I2c and SPI

I2C and SPI are two peripheral busses

Both are synchronous serial lines

- Synchronous means that the clock is delivered with the signal
- The serial port it _a_synchronous
  - UART = Universal Asynchronous Receiver Transmitter

They replaced the "textbook" legacy data bus / address bus

- Such interfaces are complex to manage, and requires many tracks
- Nowadays they are only used for RAM, and little else

Another interesting bus is "OneWire", that we won't cover today

A simple summary, that encompasses everything:

- UART (RS232) is two wires, asynchronous, point to point.
- I2C is two wires, one master and slaves
- SPI is three wires + enable, one master and slaves
- OneWire is one wire (+ GND), one master and slaves

I2C: General

Patented by Philips ages ago.

- Patents are supposed to be expired, but nobody knows for sure.
- Do not read patent documents. You get infected. And guilty.
- TWI, SMBUS and other names are the same thing.
  - IF you use TWI you can claim it's not I2C.

The name means "Inter Intgrated Circuit" bus: IIC, or I2C.

- Originalli 100kb/s, then 400kb/s, now 1MB/s

Two lines, plus power and ground:

- SCL: serial clock, open-drain, pulled up by a resistor
- SDA: serial data, open-drain, pulled up by a resistor

SDA can only move while SCL is low

- There is only one exception: the "start" condition
- And there is only one more exception: the "stop" condition

I2C: Electrical

I2C is very simple, you only need to master it

0V and 3.3 Volts. Or whatever

- Just agree about the positive level with your peripherals.
- No special impedance requirement (lines are short anyways)
- The timing is driven by the master, no special requirement
  - Only, don't be too fast

You can drive I2C from GPIO

- It works if your GPIO pins are open-drain.
- But it also works if you can switch between input and output-low

You usually can't be an I2C slave through GPIO

- You need to obey master timing choices

If something goes wrong, the oscilloscope helps, as usual

- Even a TDC (a logic analyzer) can help, but not for all bugs.

I2C in LPC11Uxx

Our microcontroller includes an I2C controller

- It is on pins 15 and 16
  - The pins are open-drain (no output-1 capability)
  - The only pins without an internal pull-up

You can run as many I2C buses as you want, one GPIO pins

- Just set the GPIO data register to 0
- Then toggle between input and output mode

Other (older) LPC11 devices are much more braindead

- When you turn from input to output, the pin value is preserved
  - They help you to prevent glitches
  - They prevent you from doing anything useful
- So pins 15 and 16 are open-drain as a bug-fix for this
  - Two bugs sometimes cancel one another, but they remain bugs

I2C Framing

Communication is byte-based, with an ack bit

- When the master transmits, the slave must ack or nack
- When the slave transmits, the master must ack or nack

Everything starts with "start"

Everything stops with "stop"

Frames are master-driven, and each addresses a single slave

- There is no such thing as auto-detection
- Still you can probe for slave presence
  - Send its address, look for ack, abort

I2C Addressing

Devices feature a 7-bit address, plus one R/W bit

- The R/W bit is the LSB
- You never know if people talks 7 or 8 bits
- Chip documentation usually hides the address very well
    - You read from page 1 to page "end", so not a problem

Write frame:

- Start + WAddress
- Data out (0 or more bytes), then Stop

Read frame

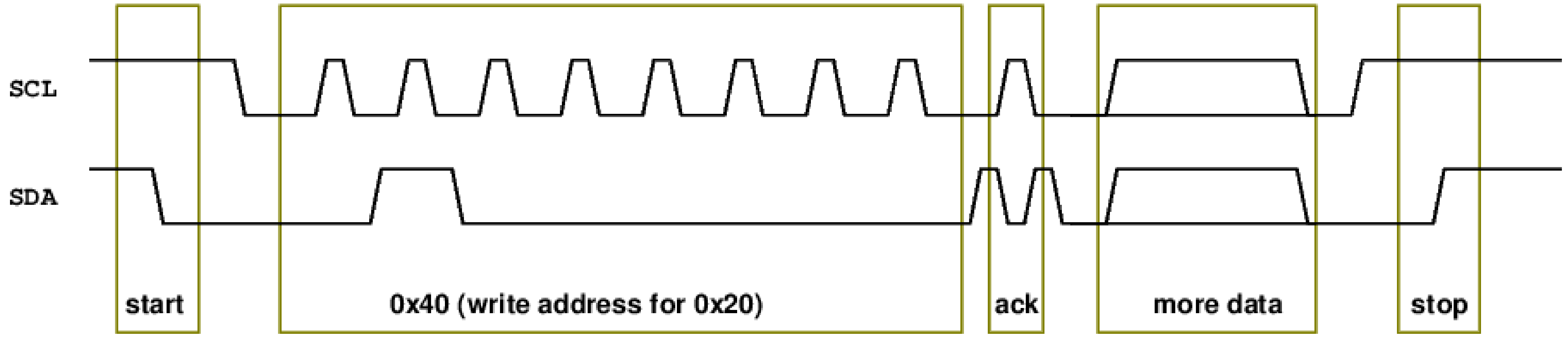
- Start + WAddress
- Internal address (0 or more bytes) (no stop)
- Start + RAddress
- Data in (0 or more bytes), then stop

