16
```

byte transfer(byte b)

Sends and receives a byte from the SPI bus.

```
Example:
n = Spi.transfer(0x2A);            // sends the byte 0x2A
                                   // and returns the byte received
```

byte transfer(byte b, byte delay)

Delays for a number of microseconds, then sends and receives a byte from the SPI bus. This function is used if there are timing considerations associated with the data transfer.

```
Example:
n = Spi.transfer(0x2A, 2);         // waits 2 usec, then sends the byte 0x2A
                                   // and returns the byte received
```

Transfer a byte using SPI

```
char spi_transfer(volatile char data)
{
  SPDR = data;                    // Start the transmission
  while (!(SPSR & (1<<SPIF)))     // Wait for the end of the transmission
  {
  };
  return SPDR;                    // return the received byte
}
```

Magic stuff happens here: By writing data to the SPDR register, the SPI transfer is Started. When the transfer is complete, the system raises the SPIF bit in the SPSR Status register. The data that comes back from the slave is in SPDR when you're Finished.

SPI Details

18.5.1 SPCR – SPI Control Register

Bit	7	6	5	4	3	2	1	0	
0x2C (0x4C)	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – SPIE: SPI Interrupt Enable**

This bit causes the SPI interrupt to be executed if SPIF bit in the SPSR Register is set and the if the Global Interrupt Enable bit in SREG is set.

- **Bit 6 – SPE: SPI Enable**

When the SPE bit is written to one, the SPI is enabled. This bit must be set to enable any SPI operations.

- **Bit 5 – DORD: Data Order**

When the DORD bit is written to one, the LSB of the data word is transmitted first.

When the DORD bit is written to zero, the MSB of the data word is transmitted first.

- **Bit 4 – MSTR: Master/Slave Select**

This bit selects Master SPI mode when written to one, and Slave SPI mode when written logic zero. If $\overline{SS}$ is configured as an input and is driven low while MSTR is set, MSTR will be cleared, and SPIF in SPSR will become set. The user will then have to set MSTR to re-enable SPI Master mode.

SPI Details

- **Bit 3 – CPOL: Clock Polarity**

When this bit is written to one, SCK is high when idle. When CPOL is written to zero, SCK is low when idle. Refer to Figure 18-3 and Figure 18-4 for an example. The CPOL functionality is summarized below:

Table 18-3. CPOL Functionality

CPOL	Leading Edge	Trailing Edge
0	Rising	Falling
1	Falling	Rising

- **Bit 2 – CPHA: Clock Phase**

The settings of the Clock Phase bit (CPHA) determine if data is sampled on the leading (first) or trailing (last) edge of SCK. Refer to Figure 18-3 and Figure 18-4 for an example. The CPOL functionality is summarized below:

