Sailing the I2Cs

with the Bus Pirate

Jared Boone

DorkbotPDX 0x08 - January 30, 2012

Tuesday, February 26, 13

Hi, I'm Jared Boone. I'm going to speak about the I2C bus, and a cool gadget, called the Bus Pirate, that makes working with the I2C bus a lot easier.

“I2C”?

- “Inter-Integrated Circuit” bus.
- Truth in naming: provides a way for integrated circuits (chips) to communicate.
- Very simple. So simple it costs virtually *nothing* to implement inside a chip.
- PCI, USB, even PC-style serial ports are more complicated and expensive to put in a chip. And they’re all overkill for chip-chip communication.

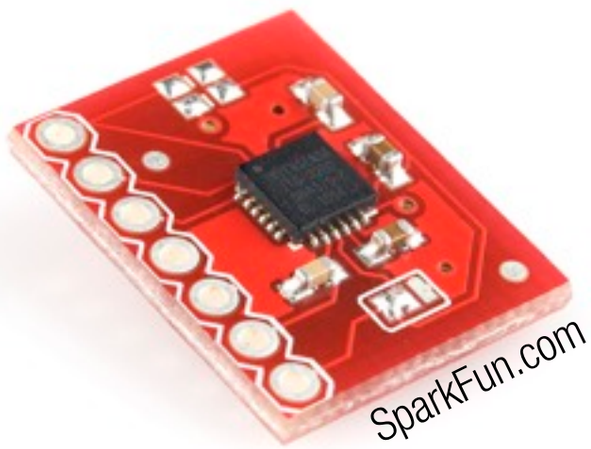
So what is I2C? It’s a communication bus designed specifically for chips to communicate with each other, very inexpensively. The best way to make something inexpensive is to make it simple. And I2C is definitely simple. It’s *almost* the simplest form of digital communication conceivable. Way simpler than PCI or PCI Express. Way simpler than USB. Simpler, even, than a PC serial port.

What Speaks I2C?

- Many integrated circuits -- sensors, memories, chips that need to be configured/controlled externally.
- Most microcontrollers speak I2C, too. The Arduino (AVR8), ARMs, MSP430s, PICs, you name it!

OK, so it's simple. But what is it good for? A lot of fun and interesting chips implement I2C, as do virtually all the microcontroller chips available. Here are some examples:

Sensors



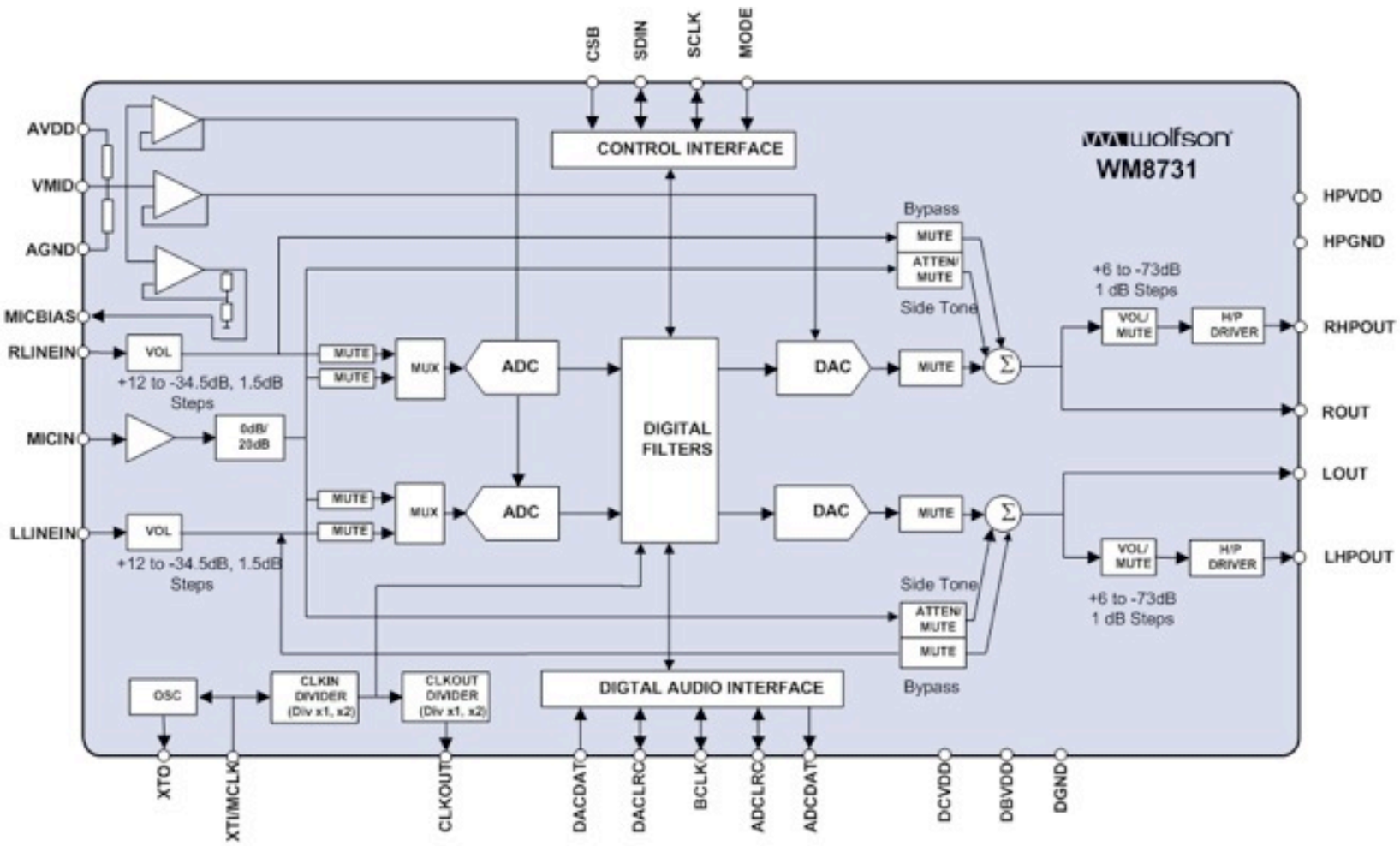
Analog Devices ADXL345 three-axis accelerometer

Honeywell HMC5843 three-axis magnetometer (compass)

Invensense ITG-3200 three-axis gyroscope

Many others available -- temperature, humidity, pressure, light/color...

Stereo Audio Codec

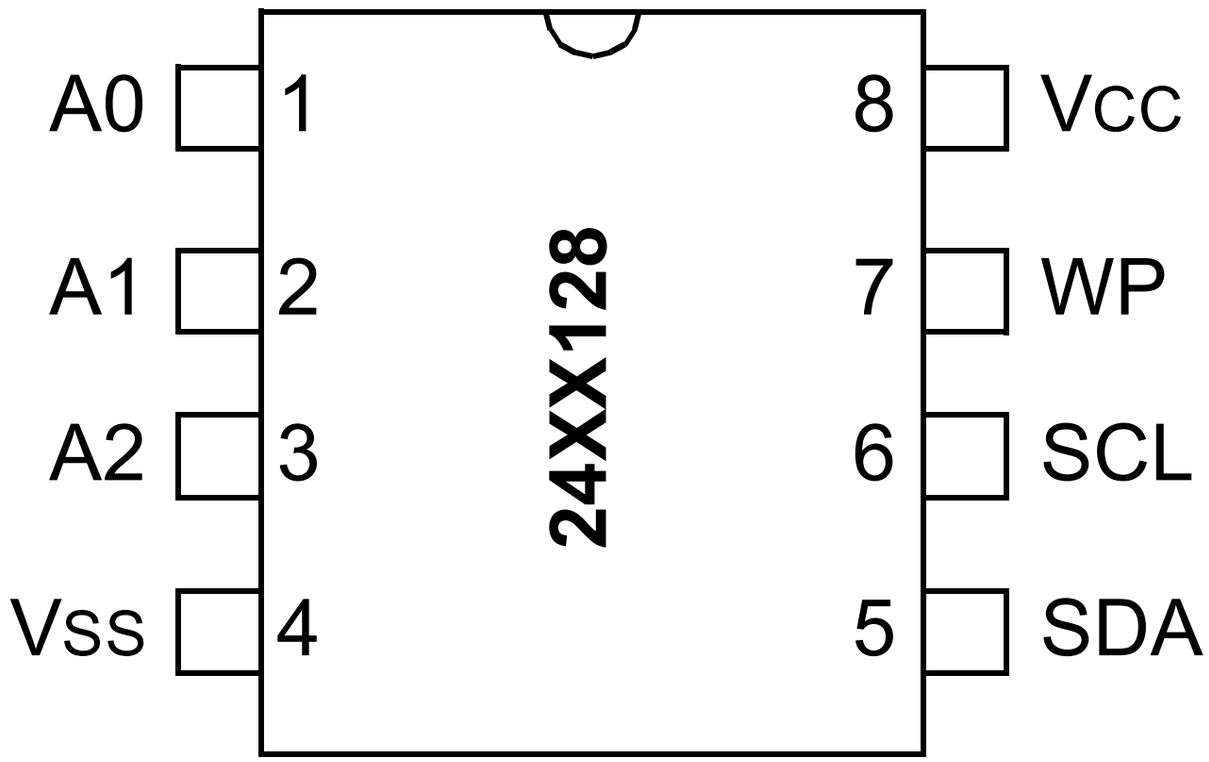


Tuesday, February 26, 13

Audio codecs, for high-quality audio.

