I2C

CONTRIBUTORS: SFUPTOWNMAKER

Introduction

In this tutorial, you will learn all about the I^{2}C communication protocol, why you would want to use it, and how it's implemented.

The Inter-integrated Circuit (I^{2}C) Protocol is a protocol intended to allow multiple "slave" digital integrated circuits ("chips") to communicate with one or more "master" chips. Like the Serial Peripheral Interface (SPI), it is only intended for short distance communications within a single device. Like Asynchronous Serial Interfaces (such as RS-232 or UARTs), it only requires two signal wires to exchange information.

Suggested Reading

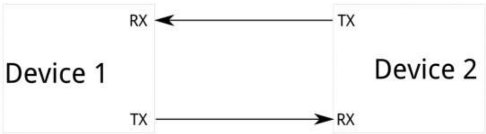
Stuff that would be helpful to know before reading this tutorial:

- Binary
- Serial Communication
- Serial Peripheral Interface
- Shift registers
- Logic Levels

Why Use I2C?

To figure out why one might want to communicate over I^{2}C, you must first compare it to the other available options to see how it differs.

What's Wrong with Serial Ports?



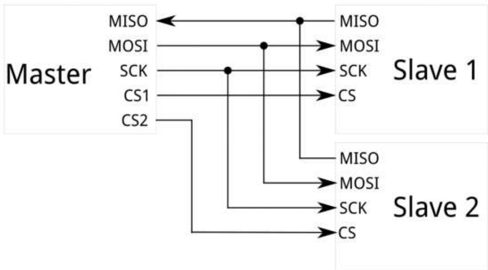
Because serial ports are **asynchronous** (no clock data is transmitted), devices using them must agree ahead of time on a data rate. The two devices must also have clocks that are close to the same rate, and will remain so–excessive differences between clock rates on either end will cause garbled data.

Asynchronous serial ports require hardware overhead–the UART at either end is relatively complex and difficult to accurately implement in software if necessary. At least one start and stop bit is a part of each frame of data, meaning that 10 bits of transmission time are required for each 8 bits of data sent, which eats into the data rate.

Another core fault in asynchronous serial ports is that they are inherently suited to communications between two, and only two, devices. While it is *possible* to connect multiple devices to a single serial port, **bus contention** (where two devices attempt to drive the same line at the same time) is always an issue and must be dealt with carefully to prevent damage to the devices in question, usually through external hardware.

Finally, data rate is an issue. While there is no *theoretical* limit to asynchronous serial communications, most UART devices only support a certain set of fixed baud rates, and the highest of these is usually around 230400 bits per second.

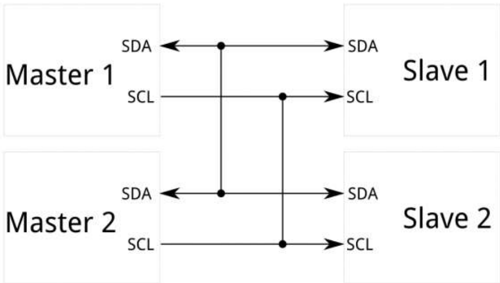
What's Wrong with SPI?



The most obvious drawback of SPI is the number of pins required. Connecting a single master to a single slave with an SPI bus requires four lines; each additional slave requires one additional chip select I/O pin on the master. The rapid proliferation of pin connections makes it undesirable in situations where lots of devices must be slaved to one master. Also, the large number of connections for each device can make routing signals more difficult in tight PCB layout situations. SPI only allows one master on the bus, but it does support an arbitrary number of slaves (subject only to the drive capability of the devices connected to the bus and the number of chip select pins available).

SPI is good for high data rate **full-duplex** (simultaneous sending and receiving of data) connections, supporting clock rates upwards of 10MHz (and thus, 10 million bits per second) for some devices, and the speed scales nicely. The hardware at either end is usually a very simple shift register, allowing easy implementation in software.

Enter I^{2}C - The Best of Both Worlds!



I^{2}C requires a mere two wires, like asynchronous serial, but those two wires can support up to 1008 slave devices. Also, unlike SPI, I^{2}C can support a multi-master system, allowing more than one master to communicate with all devices on the bus (although the master devices can't talk to each other over the bus and must take turns using the bus lines).

Data rates fall between asynchronous serial and SPI; most I^{2}C devices can communicate at 100kHz or 400kHz. There is some overhead with I^{2}C; for every 8 bits of data to be sent, one extra bit of meta data (the "ACK/NACK" bit, which we'll discuss later) must be transmitted.

The hardware required to implement I^{2}C is more complex than SPI, but less than asynchronous serial. It can be fairly trivially implemented in software.

I^2C - A Brief History

I^2C was originally developed in 1982 by Philips for various Philips chips. The original spec allowed for only 100kHz communications, and provided only for 7-bit addresses, limiting the number of devices on the bus to 112 (there are several reserved addresses, which will never be used for valid I^2C addresses). In 1992, the first public specification was published, adding a 400kHz fast-mode as well as an expanded 10-bit address space. Much of the time (for instance, in the ATMega328 device on many Arduino-compatible boards) , device support for I^2C ends at this point. There are three additional modes specified: fast-mode plus, at 1MHz; high-speed mode, at 3.4MHz; and ultra-fast mode, at 5MHz.