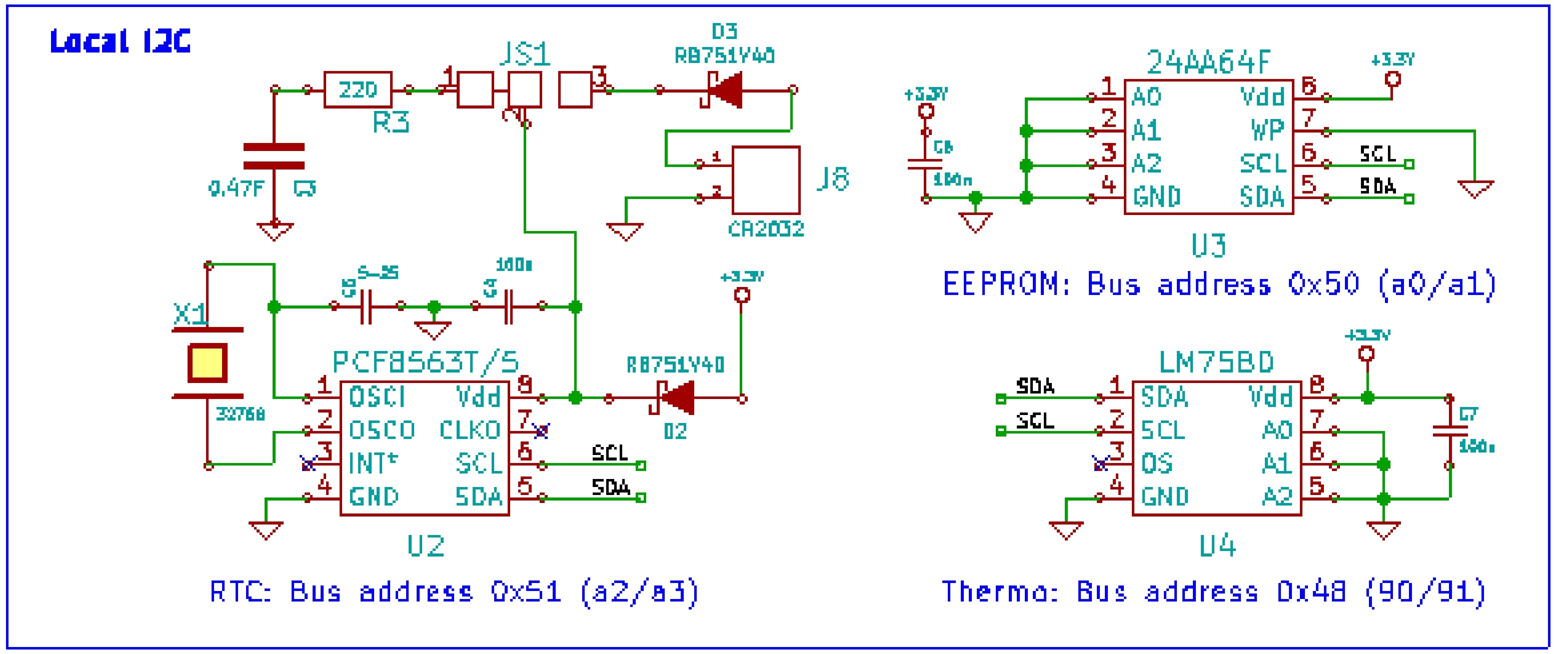
Some addresses are reserved for some special things

- Who cares, we are only master

I2C signals (eventually!)