Table 18-4. CPHA Functionality

CPHA	Leading Edge	Trailing Edge
0	Sample	Setup
1	Setup	Sample

SPI Details

- **Bits 1, 0 – SPR1, SPR0: SPI Clock Rate Select 1 and 0**

These two bits control the SCK rate of the device configured as a Master. SPR1 and SPR0 have no effect on the Slave. The relationship between SCK and the Oscillator Clock frequency f_{osc} is shown in the following table:

Table 18-5. Relationship Between SCK and the Oscillator Frequency

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

SPI Details

SPSR – SPI Status Register

Bit	7	6	5	4	3	2	1	0	
0x2D (0x4D)	SPIF	WCOL	–	–	–	–	–	SPI2X	SPSR
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – SPIF: SPI Interrupt Flag**

When a serial transfer is complete, the SPIF Flag is set. An interrupt is generated if SPIE in SPCR is set and global interrupts are enabled. If $\overline{SS}$ is an input and is driven low when the SPI is in Master mode, this will also set the SPIF Flag. SPIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, the SPIF bit is cleared by first reading the SPI Status Register with SPIF set, then accessing the SPI Data Register (SPDR).

- **Bit 6 – WCOL: Write COLlision Flag**

The WCOL bit is set if the SPI Data Register (SPDR) is written during a data transfer. The WCOL bit (and the SPIF bit) are cleared by first reading the SPI Status Register with WCOL set, and then accessing the SPI Data Register.

- **Bit 0 – SPI2X: Double SPI Speed Bit**

When this bit is written logic one the SPI speed (SCK Frequency) will be doubled when the SPI is in Master mode (see Table 18-5). This means that the minimum SCK period will be two CPU clock periods. When the SPI is configured as Slave, the SPI is only guaranteed to work at $f_{osc}/4$ or lower.

SPI Summary

- Very general way to send serial information from Arduino to another chip
  - DIY version: ShiftOut
  - Fancy version: SPI library
  - Both do pretty much the same thing
  - Make sure your chip “speaks” SPI
  - If it “speaks” I^2C, a whole different ball of wax…

I2C – a.k.a. TWI

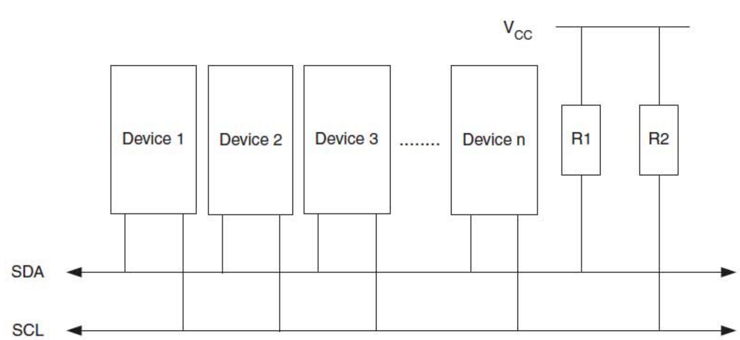
- Uses only two wires to communicate
  - Simpler?
- Each wire is bidirectional
- Can address up to 128 devices on a single I2C bus
- Actually more complex…

I2C – a.k.a. TWI

21.2 2-wire Serial Interface Bus Definition

The 2-wire Serial Interface (TWI) is ideally suited for typical microcontroller applications. The TWI protocol allows the systems designer to interconnect up to 128 different devices using only two bi-directional bus lines, one for clock (SCL) and one for data (SDA). The only external hardware needed to implement the bus is a single pull-up resistor for each of the TWI bus lines. All devices connected to the bus have individual addresses, and mechanisms for resolving bus contention are inherent in the TWI protocol.