I2C is not used for the audio signal, but is used for configuring signal routing, input and output volumes, etc.

Electrically-Erasable ROM



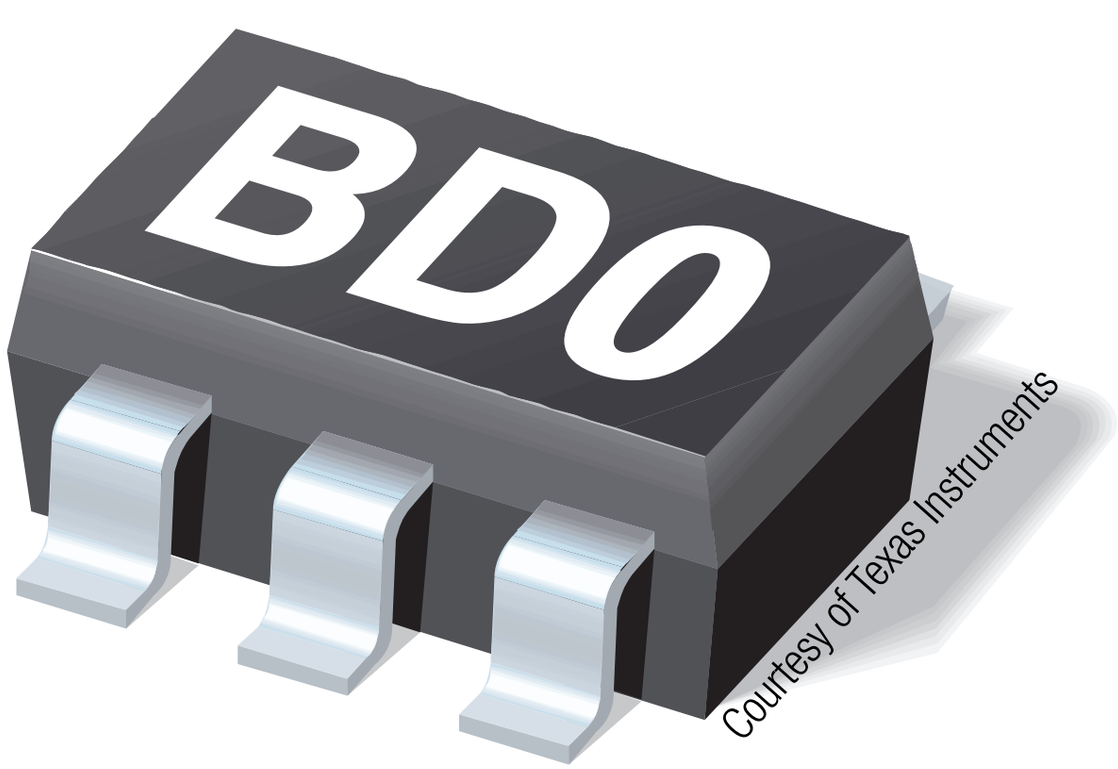
Courtesy of Microchip

Tuesday, February 26, 13

memory:

Serial EEPROMs and flash for persisting more data than your microcontroller allows.

Analog to Digital Converter



Tuesday, February 26, 13

Data converters:

Low-speed analog-to-digital and digital-to-analog converters for capturing or generating analog signals (if your microcontroller doesn’t have enough converters, or they’re not very good).

I2C-controlled RGB LED

SparkFun.com

Tuesday, February 26, 13

LEDs like the BlinkM:

For generating a complex, multi-color, multi-LED light show from your microcontroller.

How Simple Is It?

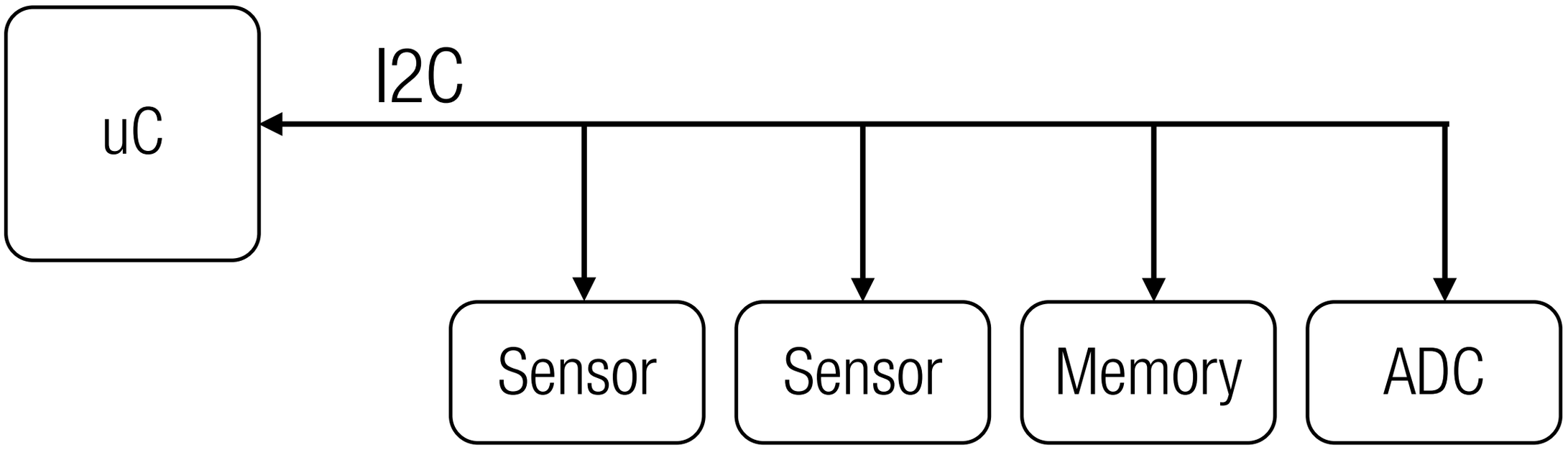
- Serial bus -- transfers data one bit at a time.
- No fancy self-clocking scheme (like USB or even PC-style serial ports). Clock signal is a separate signal from the data.
- Only two signals/wires to connect chips to each other. (If they're on the same board, that is...)

Tuesday, February 26, 13

So I2C is a simple serial bus. It transmits data one bit at a time.

What makes it simpler than other serial buses, like USB or a PC serial port? USB and serial ports use tricky encodings and hardware so the receiving device's hardware can synchronize with the transmitted data. That extra hardware is expensive to put in a chip, so instead, the I2C designers added a separate synchronization (or "clock") signal. This makes the hardware in the chip a lot simpler.

Shared Bus



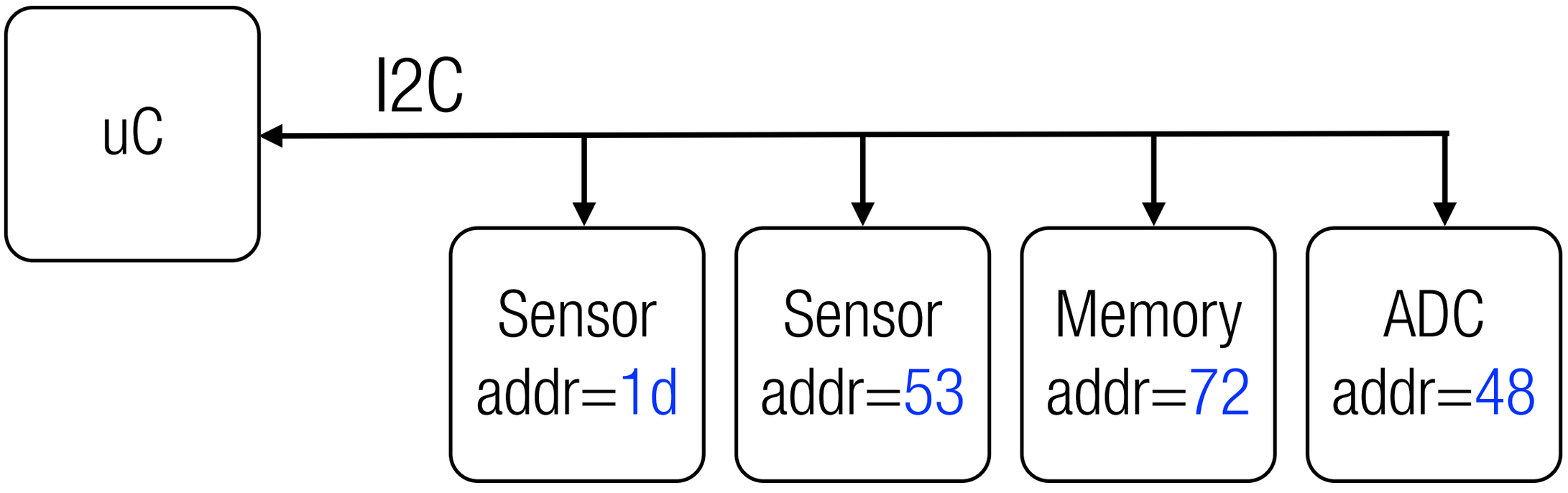
Tuesday, February 26, 13

I2C is a shared bus. You can connect lots of devices to the same bus, and communicate with them individually. With only two signals shared between all the devices, how do the devices on the bus avoid speaking at the same time?