Signals are driven from the master, but during ACK the SDA line is driven by the slave





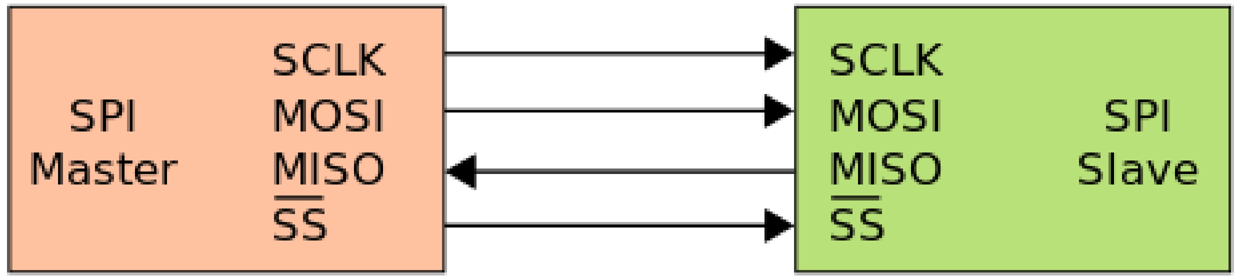
SPI: Serial Peripheral interface

Based on 4 wires

- SCKL (serial clock), MISO (master in slave out), MOSI, SSEL
- The bus is synchronous
- No line is bidirectional
- Usually run at a few MHz

No addressing defined

- You need a chip-select for each slave
- This can be called SSEL (slave select), SS or CS



SPI: Electrical

There is no special electrical requirement

- But according to you working speed you need to care about
  - Capacitive coupling
  - Bounces

Usually transmission happens at 0/3.3V, but it is variable

- Like what happens with I2C, you must agree with the peripheral
- Most peripherals nowadays are 2.7-3.6V or 1.7-3.6V

4 modes or operation are supported:

- CPOL (clock polarity) can be either one
- CPHA (clock phase) can be either one
- This is a serious problem when you connect more than 1 device

Fortunately, SSEL is always active low

- But sometimes you must pulse it high among words
- And sometimes you must keep it low for the whole transaction
- And always is not really always

SPI in LPC11Uxx

Our microcontroller family has 2 SPI controllers

- Then, MISO, MOSI and SCK can sometimes be routed to different pins
- They work as either master or slave (most uC do the same)

There logic cell includes a FIFO (everyone does)

- This allows some CPU relief
- You can manage an uninterrupted data stream

You can also do bit-banging SPI

- There are no specific GPIO requirements
- Bit-banging is slow, and not usually a wise choice
- A single SPI master can drive many peripherals, so we don't need more buses

I personally prefer driving SSEL from a GPIO pin

- You can have as many CS as you need, independent of hw features
- You can implement whatever policy for CS activation

SPI: Framing

There is no standard framing at all

- Unlike I2C, the bus is not shared, so there is no protocol
  - The bus is shared, but the individual CS is not
  - Every communication event is one-to-one

The structure of a frame depends on the peripheral

- Flash memory uses a command-response protocol
  - It's half-duplex, and usually you fill with 0xff
- With ADC chips, communication is usually full-duplex
  - You send parameters for the next sample while reading the previous one
  - Most often, the ADC speed is set by the SPI speed
- Some ADC devices use two CS lines, and one is active high.
- Some other chips offer a register array, and an IRQ pin

And there is the usual issue of CS policies.