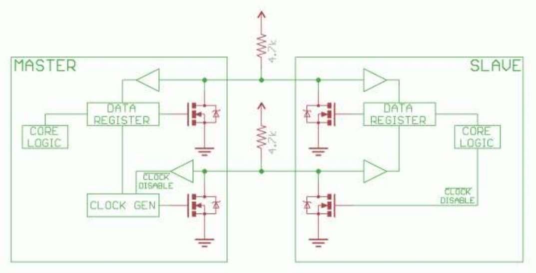
In addition to “vanilla” I^2C, Intel introduced a variant in 1995 call “System Management Bus” (SMBus). SMBus is a more tightly controlled format, intended to maximize predictability of communications between support ICs on PC motherboards. The most significant difference between SMBus is that it limits speeds from 10kHz to 100kHz, while I^2C can support devices from 0kHz to 5MHz. SMBus includes a clock timeout mode which makes low-speed operations illegal, although many SMBus devices will support it anyway to maximize interoperability with embedded I^2C systems.

I2C at the Hardware Level

Signals

Each I^2C bus consists of two signals: SCL and SDA. SCL is the clock signal, and SDA is the data signal. The clock signal is always generated by the current bus master; some slave devices may force the clock low at times to delay the master sending more data (or to require more time to prepare data before the master attempts to clock it out). This is called “clock stretching” and is described on the protocol page.

Unlike UART or SPI connections, the I^2C bus drivers are “open drain”, meaning that they can pull the corresponding signal line low, but cannot drive it high. Thus, there can be no bus contention where one device is trying to drive the line high while another tries to pull it low, eliminating the potential for damage to the drivers or excessive power dissipation in the system. Each signal line has a pull-up resistor on it, to restore the signal to high when no device is asserting it low.



Notice the two pull-up resistors on the two communication lines.

Resistor selection varies with devices on the bus, but a good rule of thumb is to start with 4.7k and adjust down if necessary. I^2C is a fairly robust protocol, and can be used with short runs of wire (2-3m). For long runs, or systems with lots of devices, smaller resistors are better.

Signal Levels

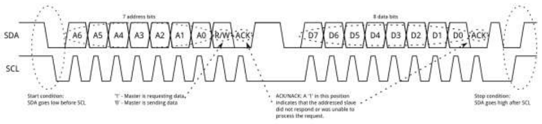
Since the devices on the bus don't actually drive the signals high, I^2C allows for some flexibility in connecting devices with different I/O voltages. In general, in a system where one device is at a higher voltage than another, it may be possible to connect the two devices via I^2C without any level shifting circuitry in between them. The trick is to connect the pull-up resistors to the lower of the two voltages. This only works in some cases, where the lower of the two system voltages exceeds the high-level input voltage of the the higher voltage system–for example, a 5V Arduino and a 3.3V accelerometer.

If the voltage difference between the two systems is too great (say, 5V and 2.5V), SparkFun offers a simple I^2C level shifter board. Since the board also includes an enable line, it can be used to disable communications to selected devices. This is useful in cases where more than one device with the same address is to be connected to a single master–Wii Nunchucks are a good example.

Protocol

Communication via I^2C is more complex than with a UART or SPI solution. The signalling must adhere to a certain protocol for the devices on the bus to recognize it as valid I^2C communications. Fortunately, most devices take care of all the fiddly details for you, allowing you to concentrate on the data you wish to exchange.

Basics



Messages are broken up into two types of frame: an address frame, where the master indicates the slave to which the message is being sent, and one or more data frames, which are 8-bit data messages passed from master to slave or vice versa. Data is placed on the SDA line after SCL goes low, and is sampled after the SCL line goes high. The time between clock edge and data read/write is defined by the devices on the bus and will vary from chip to chip.

Start Condition

To initiate the address frame, the master device leaves SCL high and pulls SDA low. This puts all slave devices on notice that a transmission is about to start. If two master devices wish to take ownership of the bus at one time, whichever device pulls SDA low first wins the race and gains control of the bus. It is possible to issue repeated starts, initiating a new communication sequence without relinquishing control of the bus to other masters; we'll talk about that later.

Address Frame

The address frame is always first in any new communication sequence. For a 7-bit address, the address is clocked out most significant bit (MSB) first, followed by a R/W bit indicating whether this is a read (1) or write (0) operation.

The 9th bit of the frame is the NACK/ACK bit. This is the case for all frames (data or address). Once the first 8 bits of the frame are sent, the receiving device is given control over SDA. If the receiving device does not pull the SDA line low before the 9th clock pulse, it can be inferred that the receiving device either did not receive the data or did not know how to parse the message. In that case, the exchange halts, and it's up to the master of the system to decide how to proceed.

Data Frames

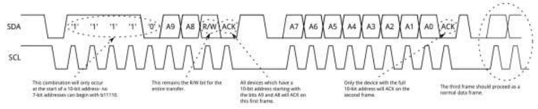
After the address frame has been sent, data can begin being transmitted. The master will simply continue generating clock pulses at a regular interval, and the data will be placed on SDA by either the master or the slave, depending on whether the R/W bit indicated a read or write operation. The number of data frames is arbitrary, and most slave devices will auto-increment the internal register, meaning that subsequent reads or writes will come from the next register in line.

Stop condition

Once all the data frames have been sent, the master will generate a stop condition. Stop conditions are defined by a 0->1 (low to high) transition on SDA *after* a 0->1 transition on SCL, with SCL remaining high. During normal data writing operation, the value on SDA should **not** change when SCL is high, to avoid false stop conditions.

Advanced Protocol Topics

10-bit Addresses



In a 10-bit addressing system, two frames are required to transmit the slave address. The first frame will consist of the code b11110xyz, where 'x' is the MSB of the slave address, y is bit 8 of the slave address, and z is the read/write bit as described above. The first frame's ACK bit will be asserted by all slaves which match the first two bits of the address. As with a normal 7-bit transfer, another transfer begins immediately, and this transfer contains bits 7:0 of the address. At this point, the addressed slave should respond with an ACK bit. If it doesn't, the failure mode is the same as a 7-bit system.

Note that 10-bit address devices can coexist with 7-bit address devices, since the leading '11110' part of the address is not a part of any valid 7-bit addresses.

Repeated Start Conditions