To start with, I2C uses a master/slave model. The master (usually the microcontroller) starts ALL conversations. The rest of the devices on the bus are slaves, and only respond to the master's requests.

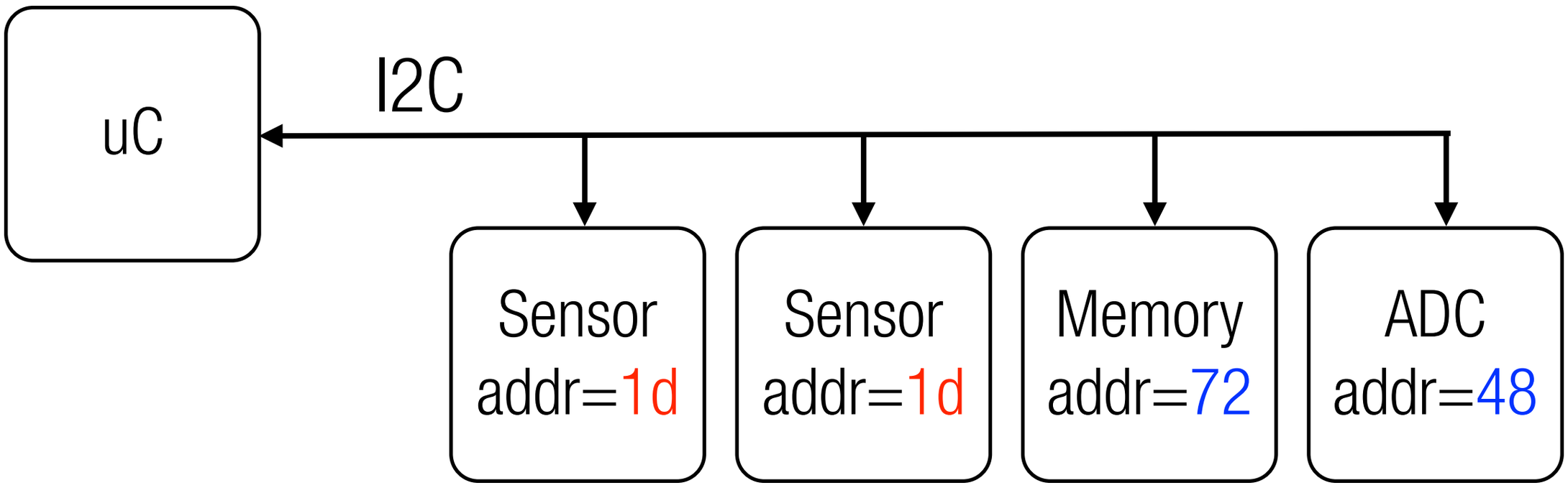
OK, so the master is in charge of the bus, initiating all requests. But, at the start of a request, how does the master indicate which device it wants to speak with? At the beginning of each request, the master sends an address over the bus. Each device has a unique address, and only the device which matches that address will respond.

Unique Addresses



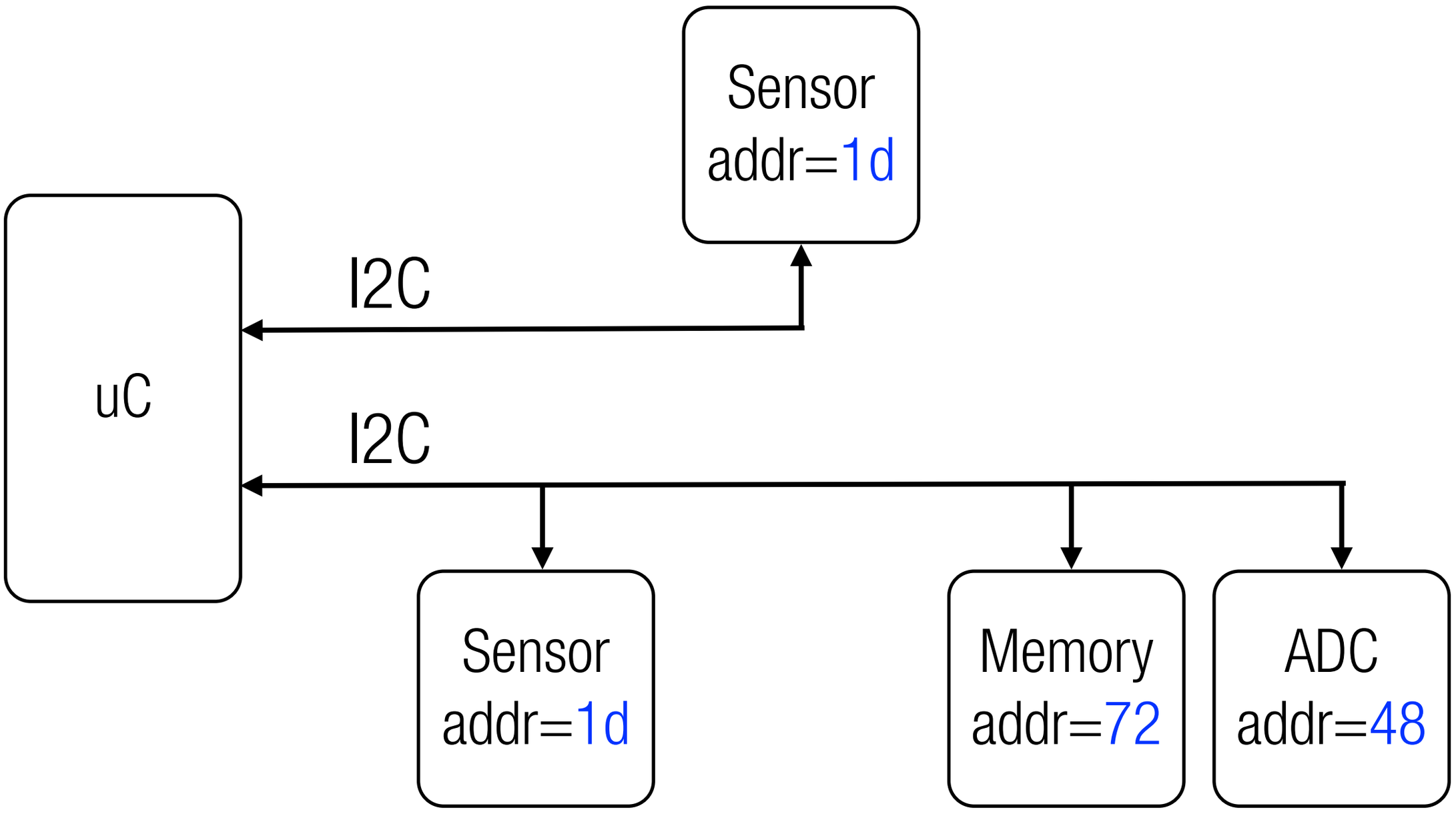
I2C addresses are seven bits. So you can only have 128 devices on the bus. In practice, that’s not a big deal. Most I2C designs I’ve seen have no more than a half-dozen devices on the bus.

Address Conflict



The problem is with how chips get their addresses. Remember, I2C is *cheap*. Most chips with I2C interfaces have a hard-coded address. For instance, the Analog Devices ADXL345 accelerometer chip has an address of 1D hexadecimal. ALL ADXL345s have that same address. So if you have two of those chips on the same I2C bus, they'll both respond when you try to talk to address 1D, and things won't work right.