- As usual, read the whole data sheet before routing/coding

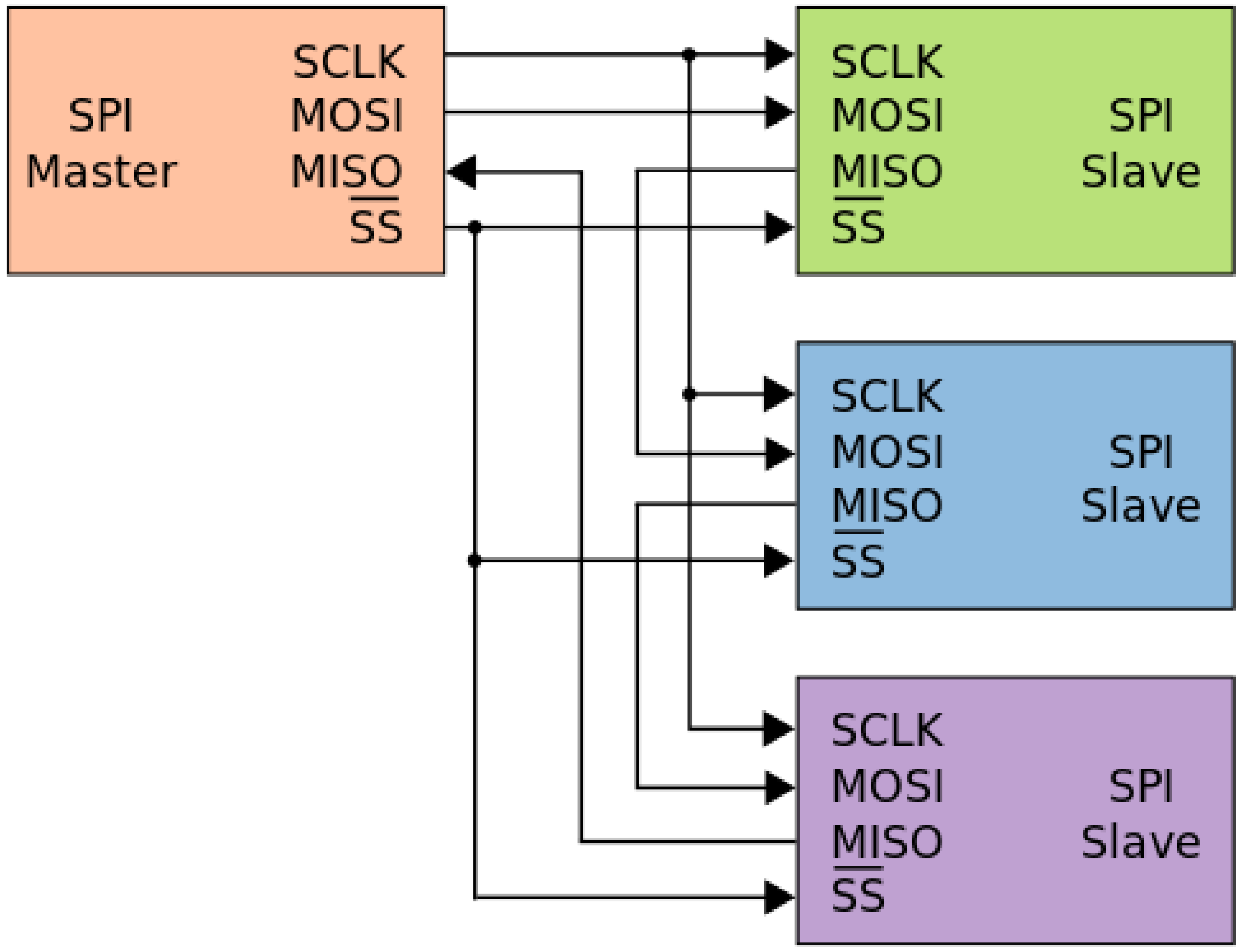
SPI: Variations on the base idea

Currently, quad-spi (QSPI, QPI) flash memory is common

- Same basics as SPI, but with 4 data lines
- Usually, commands are on MOSI and data is by nibbles
- The chip may support dual-SPI too
  - The hardware designer must trade between tracks and speed

Some devices provide for strange daisy-chain configurations

- Tey act like one device, e.g. many ADC channels
- Very handy, but not trivial to get right



SPI: API

By its features, SPI doesn't properly fit read()/write()

- You must always read back as many bytes as you wrote

You may setup a socket interface

- For any outcoing packet you receive an incoming one

Still, some devices (e.g. flash) work half duplex

- You can write and avoid reading back the empty reply
- You can read without filling a write buffer

So my choice is using this function as main engine:

```
int spi_xfer(struct spi_dev *dev, enum spi_flags flags,
       const struct spi_ibuf *ibuf, struct spi_obuf *obuf);
```

This works both for SPI master and SPI slave

- And either of ibuf and obuf can be NULL

SPI: Device API

Besides the transfer, we need an abstraction for the device

- We may rely on the fact that read-only memory is cheaper
- We must support various pin configurations

```
#define SPI_CS(x)                (-1 - (x)) /* arg is 0.., value is negative */
#define __SPI_IS_HW_CS(x)        ((x) < 0)
#define __SPI_GET_HW_CS(x)       (-(x) - 1)

/* This is an interface for SPI, master and slave */
struct spi_cfg {
        int gpio_cs;             /* Either GPIO number or SPI_CS(x) */
        int freq;                /* Suggested frequency */
        int timeout;             /* Jiffies */
        unsigned long flags;
        uint8_t pol, phase, bits, devn; /* devn selects spi0 or spi1 or more */
};

/* This would be opaque if we had malloc */
struct spi_dev {
        const   struct spi_cfg *cfg;
        unsigned long base;             int     current_freq;
};
extern struct spi_dev *spi_create(struct spi_dev *cfg);
extern void spi_destroy(struct spi_dev *dev);
```