SPI + I2C

Jonathan Bachrach
EECS UC Berkeley
September 22, 2016
Basic Electronics
- Embedded Communication
Waveforms

Vcc

0V

Vcc

0V

1 0 0 1 1 1 0 1 0 0 1 1 0 0
Asynchronous Serial Ports

- asynchronous
- full duplex
- start, stop, and parity bits
- point to point

start bit

data bits

parity bit

stop bit
What’s Wrong with Serial Ports?

<table>
<thead>
<tr>
<th>feature</th>
<th>disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>two clocks</td>
<td>complexity</td>
</tr>
<tr>
<td>full duplex</td>
<td>overhead</td>
</tr>
<tr>
<td>start, stop, and parity bits</td>
<td>wires</td>
</tr>
<tr>
<td>point to point</td>
<td></td>
</tr>
</tbody>
</table>
3 signals + 1 signal per slave
- single master
- synchronous
- full duplex
- relatively high speed up to few MHz
- no flow control
3 signals + 1 signal per slave
- synchronous
- full duplex
- relatively high speed
- only a few feet
SPI Signals

- Master generated clock SCK
- Master-Out-Slave-In MOSI data
- Master-In-Slave-Out MISO data
- Slave Select SS
- active high/low on
- low Mhz
<table>
<thead>
<tr>
<th>master</th>
<th>slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS low</td>
<td>Temperature Data</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SS high</td>
<td></td>
</tr>
</tbody>
</table>
## Write Register Transaction

<table>
<thead>
<tr>
<th>master</th>
<th>slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS low</td>
<td></td>
</tr>
<tr>
<td>Write Cmd</td>
<td>0</td>
</tr>
<tr>
<td>Register Address</td>
<td>0</td>
</tr>
<tr>
<td>Register Data</td>
<td>0</td>
</tr>
<tr>
<td>SS high</td>
<td></td>
</tr>
<tr>
<td>master</td>
<td>slave</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SS low</td>
<td></td>
</tr>
<tr>
<td>Read Cmd</td>
<td>0</td>
</tr>
<tr>
<td>Register Address</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SS high</td>
<td>Register Data</td>
</tr>
</tbody>
</table>
SPI Waveform

- **CS**: CS signal.
- **Vcc**: Voltage level.
- multiple slave selects
- MOSI, MISO, SCK on bus

```
slave0 ← cs0 ─── master ─── cs1 ─── slave1
       |             |             |
       |             |             |
       |             |             |
       |             |             |
       |             |             |
       |             |             |
       |             |             |
       |             |             |
       |             |             |
       |             |             |
       |             |             |
clk ───────────────────────────
miso
mosi
```
Daisy Chained Slaves

- shared slave select
- shared SCK
- Daisy-chaned MOSI and MISO
- keep pumping data down chain
- can bit bang gpio or
- use few dedicated lines for the purpose
Initialization

- bit order – endianess
- data mode – for when to sample data on clock edge
- clock divider – for speed of clock
use logic analyzer like Saleae
Tips and Tricks

for longer distance travel use

- slower clock
- special driver chips
SPI(PinName mosi, PinName miso, PinName sclk, PinName ssel=NC)
void format(int bits, int mode=0)
void frequency(int hz=1000000)
virtual int write(int value)
virtual void lock(void)
virtual void unlock(void)
#include "mbed.h"

SPI spi(p5, p6, p7); // mosi, miso, sclk
DigitalOut cs(p8);

int main() {
    cs = 1; // Chip must be deselected
    // Setup the spi for 8 bit data, high steady state clock,
    // second edge capture, with a 1MHz clock rate
    spi.format(8,3);
    spi.frequency(1000000);
    cs = 0; // Select the device by setting chip select low
    // Send 0x8f, the command to read the WHOAMI register
    spi.write(0x8F);
    // Send a dummy byte to receive the contents of the WHOAMI register
    int whoami = spi.write(0x00);
    printf("WHOAMI register = 0x%X\n", whoami);
    cs = 1; // Deselect the device
}
asynchronous SPI transfers
- can set up more work to be done
- callback on completion
- not implemented on all platforms
- will look into this more
What’s Wrong with SPI?

- high speed
- 3 pins + pin per slave
- fast and full-duplex
- only two pins
- half-duplex
- shareable bus
- medium speed
- ack/nack per 8 bits
- SCL for clock and SDA for data
- pull up resistors 4.7K for active pull down bus
- open drain – pull down but not up
- up to 2-3m lengths
I2C Packets

- Start
- Address + R/W
- Data
- Stop
1->0 transition on SDA with SCL high
First master wins
Possible to do repeated starts
Address Frame

- 7-bit or 10-bit address
- R/W bit
- NACK/ACK bit from slave or not
Data Frames

- clock out data frames
- from master or slave depending on R/W
End of data frames

0->1 transition on SDA after 1->0 transition on SCL

During normal data avoid stop condition
Repeated Start Conditions

- No end condition after data frame
- Master maintains bus and continues transfer
- Any number of repeated starts is allowed
- Ends with stop condition
Clock Stretching

- Sometimes clock rate is too fast for slave
- During Ack/Nack period slave can hold the SCL line low after master releases it
- Master is required to not clock until slave releases it
- Limits to how long you can clock stretch
<table>
<thead>
<tr>
<th>master</th>
<th>slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td></td>
</tr>
<tr>
<td>Device Address + WR</td>
<td>Ack</td>
</tr>
<tr>
<td>Write Cmd</td>
<td>Ack</td>
</tr>
<tr>
<td>Register Address</td>
<td>Ack</td>
</tr>
<tr>
<td>Register Data</td>
<td>Ack</td>
</tr>
<tr>
<td>Stop</td>
<td></td>
</tr>
</tbody>
</table>
# Read Transaction

<table>
<thead>
<tr>
<th>master</th>
<th>slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Ack</td>
</tr>
<tr>
<td>Device Address + WR</td>
<td>Ack</td>
</tr>
<tr>
<td>Read Cmd</td>
<td>Ack</td>
</tr>
<tr>
<td>Register Address</td>
<td>Ack</td>
</tr>
<tr>
<td>Start</td>
<td>Ack</td>
</tr>
<tr>
<td>Device Address + RD</td>
<td>Ack</td>
</tr>
<tr>
<td>Register Data</td>
<td>NAck</td>
</tr>
<tr>
<td>Stop</td>
<td></td>
</tr>
</tbody>
</table>
I2C(PinName sda, PinName scl)
void frequency(int hz)
int read(int address, char *data, int length, bool repeated=false)
int read(int ack)
int write(int address, const char *data, int length, bool repeated=false)
int write(int data)
void start(void)
void stop(void)
#include "mbed.h"

I2C i2c(p28, p27);
const int addr = 72;

int main() {
    char cmd;
    i2c.frequency(9600);
    while (1) {
        cmd = 0;
        i2c.write(addr, cmd, 1);
        wait(0.5);
        i2c.read(addr, cmd, 1);
        float tmp = (float)(cmd[0]) / 256.0;
        printf("Temp = %.2f\n", tmp);
    }
}

from mbed website
asynchronous I2C transfers
- can set up more work to be done
- callback on completion
- not implemented on all platforms
- will look into this more
JITPCB Circuit Design