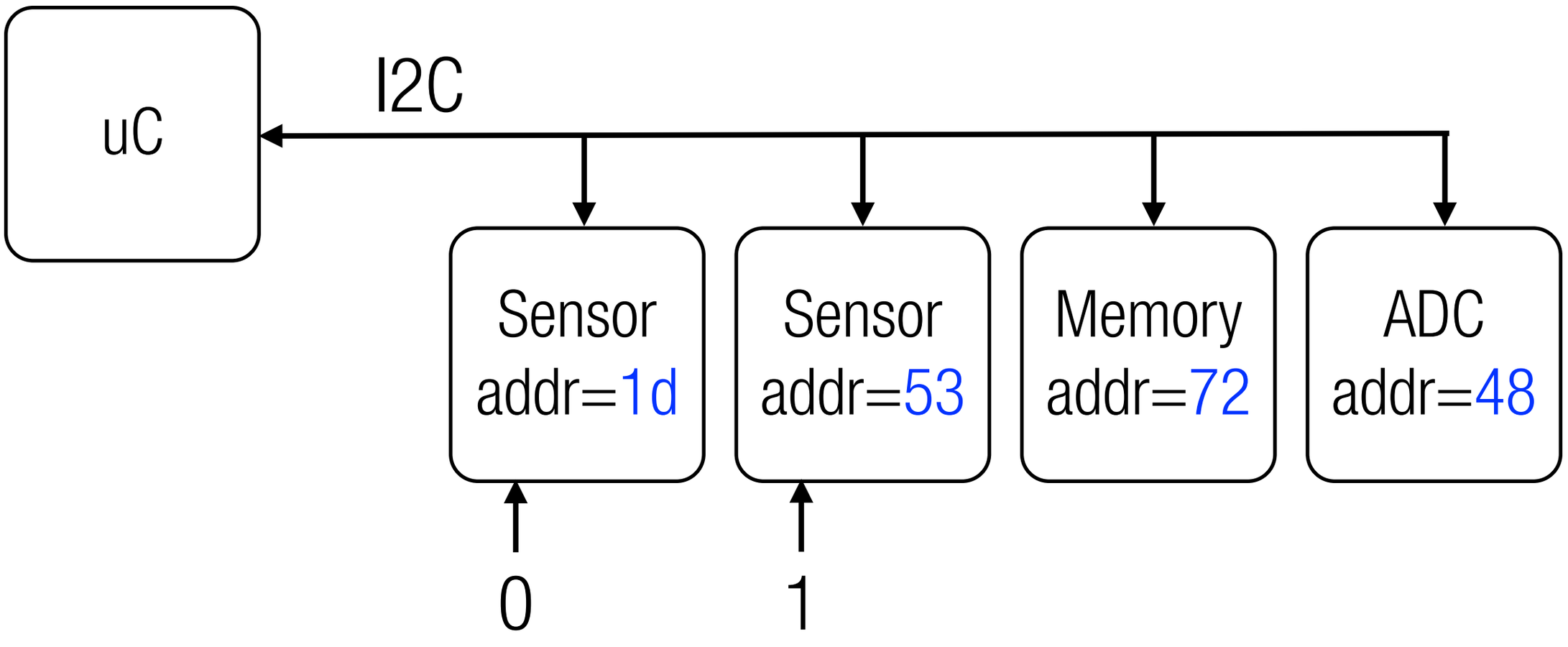
Separate Buses



Tuesday, February 26, 13

One thing you can do is move one of the conflicting devices to a different I2C bus, if your microcontroller has another I2C bus.

Device Reconfiguration



Or, some I2C chips have configuration pins that allow you to set the address to one of several choices. In this case, we can configure each of the ADXL345 sensor chips to operate on one of two addresses (1d hex, or 53 hex). Of course, if we had THREE ADXL345s on the same bus, we'd once again have a conflict, since we can choose only one of TWO addresses for those chips.

Transactions

A request from the master, to a slave device:

- Master sends slave device address.
- Master sends direction of transfer -- “read” or “write”.
- Slave (if any) responds to “go ahead”.
- Master or slave transmits data (depending on direction specified above).
- Master ends transaction.

Tuesday, February 26, 13

I2C communication happens in units called “transactions”, which involve transferring data between the master and one of the slaves.

Each transaction starts with the address of the device the master wants to talk to, and the direction of the data transfer (“read” or “write”). Once the address and direction is sent, the addressed device will send an acknowledgement. If no device recognizes the address, no acknowledgement is sent.

If a slave responded, the master and slave transfer data.

Single-Byte Transmit

Master

Slaves

A transaction is starting!

(We're listening...)

I am transmitting to device 72!

Device 72: OK, ready!
Other devices: (crickets)

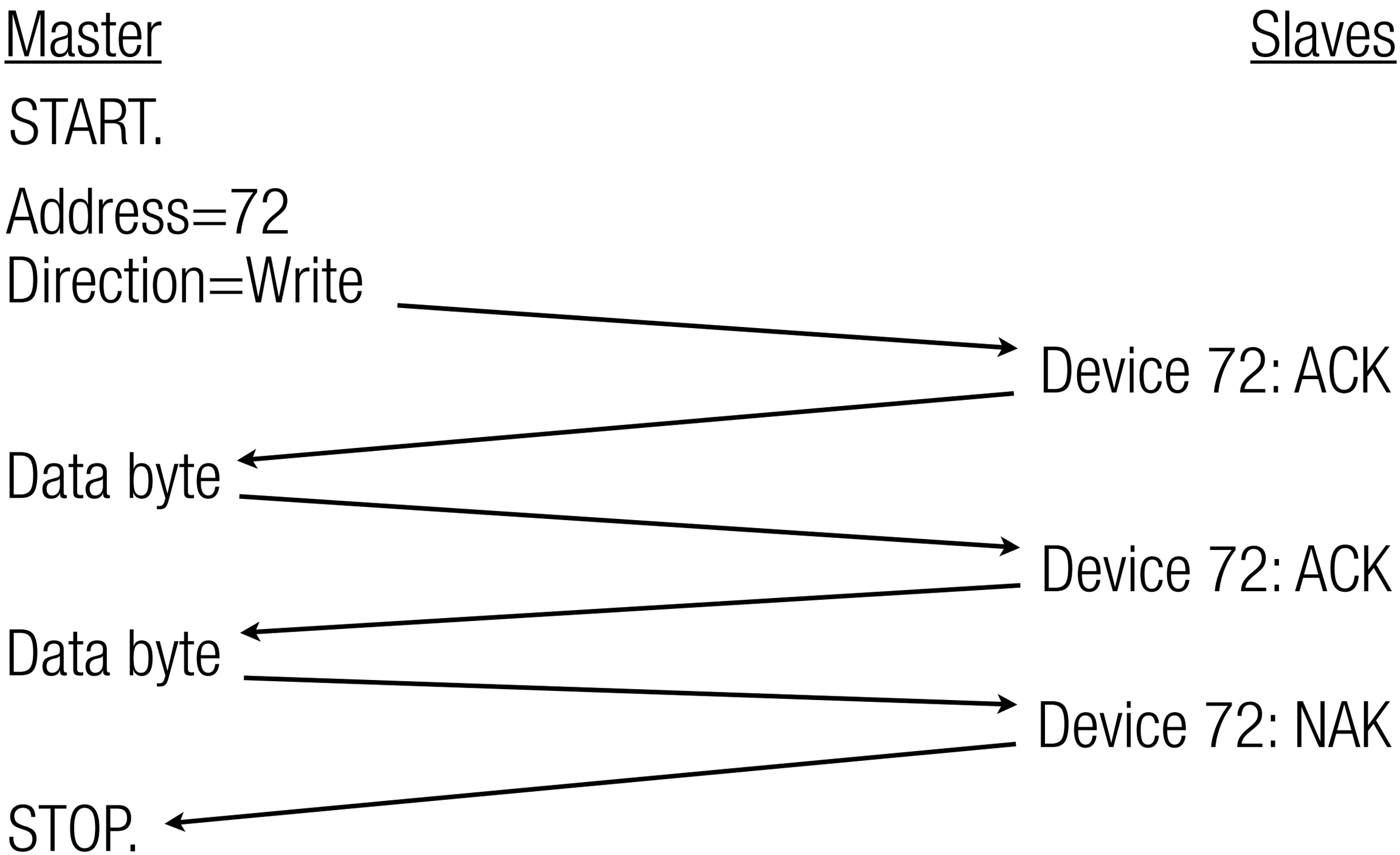
Here is a byte of data!

Device 72: OK, ready for more!

The transaction is done!

Here's a more diagrammatical example where the master is sending one byte of data to slave with address 72.