Figure 21-1. TWI Bus Interconnection



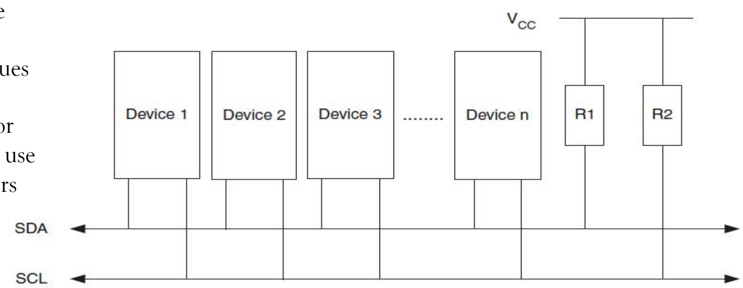
21.2 2-wire Serial Interface Bus Definition

The 2-wire Serial Interface (TWI) is ideally suited for typical microcontroller applications. The TWI protocol allows the systems designer to interconnect up to 128 different devices using only two bi-directional bus lines, one for clock (SCL) and one for data (SDA). The only external hardware needed to implement the bus is a single pull-up resistor for each of the TWI bus lines. All devices connected to the bus have individual addresses, and mechanisms for resolving bus contention are inherent in the TWI protocol.

Figure 21-1. TWI Bus Interconnection

1.8k, 4.7k. 10k are commonly used pullup resistor values

The wire library for Arduino can even use the built-in resistors on the AVR



C_i[1]	Capacitance for each I/O Pin		–	10	pF
f_{SCL}	SCL Clock Frequency	f_{CK}[4] > max($16f_{SCL}$, 250kHz)[5]	0	400	kHz
Rp	Value of Pull-up resistor	$f_{SCL} \le 100$ kHz	$\frac{V_{CC} - 0{,}4V}{3mA}$	$\frac{1000ns}{C_b}$	Ω
		$f_{SCL} > 100$ kHz	$\frac{V_{CC} - 0{,}4V}{3mA}$	$\frac{300ns}{C_b}$	Ω

Address vs. Data

Figure 21-4. Address Packet Format

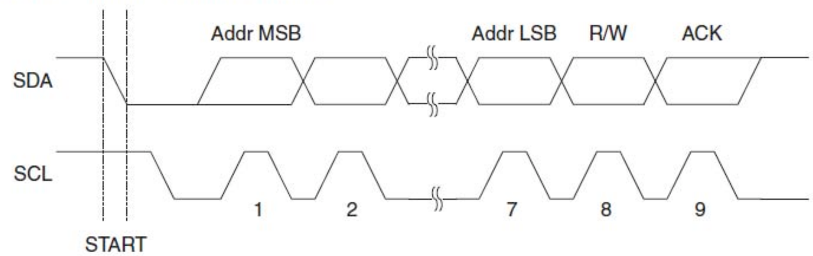
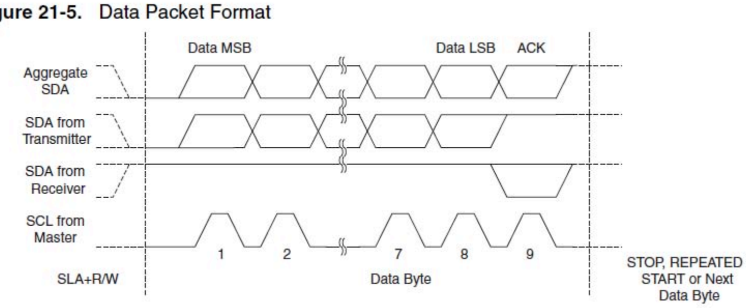
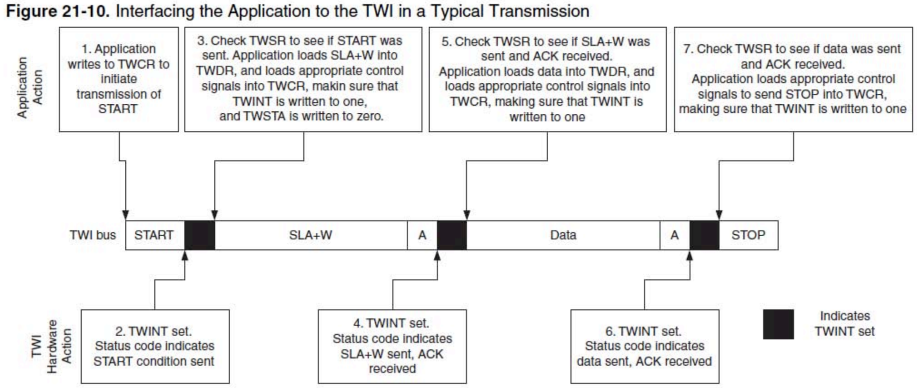


Figure 21-5. Data Packet Format



Using I2C/TWI

Figure 21-10. Interfacing the Application to the TWI in a Typical Transmission



Luckily Arduino comes with an I2C library!

Wire Library

Buy | Download | Getting Started | Learning | Reference | Hardware | FAQ   Blog » | Forum » | Playground »

Reference Language (extended) | Libraries | Comparison | Changes

Wire Library

This library allows you to communicate with I2C / TWI devices. On most Arduino boards, SDA (data line) is on analog input pin 4, and SCL (clock line) is on analog input pin 5. On the Arduino Mega, SDA is digital pin 20 and SCL is 21.

Functions

- begin()
- begin(address)
- requestFrom(address, count)
- beginTransmission(address)
- endTransmission()
- send()
- byte available()
- byte receive()
- onReceive(handler)
- onRequest(handler)

Note

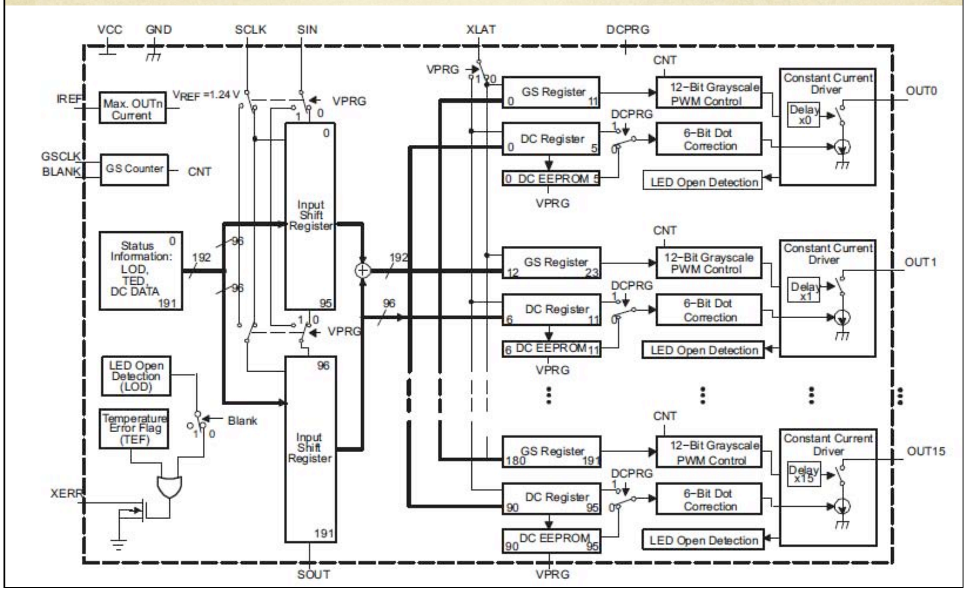
There are both 7- and 8-bit versions of I2C addresses. 7 bits identify the device, and the eighth bit determines if it's being written to or read from. The Wire library uses 7 bit addresses throughout. If you have a datasheet or sample code that uses 8 bit address, you'll want to drop the low bit (i.e. shift the value one bit to the right), yielding an address between 0 and 127.

