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 off 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.

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
  - **I²C/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...

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

![Diagram of Shifting w/Latch]

Arduino

- +5V
- 0V

External Chip

- +5V
- 0V

OE

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

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

```cpp
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

```c
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)

```c
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
  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)

```c
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++) {
        digitalWrite(latchPin, LOW);    digitalWrite(OEPin, HIGH);
        shiftOut(dataPin, clockPin, LSBFIRST, j);
        digitalWrite(latchPin, HIGH);  digitalWrite(OEPin, LOW);
        delay(1000);  }
}
```

Chaining Multiple Chips

[Diagram of STP08DP05 block diagram]
Choosing a Resistor

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

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](image)
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

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary MSB LSB</th>
<th>Hex Decode (Invoked via Bits C1 to C5)</th>
<th>Special Decode (Invoked via Bits C1 to C7)</th>
<th>Lamp Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>L L L L</td>
<td>0</td>
<td>c</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$1$</td>
<td>L L L H</td>
<td>1</td>
<td>H</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$2$</td>
<td>L L H L</td>
<td>2</td>
<td>h</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$3$</td>
<td>L H H L</td>
<td>3</td>
<td></td>
<td>on on on on on</td>
</tr>
<tr>
<td>$4$</td>
<td>L H L L</td>
<td>4</td>
<td>j</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$5$</td>
<td>L H H H</td>
<td>5</td>
<td>l</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$6$</td>
<td>L H H H</td>
<td>6</td>
<td>n</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$7$</td>
<td>H H H H</td>
<td>7</td>
<td>o</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$8$</td>
<td>H L L L</td>
<td>8</td>
<td>p</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$9$</td>
<td>H L L H</td>
<td>9</td>
<td>r</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$A$</td>
<td>H L H L</td>
<td>A</td>
<td>u</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$B$</td>
<td>H L H H</td>
<td>B</td>
<td>v</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$C$</td>
<td>H H L L</td>
<td>C</td>
<td>y</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$D$</td>
<td>H H L H</td>
<td>D</td>
<td>-</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$E$</td>
<td>H H H L</td>
<td>E</td>
<td>-</td>
<td>on on on on on</td>
</tr>
<tr>
<td>$F$</td>
<td>H H H H</td>
<td>F</td>
<td>z</td>
<td>on on on on on</td>
</tr>
</tbody>
</table>
Use shiftOut to send data to the chip
one-byte = command byte  three bytes = data
MC14489 Summary

- 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

<table>
<thead>
<tr>
<th>Table 6. No-Decode Mode Data Bits and Corresponding Segment Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGISTER DATA</td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>D7</td>
</tr>
<tr>
<td>Corresponding Segment Line</td>
</tr>
</tbody>
</table>

Common-Cathode LEDs
Vf = 1.6v
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
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 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|---------|-----|------|-----|
| X   | X   | X   | X   | X   | X   | X   | X   | X   | X   | X   | X   | X   | X   | X   | X   | X   | X   | X   | ADDRESS | MSB | DATA | LSB |

Figure 1. Timing Diagram
Table 2. Register Address Map

<table>
<thead>
<tr>
<th>REGISTER</th>
<th>ADDRESS</th>
<th>HEX CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D15-D12</td>
<td>D11 D10 D9 D8</td>
</tr>
<tr>
<td>No-Op</td>
<td>X</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>Digit 0</td>
<td>X</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>Digit 1</td>
<td>X</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>Digit 2</td>
<td>X</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>Digit 3</td>
<td>X</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>Digit 4</td>
<td>X</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>Digit 5</td>
<td>X</td>
<td>1 1 1 0</td>
</tr>
<tr>
<td>Digit 6</td>
<td>X</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>Digit 7</td>
<td>X</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>Decode Mode</td>
<td>X</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>Intensity</td>
<td>X</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>Scan Limit</td>
<td>X</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>Shutdown</td>
<td>X</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>Display Test</td>
<td>X</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

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 brightness of the screen.
Matrix Library

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

Examples

```c
#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, 8, 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);

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. RSET vs. Segment Current and LED Forward Voltage

<table>
<thead>
<tr>
<th>Iseg (mA)</th>
<th>1.5</th>
<th>2.0</th>
<th>2.5</th>
<th>3.0</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>12.2</td>
<td>11.8</td>
<td>11.0</td>
<td>10.6</td>
<td>9.69</td>
</tr>
<tr>
<td>30</td>
<td>17.8</td>
<td>17.1</td>
<td>15.8</td>
<td>15.0</td>
<td>14.0</td>
</tr>
<tr>
<td>20</td>
<td>29.8</td>
<td>28.0</td>
<td>25.9</td>
<td>24.5</td>
<td>22.6</td>
</tr>
<tr>
<td>10</td>
<td>66.7</td>
<td>63.7</td>
<td>59.3</td>
<td>55.4</td>
<td>51.2</td>
</tr>
</tbody>
</table>