Multi-Byte Transmit

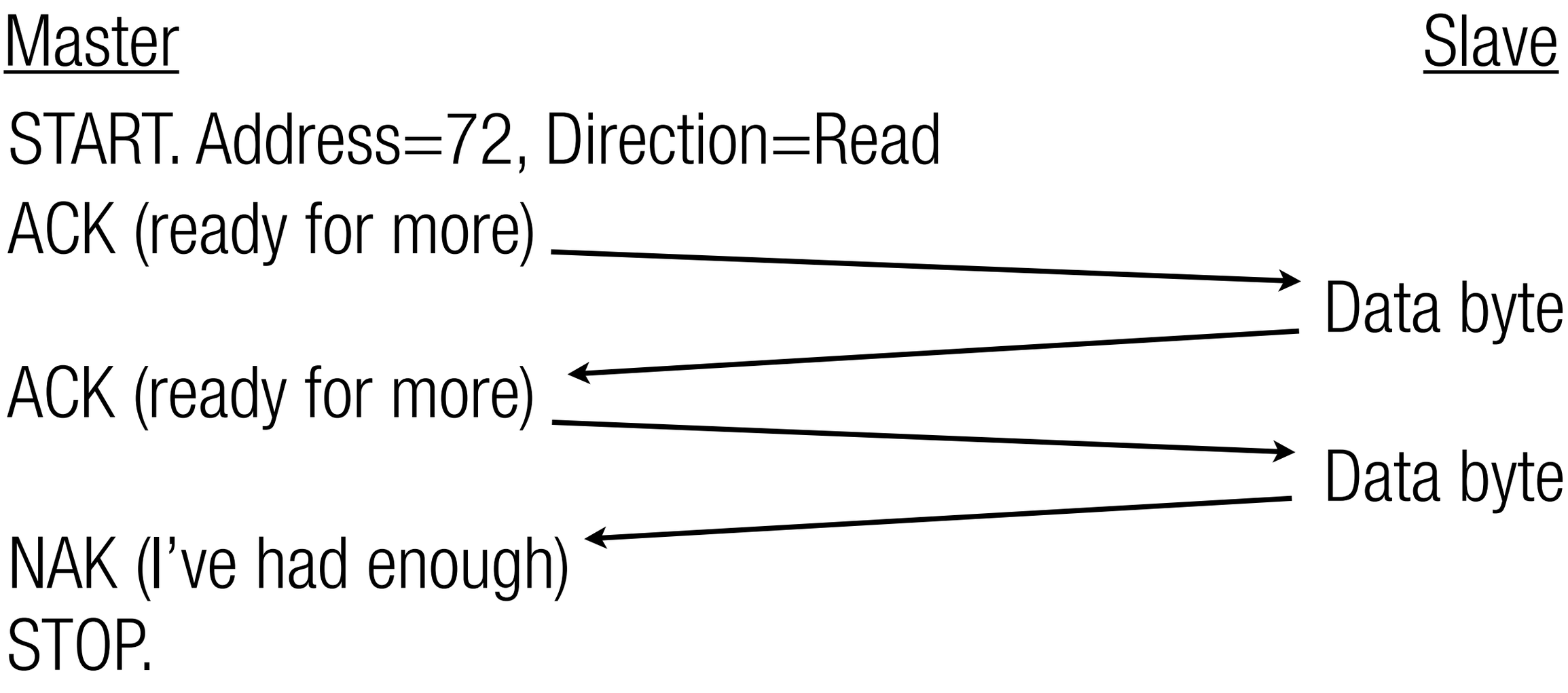


Tuesday, February 26, 13

Here’s another transaction, this time with better I2C terminology:

The beginning and end of a transaction are indicated by START and STOP bits sent by the master.
If the slave can’t accept the data for whatever reason, it sends back a “NAK” (negative acknowledgement), and the master will end the transaction.

Multi-Byte Receive

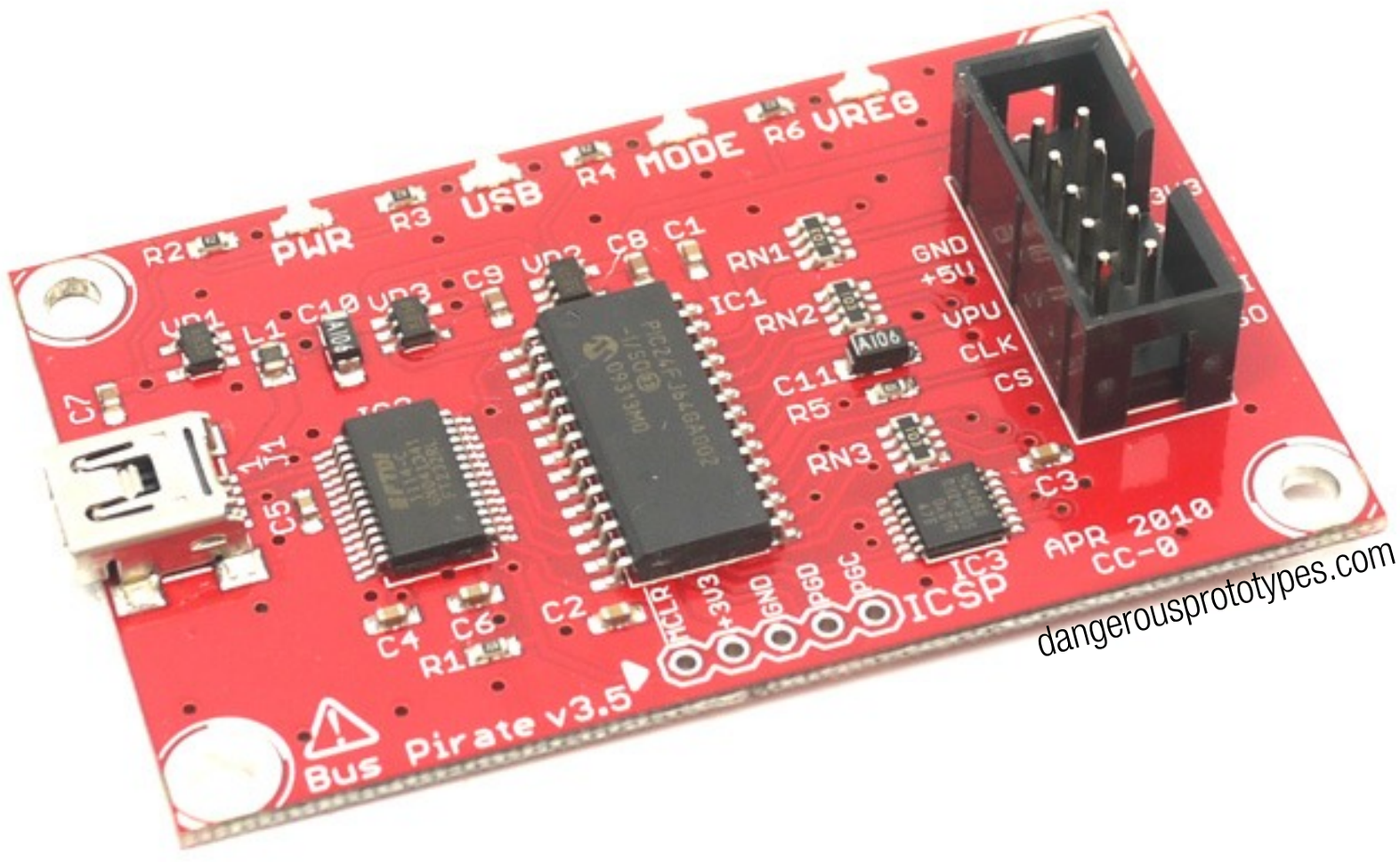


Tuesday, February 26, 13

Here, we send data in the other direction, from the slave to the master.

Pretty much the same stuff, but in the opposite direction... START bit, address, direction, acknowledgement after every byte, and the STOP bit.

Bus Pirate



Dangerous Prototypes, available from SeeedStudio

Tuesday, February 26, 13

Talking with I2C devices using a microcontroller can be painful -- keeping track of START and STOP bits and acknowledgements... The Bus Pirate is the easiest way I've seen to get to know a new I2C device. You don't have to write any code to play with a new I2C device, and you can be sure the Bus Pirate is doing I2C right. So all you have to do is understand the chip you're communicating with.

Dangerous Prototypes designed the Bus Pirate, and they sell it through SeeedStudio (yes, that's THREE "e"s.) It interfaces directly with computer via USB, shows up as a serial port. Via the serial port, you choose options from a menu and issue commands to transmit over I2C. It can supply power to target device (either 3.3 or 5 volts), so you really don't need anything else to play with an I2C chip.

It's good for a lot more than I2C, too...

Bus Pirate “Protocols”

- I2C, SPI, raw two- and three-wire serial
- Dallas/Maxim 1-WIRE
- UART (e.g. PC serial port)
- HD44780 LCD (requires adapter board)
- Supports AVR ISP programming via SPI w/AVRDude.
- PWM (servo motors and other stuff!)
- ADC voltage sampling (very slow oscilloscope)
- Alternate firmwares do a bunch of other cool stuff.

Tuesday, February 26, 13

Here’s some of the other stuff you can do with a Bus Pirate. I have yet to try most of this stuff...

Bus Pirate Target Interface

Pin Name	Description (Bus Pirate is the master)
MOSI	Master data out, slave in (SPI, JTAG), Serial data (1-Wire, I2C, KB), TX* (UART)
CLK	Clock signal (I2C, SPI, JTAG, KB)
MISO	Master data in, slave out (SPI, JTAG) RX (UART)
CS*	Chip select (SPI), TMS (JTAG)
AUX	Auxiliary IO, frequency probe, pulse-width modulator
ADC	Voltage measurement probe (max 6volts)
Vpu	Voltage input for on-board pull-up resistors (0-5volts).
+3.3v	+3.3volt switchable power supply
+5.0v	+5volt switchable power supply
GND	Ground, connect to ground of test circuit

dangerousprototypes.com

Here's the signals available from the Bus Pirate. I've outlined the I2C-specific signals. You can see the other buses and signals it supports, and imagine how to hook it up to a JTAG or SPI or UART device.

Useful Accessories

- 0.1” stake pin jumpers! Short bits of wire with female pin connectors on both ends.
- The SeeedStudio Bus Pirate Probe Kit is not so great. The clips are quite uncooperative...
- Reliable USB cable. (Long story...)