```
#include <Wire.h>
    // TWI (I2C) sketch to communicate with the LIS3LV02DQ accelerometer
    // Using the Wire library (created by Nicholas Zambetti)
    // On the Arduino board, Analog In 4 is SDA, Analog In 5 is SCL
    // The Wire class handles the TWI transactions, abstracting the nitty-gritty to make
    // prototyping easy.
    void setup(){
    pinMode(9, OUTPUT);
    digitalWrite(9, HIGH);
    Serial.begin(9600);

    Wire.begin(); // join i2c bus (address optional for master)
    Wire.beginTransmission(0x1D);
    Wire.send(0x20); // CTRL_REG1 (20h)
    Wire.send(0x87); // Device on, 40hz, normal mode, all axis's enabled
    Wire.endTransmission();
    }

// Switch to Wii Nunchuck Slides!
```

Roll your Own Interface

- TLC 5940 – 16-output LED driver with PWM on each output
  - 12-bits of PWM = 4096 levels of brightness
  - 16 bits with 12-bits of PWM each = 192 bits to send for each change of the LEDs
  - Communicates with a serial protocol, so you can chain them together
  - BUT, it's not SPI or I2C!
    - Rats…

TLC 5940



TLC 5940

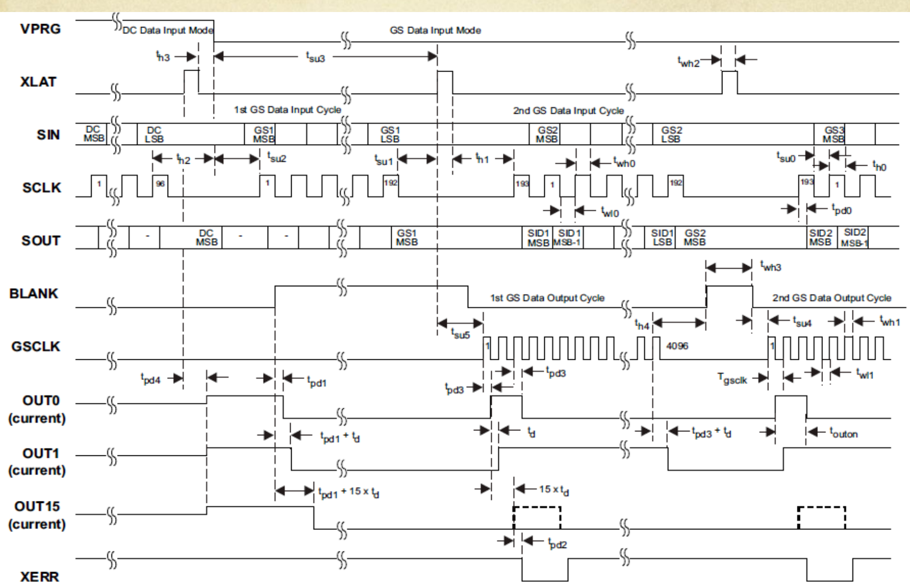


Figure 11. Serial Data Input Timing Chart

PWM…

- Based on the “grayscale counter” which runs at a frequency that you send the chip

GRAYSCALE PWM OPERATION

The grayscale PWM cycle starts with the falling edge of BLANK. The first GSCLK pulse after BLANK goes low increases the grayscale counter by one and switches on all OUTn with grayscale value not zero. Each following rising edge of GSCLK increases the grayscale counter by one. The TLC5940 compares the grayscale value of each output OUTn with the grayscale counter value. All OUTn with grayscale values equal to the counter values are switched off. A BLANK=H signal after 4096 GSCLK pulses resets the grayscale counter to zero and completes the grayscale PWM cycle (see Figure 21). When the counter reaches a count of FFFh, the counter stops counting and all outputs turn off. Pulling BLANK high before the counter reaches FFFh immediately resets the counter to zero.

This means there are some relatively complex timings and relationships
Between the different signals that you have to get right

The Arduino 5940 library uses interrupt-driven control to get this right…

PWM…

- Based on the “grayscale counter” which runs at a frequency that you send the chip

GRAYSCALE PWM OPERATION

The grayscale PWM cycle starts with the falling edge of BLANK. The first GSCLK pulse after BLANK goes low increases the grayscale counter by one and switches on all OUTn with grayscale value not zero. Each following rising edge of GSCLK increases the grayscale counter by one. The TLC5940 compares the grayscale value of each output OUTn with the grayscale counter value. All OUTn with grayscale values equal to the counter values are switched off. A BLANK=H signal after 4096 GSCLK pulses resets the grayscale counter to zero and completes the grayscale PWM cycle (see Figure 21). When the counter reaches a count of FFFh, the counter stops counting and all outputs turn off. Pulling BLANK high before the counter reaches FFFh immediately resets the counter to zero.

8196 8196

Timer

2 2 2

1 1

Blank

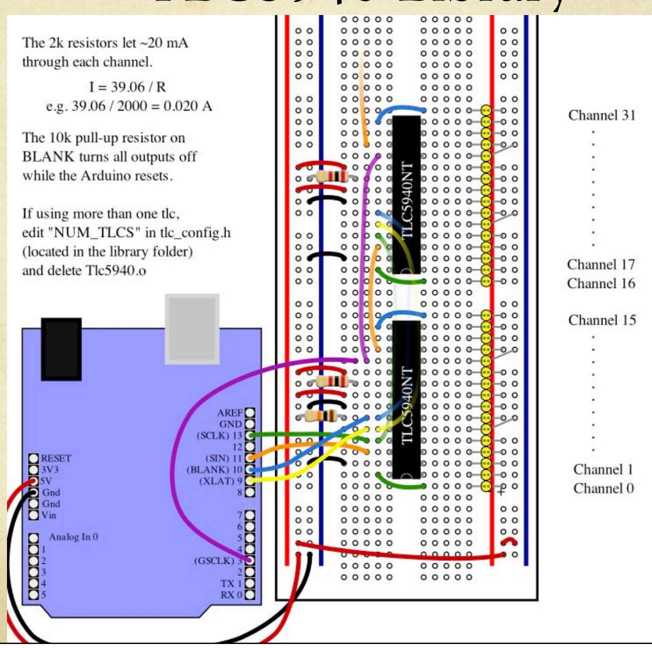
XLAT

TLC5940 Library