These values are in kOhms!!!
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

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 RXC SPI clock
- pin 12 MOSI SPI master in, slave out
- pin 11 MOSI SPI master out, slave in
- pin 10 SS SPI slave select

The default SPI configuration 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

```c
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
or
Spi.mode((<<GPR0)); // set SPI clock to system clock / 16
```

```c
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
```

```c
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

```c
char spi_transfer(volatile char data) {
    SPDR = data; // Start the transmission
    while (1{(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

<table>
<thead>
<tr>
<th>Bit</th>
<th>SPIE</th>
<th>SPE</th>
<th>DOM</th>
<th>MSTR</th>
<th>CPOL</th>
<th>CPHA</th>
<th>SPIF</th>
<th>SPEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
</tr>
</tbody>
</table>

- **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 S5S 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>
<thead>
<tr>
<th>CPOL</th>
<th>Leading Edge</th>
<th>Trailing Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Rising</td>
<td>Falling</td>
</tr>
<tr>
<td>1</td>
<td>Falling</td>
<td>Rising</td>
</tr>
</tbody>
</table>

• 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>
<thead>
<tr>
<th>CPHA</th>
<th>Leading Edge</th>
<th>Trailing Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sample</td>
<td>Setup</td>
</tr>
<tr>
<td>1</td>
<td>Setup</td>
<td>Sample</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SPI2X</th>
<th>SPR1</th>
<th>SPR0</th>
<th>SCK Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>f_{osc}/4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>f_{osc}/16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>f_{osc}/64</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>f_{osc}/128</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>f_{osc}/2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>f_{osc}/8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>f_{osc}/32</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>f_{osc}/64</td>
</tr>
</tbody>
</table>

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:
SPI Details

**SPSR – SPI Status Register**

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>SPSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7C</td>
<td>SPIF</td>
<td>WCOL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SPI2X</td>
</tr>
<tr>
<td>Read/Write</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>Initial Value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- **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 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 19.5). 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 fosc/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” I2C, 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...

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
I2C – a.k.a. TWI

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

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.

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

<table>
<thead>
<tr>
<th>Capacitance for each I/O Pin</th>
<th>10 pF</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCL Clock Frequency ( f_s^{(1)} )</td>
<td>( 0 ) ( 400 ) kHz</td>
</tr>
<tr>
<td>Value of Pull-up resistor ( R_p )</td>
<td>( f_s \leq 100 ) kHz ( V_{CC} = 0.4 \text{V} ) ( 1000 \mu\text{s} ) ( C_b ) ( \Omega )</td>
</tr>
<tr>
<td>( f_s &gt; 100 ) kHz ( V_{CC} = 0.4 \text{V} ) ( 300 \mu\text{s} ) ( C_b ) ( \Omega )</td>
<td></td>
</tr>
</tbody>
</table>

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

1. Application writes to TWI to initiate transmission of START
2. TWI init set
3. Check TWI to see if START was sent. Application loads SLA+W into TWDR, and loads appropriate control signals into TWCR, making sure that TWINT is written to one, and TWDA is written to zero.
4. TWINT set. Status code indicates SLA+W sent, ACK received
5. Check TWI to see if SLA+W was sent and ACK received. Application loads data into TWDR, and loads appropriate control signals into TWCR, making sure that TWINT is written to one
6. TWINT set. Status code indicates data sent, ACK received
7. Check TWI to see if data was sent and ACK received. Application loads appropriate control signals to send STOP into TWCR, making sure that TWINT is written to one

Luckily Arduino comes with an I2C library!

Wire Library

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 receiving()`
- `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.
```cpp
#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...
```
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 FFFFh, the counter stops counting and all outputs turn off. Pulling BLANK high before the counter reaches FFFFh 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...
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 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.)

The 2k resistors let ~20 mA through each channel.

\[ I = \frac{39.06}{R} \]
\[ e.g. \frac{39.06}{2000} = 0.020 \, \text{A} \]

The 10k pull-up resistor on BLANK turns all outputs off while the Arduino resets.

If using more than one tk, code "NUM_TLCS" to tk, config.h (located in the library folder) and delete Tk5940.o.
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 to set 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

![Graph showing reference resistor vs output current](image)

Min = 5mA
Max = 120mA

\[
I_{max} = \frac{V_{REF}}{R_{REF}} \times 31.5
\]

where:

\[
V_{REF} = 1.24 \text{V}
\]

\[
R_{REF} = \text{User-selected external resistor.}
\]
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...

```c
#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 GCLK 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).
```
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, 0x9f9: 500 microseconds
- 3584, 0xe00: 1500 microseconds
- 3168, 0xc60: 2500 microseconds

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