Tuesday, February 26, 13

A few recommendations if you’re going to buy a Bus Pirate. Definitely get the stake pin jumpers. I use them all the time, and they’re great. And they’re good for a lot more than Bus Pirate stuff.

I’d avoid the probe kit. The one I got turned to junk in about a day. The clips won’t clip, or fall apart.

Bus Pirate “Open”

- Mac OS Terminal
  screen /dev/tty.usbserial-<something> 115200

- Linux:
  screen /dev/tty<something> 115200

- Windows: I have no idea. I hear HyperTerminal is no longer included as of Vista...

How do you communicate with the Bus Pirate? Just like any other serial port, though by default, it runs at 115 kilobaud.

Bus Pirate “Login”

- Press ENTER/RETURN for a prompt.
- Default mode is “HiZ”. Prompt indicates bus mode.
- “?” command gets you the menu.

Tuesday, February 26, 13

Once you’ve connected to the Bus Pirate via the serial port, you need to get its attention. Press ENTER/RETURN, and you’ll be greeted with a “HiZ” prompt. Type “?” to get the main menu.

Bus Pirate Menu

```
General                                                 Protocol interaction
---------------------------------------------------------------------------
?       This help                               (0)     List current macros
=X/|X   Converts X/reverse X                    (x)     Macro x
~       Selftest                                [       Start
#       Reset the BP                            ]       Stop
$       Jump to bootloader                      {       Start with read
&/%     Delay 1 us/ms                           }       Stop
a/A/@   AUXPIN (low/HI/READ)                    "abc"   Send string
b       Set baudrate                            123
c/C     AUX assignment (aux/CS)                 0x123
d/D     Measure ADC (once/CONT.)                0b110   Send value
f       Measure frequency                       r       Read
g/S     Generate PWM/Servo                      /       CLK hi
h       Commandhistory                          \       CLK lo
i       Versioninfo/statusinfo                  ^       CLK tick
l/L     Bitorder (msb/LSB)                      -       DAT hi
m       Change mode                             _       DAT lo
o       Set output type                         .       DAT read
p/P     Pullup resistors (off/ON)               !       Bit read
s       Script engine                           :       Repeat e.g. r:10
v       Show volts/states                       ;       Bits to read/write e.g. 0x55;2
w/W     PSU (off/ON)                <x>/<x= >/<0>       Usermacro x/assign x/list all
HiZ>
```

Tuesday, February 26, 13

Here’s the main menu. There’s lots of options I haven’t used yet -- the ability to bit-bang signals, sample analog voltages, control servos, etc. etc.

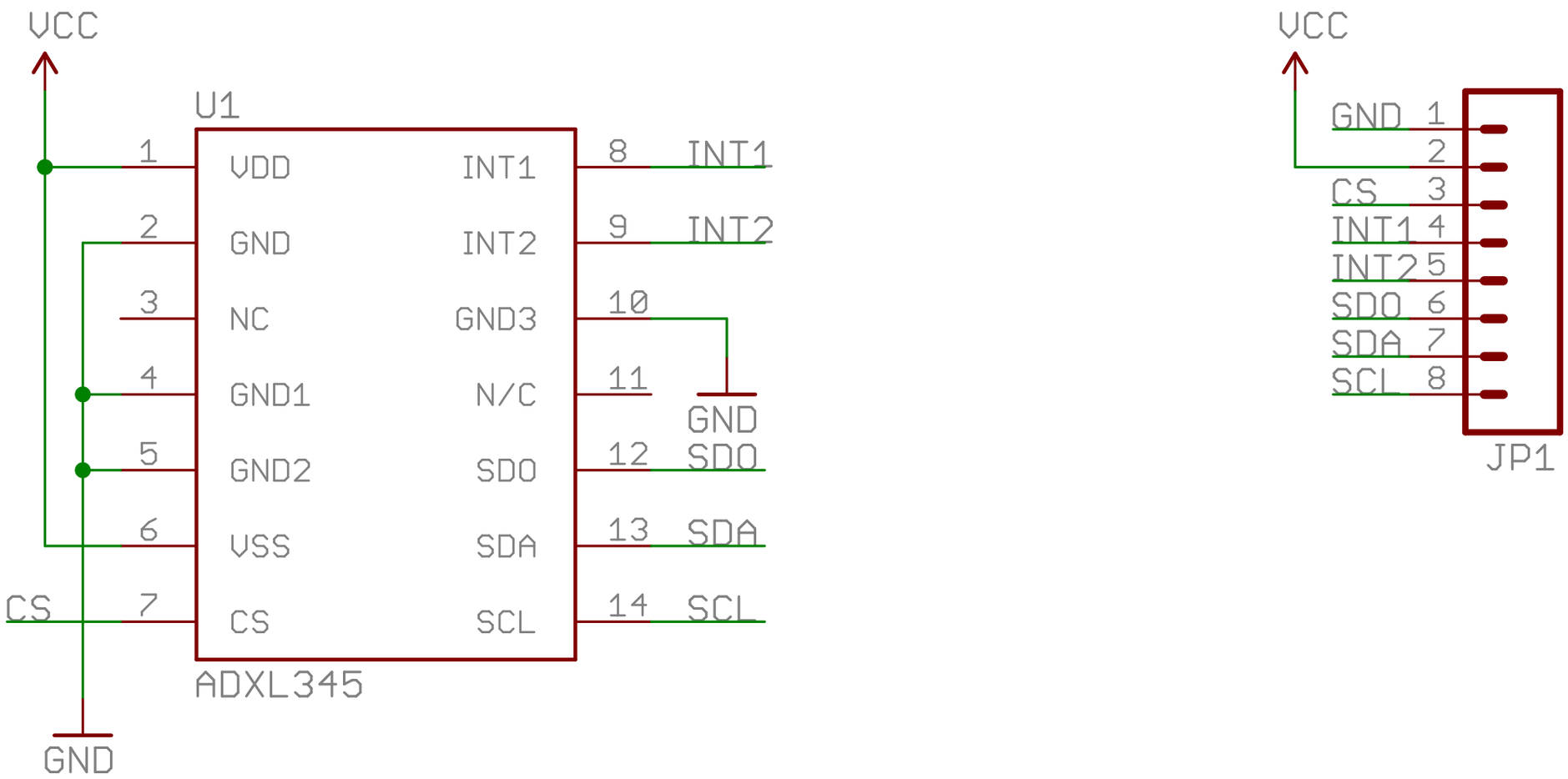
Demo!

Three-Axis Accelerometer

Tuesday, February 26, 13

Demo time.
I had this breakout board laying around from a car racing project I built a couple of years ago. It’s an Analog Devices ADXL345 breakout board from SparkFun. Who doesn’t love accelerometers?

Board Schematic



Here's the breakout board schematic. The I2C signals are SDA (data) and SCL (clock). We also need to hook up power. Reading the datasheet, I see the chip can accept supply voltage up to 3.6 volts, so we'll use the Bus Pirate's 3.3 volt supply.

Also in the datasheet: This chip will operate on two different kinds of buses -- I2C and SPI. To select I2C, the CS pin on the chip must be at the supply voltage when the chip is turned on. The Bus Pirate has a couple of spare signals, so I'll use one of those to control the chip's CS pin.

Bus Pirate Connections

Bus Pirate	Target Board
GND	GND
CLK (I2C:SCL)	SCL
MOSI (I2C:SDA)	SDA
CS	CS (for I2C bus mode)
+3V3	VCC (2.0 to 3.6V only!)
AUX looped back to VPU	

Tuesday, February 26, 13

Here's how I connected the Bus Pirate to the board.

I2C requires you to connect two resistors to make the bus work. One goes between SCL and the power supply, and the other between SDA and the power supply. It's too long of a story to get into right now. But regardless, the Bus Pirate provides controllable pull-up resistors, so you don't need to wire them up yourself if your I2C circuit doesn't already have them. You do need to supply power to the pull-up resistors (via the Bus Pirate's VPU pin), so I played a little trick, using one of the Bus Pirate's extra signals (AUX) to drive power into the VPU pin. (Apparently, they're eliminating the need for this trick in the forthcoming Bus Pirate v4.)

Enter I2C Mode

```
HiZ>m
1. HiZ
2. 1-WIRE
3. UART
4. I2C
5. SPI
6. 2WIRE
7. 3WIRE
8. LCD
x. exit(without change)

(1)>4
Set speed:
 1. ~5KHz
 2. ~50KHz
 3. ~100KHz
 4. ~400KHz

(1)>1
Ready
I2C>
```


Slow is good when you’re starting out with a new chip. I usually start with 5KHz -- still way faster than I can type. At 5KHz, I don’t have to worry about how bad or ugly my wiring is, or how fast the chip can communicate via I2C -- it should just work.

Turn On The Things!

```
I2C>c
a/A/@ controls AUX pin
I2C>A
AUX HIGH
I2C>C
a/A/@ controls CS pin
I2C>A
AUX HIGH
I2C>P
Pull-up resistors ON
I2C>W
POWER SUPPLIES ON
I2C>
```


Here, I'm setting up the I2C pull-up resistor power, setting the chip's bus mode, and turning on the power to the chip.