First, for a serial interfaced part it has a rather large number of signals. Fortunately we can ignore many of them if we wish.

- XERR : open collector, wire or-ed output that lets you know a TLC5940 is over heated or has a burnt out LED. We can ignore this as it will always be on unless you have current using elements on all of the outputs.
- SOUT: serial data out from the TLC5940. Unless you wish to try to read the error bits you do not need this to come to the Arduino. If you have more than one TLC5940 this is the line you daisy chain to the SIN of the next package.
- DCPRG: this selects the source of the current limiter register, you could just tie it high.
- XLAT: you will need this to latch data after shifting.
- SCLK: you will need this to shift data.
- SIN: serial in to TLC5940, this is the output from the Arduino.
- VPRG: you need this to select either the current limit registers or the duty cycle registers for writing.
- GSCLK: this is the clock for the PWM. We will reprogram TIMER2 in the Arduino to make this signal. That will cost us the native PWM on that timer, digital 11 on a mega8, 11 and 3 on a mega168.
- BLANK: this marks the end of a PWM cycle in addition to blanking the output. We will reprogram TIMER1 to generate this signal. That will cost us the native PWMs on digital 9 and digital 10. (Tie a real, physical pull-up resistor on this line to keep things blanked while your Arduino boots. Depending on your hardware, it is possible that the TLC5940 would come up in a configuration that would dissipate too much power.)

TLC5940 Library



TLC5940 Library

Hardware Setup

The basic hardware setup is explained at the top of the Examples. A good place to start would be the BasicUse Example. (The examples are in File->Sketchbook->Examples->Library-Tlc5940).

All the options for the library are located in **tlc_config.h**, including **NUM_TLCS**, what pins to use, and the PWM period. After changing **tlc_config.h**, be sure to delete the Tlc5940.o file in the library folder to save the changes.

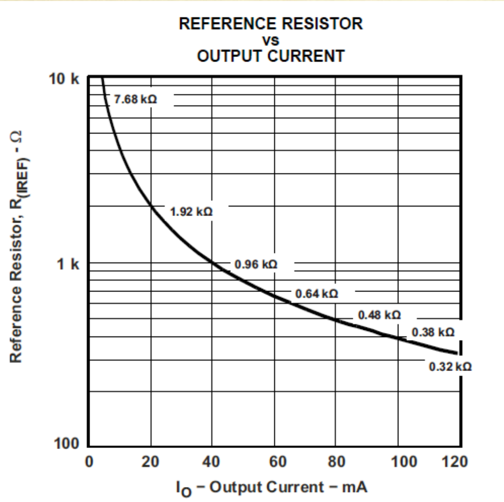
Library Reference

Core Functions (see the BasicUse Example and **Tlc5940**):

- **Tlc.init(int initialValue (0-4095))** - Call this is to setup the timers before using any other Tlc functions. initialValue defaults to zero (all channels off).
- **Tlc.clear()** - Turns off all channels (Needs Tlc.update())
- **Tlc.set(uint8_t channel (0-(NUM_TLCS * 16 - 1)), int value (0-4095))** - sets the grayscale data for channel. (Needs Tlc.update())
- **Tlc.setAll(int value(0-4095))** - sets all channels to value. (Needs Tlc.update())
- **uint16_t Tlc.get(uint8_t channel)** - returns the grayscale data for channel (see set).
- **Tlc.update()** - Sends the changes from any Tlc.clear's, Tlc.set's, or Tlc.setAll's.

TLC5940 – setting the resistor

- One resistor sets current for all channels



Min = 5ma
Max = 120ma

$$I_{max} = \frac{V_{(IREF)}}{R_{(IREF)}} \times 31.5$$

where:

$V_{(IREF)}$ = 1.24 V

$R_{(IREF)}$ = User-selected external resistor.

TLC5940 Summary

- Easy to use – if you use the tlc5940 library!
- Can also use for servo control
  - Use the PWM channels to drive servos
  - Remember about power issues!
  - Separate tlc5940 servo library
  - Resets some timing to get the servo timing right…

TLC servo functions. More...