Turn on the AUX pin, to power the I2C pull-up resistors.
Turn on the CS pin, to set the chip's bus mode.
Enable the Bus Pirate's I2C pull-up resistors.
Turn on the Bus Pirate's power supplies.

The chip now has power, is in I2C mode, and should respond to I2C commands.

Address Search

```
I2C>(1)
Searching I2C address space. Found devices at:
0xA6(0x53 W) 0xA7(0x53 R)

I2C>
```

Reveals chip responding at:
0xA6 (address 0x53 + write)
0xA7 (address 0x53 + read)

The Bus Pirate can scan the I2C bus, looking for slave chips. It'll then print out what it found. A chip's address is usually in the data sheet, but it can be hard to find. Letting the Bus Pirate find your chip's address is a whole lot easier...

The Bus Pirate printed out the full eight-bit address+direction values for reading (0xa7) and writing (0xa6), and also shows the address value by itself (0x53).

Now What?

REGISTER MAP

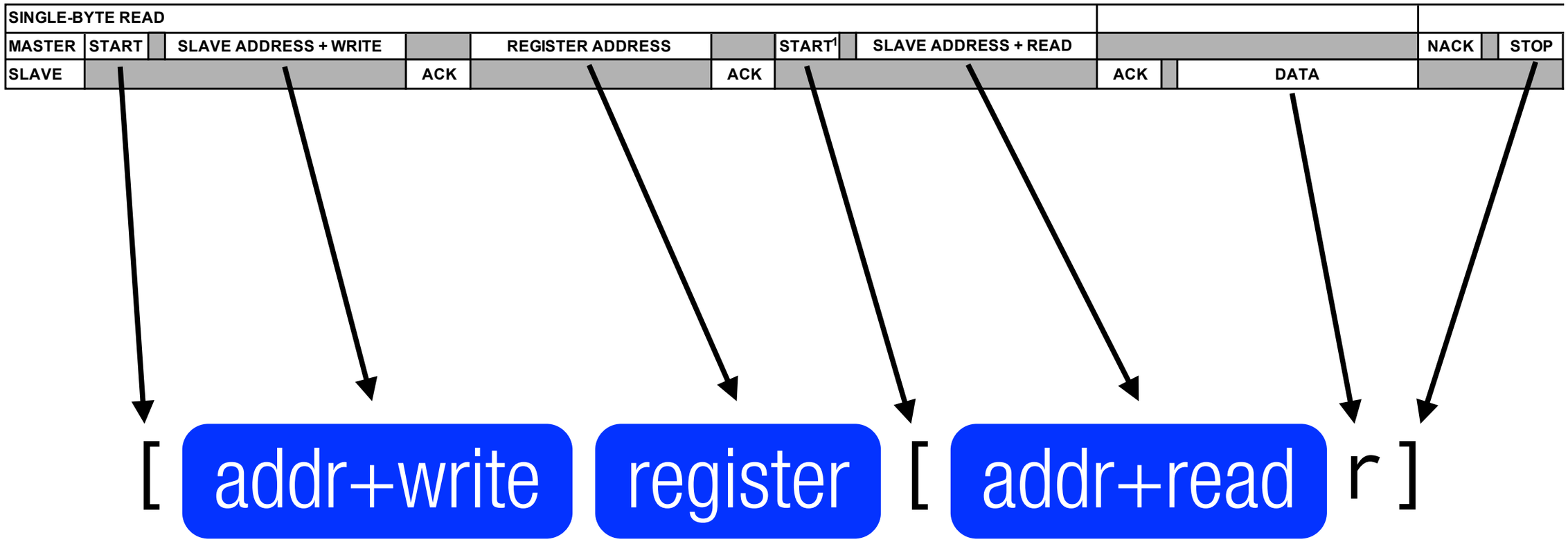
Table 19.

Address					
Hex	**Dec**	**Name**	**Type**	**Reset Value**	**Description**
0x00	0	DEVID	R	11100101	Device ID
0x01 to 0x1C	1 to 28	Reserved			Reserved; do not access
0x1D	29	THRESH_TAP	R/$\overline{W}$	00000000	Tap threshold
0x1E	30	OFSX	R/$\overline{W}$	00000000	X-axis offset
0x1F	31	OFSY	R/$\overline{W}$	00000000	Y-axis offset
0x20	32	OFSZ	R/$\overline{W}$	00000000	Z-axis offset
0x21	33	DUR	R/$\overline{W}$	00000000	Tap duration
0x22	34	Latent	R/$\overline{W}$	00000000	Tap latency
0x23	35	Window	R/$\overline{W}$	00000000	Tap window
0x24	36	THRESH_ACT	R/$\overline{W}$	00000000	Activity threshold
0x25	37	THRESH_INACT	R/$\overline{W}$	00000000	Inactivity threshold
0x26	38	TIME_INACT	R/$\overline{W}$	00000000	Inactivity time
0x27	39	ACT_INACT_CTL	R/$\overline{W}$	00000000	Axis enable control for activity and inactivity detection
0x28	40	THRESH_FF	R/$\overline{W}$	00000000	Free-fall threshold
0x29	41	TIME_FF	R/$\overline{W}$	00000000	Free-fall time
0x2A	42	TAP_AXES	R/$\overline{W}$	00000000	Axis control for single tap/double tap
0x2B	43	ACT_TAP_STATUS	R	00000000	Source of single tap/double tap
0x2C	44	BW_RATE	R/$\overline{W}$	00001010	Data rate and power mode control
0x2D	45	POWER_CTL	R/$\overline{W}$	00000000	Power-saving features control
0x2E	46	INT_ENABLE	R/$\overline{W}$	00000000	Interrupt enable control
0x2F	47	INT_MAP	R/$\overline{W}$	00000000	Interrupt mapping control
0x30	48	INT_SOURCE	R	00000010	Source of interrupts
0x31	49	DATA_FORMAT	R/$\overline{W}$	00000000	Data format control
0x32	50	DATAX0	R	00000000	X-Axis Data 0
0x33	51	DATAX1	R	00000000	X-Axis Data 1
0x34	52	DATAY0	R	00000000	Y-Axis Data 0
0x35	53	DATAY1	R	00000000	Y-Axis Data 1
0x36	54	DATAZ0	R	00000000	Z-Axis Data 0
0x37	55	DATAZ1	R	00000000	Z-Axis Data 1
0x38	56	FIFO_CTL	R/$\overline{W}$	00000000	FIFO control
0x39	57	FIFO_STATUS	R	00000000	FIFO status

Tuesday, February 26, 13

The accelerometer chip has a bunch of data registers that we can read and write, all described in the datasheet. Some registers allow us to configure how fast the accelerometer measures, others allow us to detect taps and change the format of the accelerometer data.

Single-Register Read



Read device ID: `[0xa6 0x00[0xa7 r]`

From the datasheet, we see how to send a single-register read to the accelerometer.
The diagram on the top is from the datasheet, and below is the Bus Pirate command we use to perform this transaction.
First, we send a START bit, then the slave address and write bit.
Then we send the desired register address.
Then we send another START bit. (What, START again? Long story short, this is called a “repeated START”, which allows us to change the direction of the conversation without having to start a new transaction.)
Then, the same slave address, but with the read bit instead of the write bit.
Then we read a byte from the slave (the acceleromter).
Lastly, we send a STOP bit.

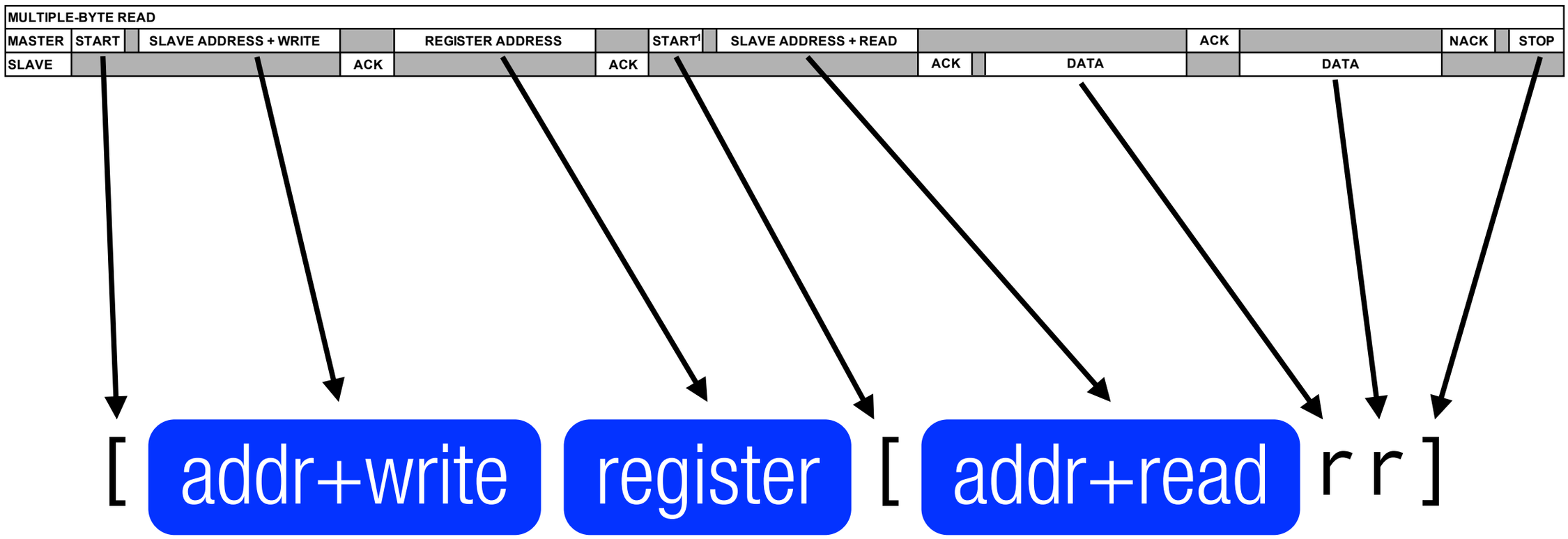
Read Device ID?

```
I2C>[0xa6 0x00[0xa7 r]
I2C START BIT
WRITE: 0xA6 ACK
WRITE: 0x00 ACK
I2C START BIT
WRITE: 0xA7 ACK
READ: 0xE5
NACK
I2C STOP BIT
I2C>
```