```
#include <avr/io.h>
#include "Tlc5940.h"
```

Go to the source code of this file.

Defines

#define	SERVO_MAX_ANGLE 180 The maximum angle of the servo.
#define	SERVO_MIN_WIDTH 204 The 1ms pulse width for zero degrees (0 - 4095).
#define	SERVO_MAX_WIDTH 410 The 2ms pulse width for 180 degrees (0 - 4095).
#define	SERVO_TIMER1_TOP 20000 The top value for XLAT and BLANK pulses.
#define	SERVO_TIMER2_TOP 77 The top value for GSCLK pulses.

Functions

void	tlc_initServos (uint8_t initAngle) Initializes the tlc.
void	tlc_setServo (TLC_CHANNEL_TYPE channel, uint8_t angle) Sets a servo on channel to angle.
uint8_t	tlc_getServo (TLC_CHANNEL_TYPE channel) Gets the current angle that channel is set to.
uint16_t	tlc_angleToVal (uint8_t angle) Converts and angle (0 - SERVO_MAX_ANGLE) to the inverted tlc channel value (4095 - 0).
uint8_t	tlc_valToAngle (uint16_t value) Converts an inverted tlc channel value (4095 - 0) into an angle (0 - SERVO_MAX_ANGLE).

Servos

```
/*
    This sketch sweeps a servo on channel 0.