And here's what it looks like, through the Bus Pirate.

Cool, 0xE5 matches the device ID in the datasheet. We're getting the right data from register 0 on the accelerometer.

Multiple-Register Read



Read X acceleration: `[0xa6 0x32[0xa7 rr]`

Here's how to read consecutive registers. The datasheet says the chip will return the next register in order, for every byte we read. So if we start at register 0x32 and read two bytes, we'll get the value for register 0x32 in the first byte we read, and the value for register 0x33 in the second byte we read. We could keep reading bytes and get the Y and Z axis data if we wanted to...

Read X Acceleration?

```
I2C>[0xa6 0x32[0xa7 rr]
I2C START BIT
WRITE: 0xA6 ACK
WRITE: 0x32 ACK
I2C START BIT
WRITE: 0xA7 ACK
READ: 0x00
READ:  ACK 0x00
NACK
I2C STOP BIT
I2C>
```


Back to the Bus Pirate.

Hmmm, the bytes are coming back as zeros. That's the default register value. No matter how I orient the accelerometer, I get zeros. Something's wrong.

Y U NO MEASURE?

Measure Bit

A setting of 0 in the measure bit places the part into standby mode, and a setting of 1 places the part into measurement mode. The ADXL345 powers up in standby mode with minimum power consumption.

Flipping through the datasheet...
By default, the MEASURE bit is zero. When the MEASURE bit is zero, the chip does not measure acceleration. Most chips do this -- they come up in low-power mode, and wait for a command to turn on and do their work. This way, it's easier for low-power devices (mobile phones, video game controllers, whatever) to turn on and off quickly, and avoid burning any more battery power than necessary.

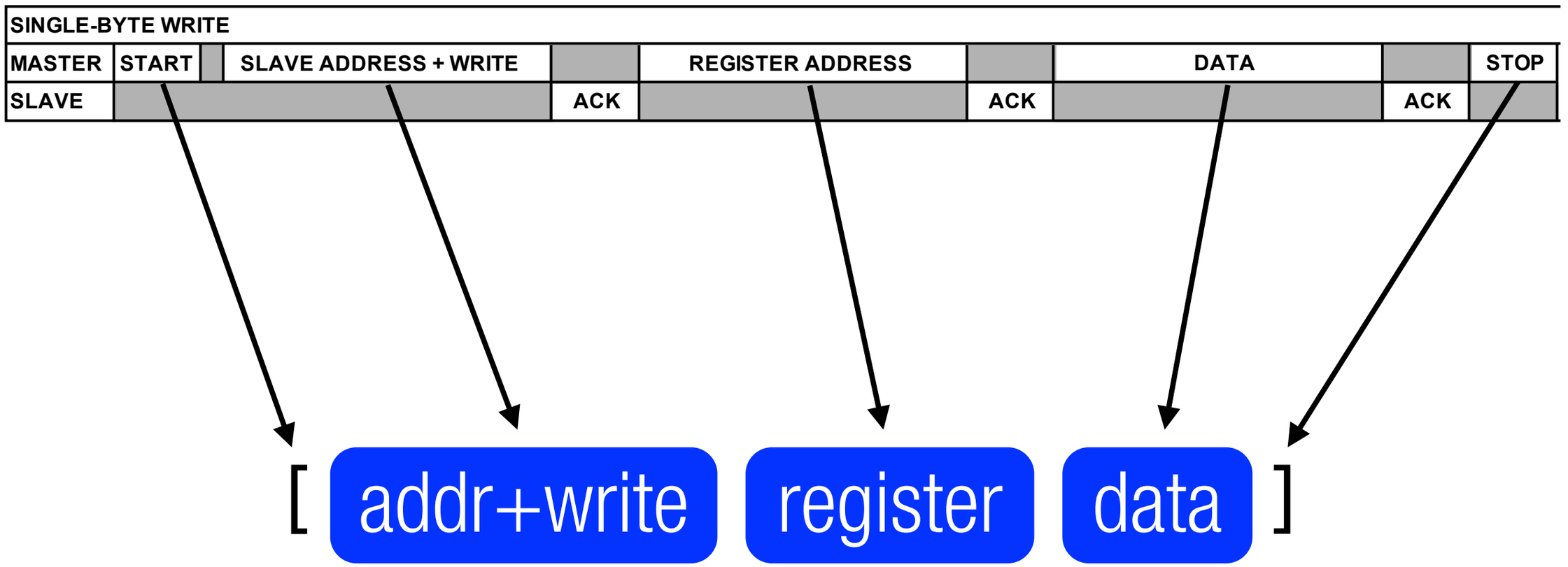
Device Setup

Register 0x2D—POWER_CTL (Read/Write)

D7	D6	D5	D4	D3	D2	D1	D0
0	0	Link	AUTO_SLEEP	Measure	Sleep	Wakeup	

So how do we turn on the MEASURE bit? The datasheet says that bit is in register 2d, so we'll need to write the correct value that register inside the accelerometer, via I2C. The MEASURE bit is the third bit (starting from zero), and all the other bits can be zero (according to the data sheet). So I need to write 0x08 to this register to make the chip measure acceleration.

Single-Register Write



Set MEASURE bit in POWER_CTL register: `[0xa6 0x2d 0x08]`

Tuesday, February 26, 13

From the datasheet, here's how to write to a register in the accelerometer, via I2C. It turns out to be easier than *reading* registers!

Double-Check Your Writes

```
I2C>[0xa6 0x2d 0x08]
I2C START BIT
WRITE: 0xA6 ACK
WRITE: 0x2D ACK
WRITE: 0x08 ACK
I2C STOP BIT
I2C>[0xa6 0x2d[0xa7 r]
I2C START BIT
WRITE: 0xA6 ACK
WRITE: 0x2D ACK
I2C START BIT
WRITE: 0xA7 ACK
READ: 0x08
NACK
I2C STOP BIT
I2C>
```


Trying this on the Bus Pirate...
First, I write the value to the POWER_CTL register. See where the 0x08 is sent to register 0x2d?
Then, I read the value back, to verify that it works.
Yep, the data (0x08) is in register 0x2d.

If you're doing something wrong, writing or reading registers, this is the easiest way to tell.

Read X Acceleration!

```
I2C>[0xa6 0x32[0xa7 rr]
I2C START BIT
WRITE: 0xA6 ACK
WRITE: 0x32 ACK
I2C START BIT
WRITE: 0xA7 ACK
READ: 0x7C
READ:  ACK 0x00
NACK
I2C STOP BIT
I2C>
```


Now, let's try reading the X acceleration value again.

That's better! Multiple reads, different values in different positions. The first read is from register 0x32, and the second read is from the next register, 0x33. 0x32 holds the low portion of the X acceleration value, and 0x33 holds the high portion of the X acceleration value.

Known-Good Commands

Write Power Control register, turn on MEASURE bit.	`[0xa6 0x2d 0x08]`
Read device ID (register 0x00)	`[0xa6 0x00[0xa7 r]`
Read X acceleration (registers 0x32, 0x33)	`[0xa6 0x32[0xa7 rr]`

So now we know how to initialize the accelerometer, to make it start measuring acceleration.
We know how to read the device ID (so we know it's the ADXL345 accelerometer).
And we know how to read X acceleration values from the accelerometer.

Now, we can go to our microcontroller of choice and write some code, and we'll know that the I2C commands we're sending are correct!

Victory!

Go Forth and Measure.

Thanks for listening. If you have further questions or want to know some of the more gory details of I2C that I couldn't talk about in twenty minutes, feel free to ask now, or find me after the "official" portion of tonight's Dorkbot.