    To connect a servo:
     1. Put a 2k-5k pull-up resistor (R0 below; I've tried with 3.3k) between the
          servo control output pin and +5v.
     2. Connect that same pin to the servo's control line like so

        servo data pin
              |     _____
     OUTn ----+----[_____]---+5v
                      R0

    Steve Pomeroy <steve ~AT~ staticfree.info>, 2009-01-20 */

#include "Tlc5940.h"
#include "tlc_servos.h"

#define SERVO_CHANNEL   0
#define DELAY_TIME      20

void setup()
{
  tlc_initServos();  // Note: this will drop the PWM freqency down to 50Hz.
}

void loop()
{
  for (int angle = 0; angle < 180; angle++) {
    tlc_setServo(SERVO_CHANNEL, angle);
    Tlc.update();
    delay(DELAY_TIME);
  }
  for (int angle = 180; angle >= 0; angle--) {
    tlc_setServo(SERVO_CHANNEL, angle);
    Tlc.update();
    delay(DELAY_TIME);
  }
}
```

Issue with 5940 and servos?

Servos

Hobby servos are driven by short, high, pulses every 10-40ms. The constants have been chosen in the following sample code to allow you to drive servos as well as LEDs or motors. Because the servos use high pulses and the TLC5940 is active low on the outputs the useful servo values are all at the upper end.

- 3993, 0xf99: 500 microseconds
- 3584, 0xe00: 1500 microseconds
- 3168, 0xc60: 2500 microsecond

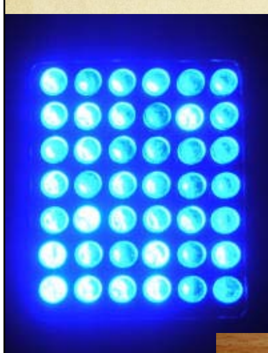
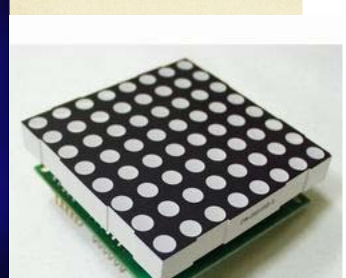
That will drive my cheap HS-311 through about 180 degrees. Do be careful not to hold your servo past its limits. It will probably use a lot of power and possibly burn up its motor if you try to hold it there. Remember: the outputs only sink current, you will want a pull-up resistor on here.

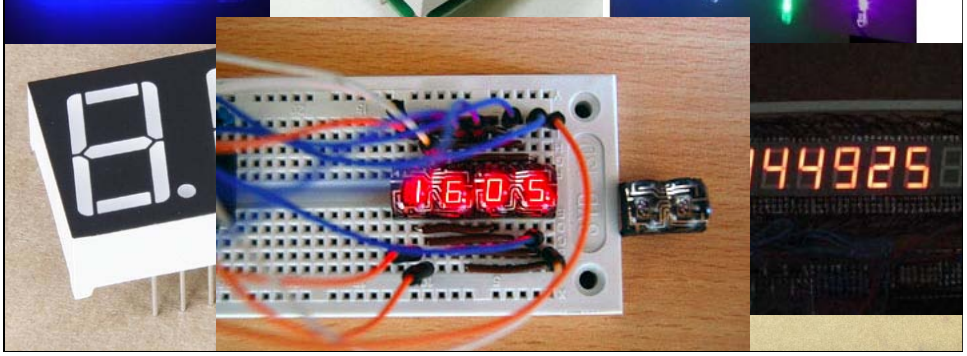
Use current-limiting feature of TLC5940 for this? More study may be in order…

Summary

- There are lots of ways to interface with other chips
  - shiftOut() – simple serial
    - Output only
  - SPI – standard serial protocol – three wires CLK, DATA, En
    - Can be bi-directional
  - I2C / TWI – two wire protocol – requires a little more complex addressing and protocol, and pullup resistors
    - Can also be bidirectional
  - Non-standard serial – read the data sheet carefully!

LEDs



LED Driver Chips

- 74HC595 – shift register with output latch ($0.62)
  - Drives 8 LEDs, but each one needs a current-limiting resistor
- STP08DP05 – Drives 8 LEDs with constant-current sources ($1.82)
  - SPI interface
- MC14489 – drives 5-digits of 7-segment display or 20 LEDs ($4.50)
  - Common-cathode LED arrays or digits – SPI interface
- MAX 7219/7221 – drives 8 digits of 7-segment display or 84 LEDs (8x8 array)($10.86)
  - Common-cathode LED arrays or digits – SPI interface
- TLC5940/5941– Drives 16 LEDs with each LED having 12 bits of PWM brightness ($3.50)
  - Complicated communication protocol
  - Can also be used for multiple servos