e-ale

Authors and license

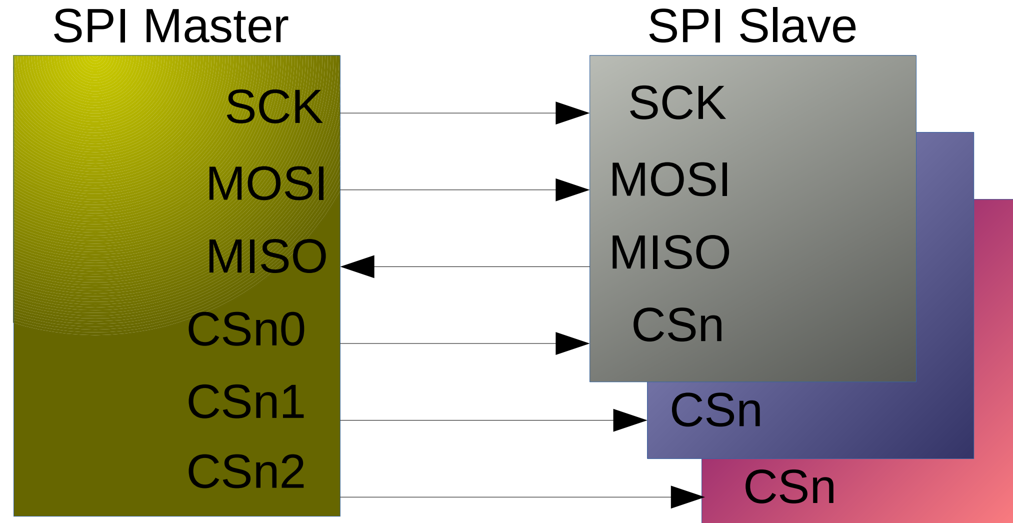
- Authors
  - Michael Welling
    QWERTY Embedded Design, LLC
    www.qwertyembedded.com
- License
  - Creative Commons Attribution – Share Alike 4.0
    https://creativecommons.org/licenses/by-sa/4.0/

What is SPI?

e-ale

- SPI (Serial Peripheral Interface) is a full duplex synchronous serial master/slave bus interface.
- De facto standard first developed at Motorola in the 1980s.
- A SPI bus consists of a single master device and possibly multiple slave devices.
- Typical device interface
  - SCK – serial clock
  - MISO – master in slave out
  - MOSI – master out slave in
  - CSn / SSn – chip select / slave select
  - IRQ / IRQn – interrupt

What is SPI?



Example SPI devices

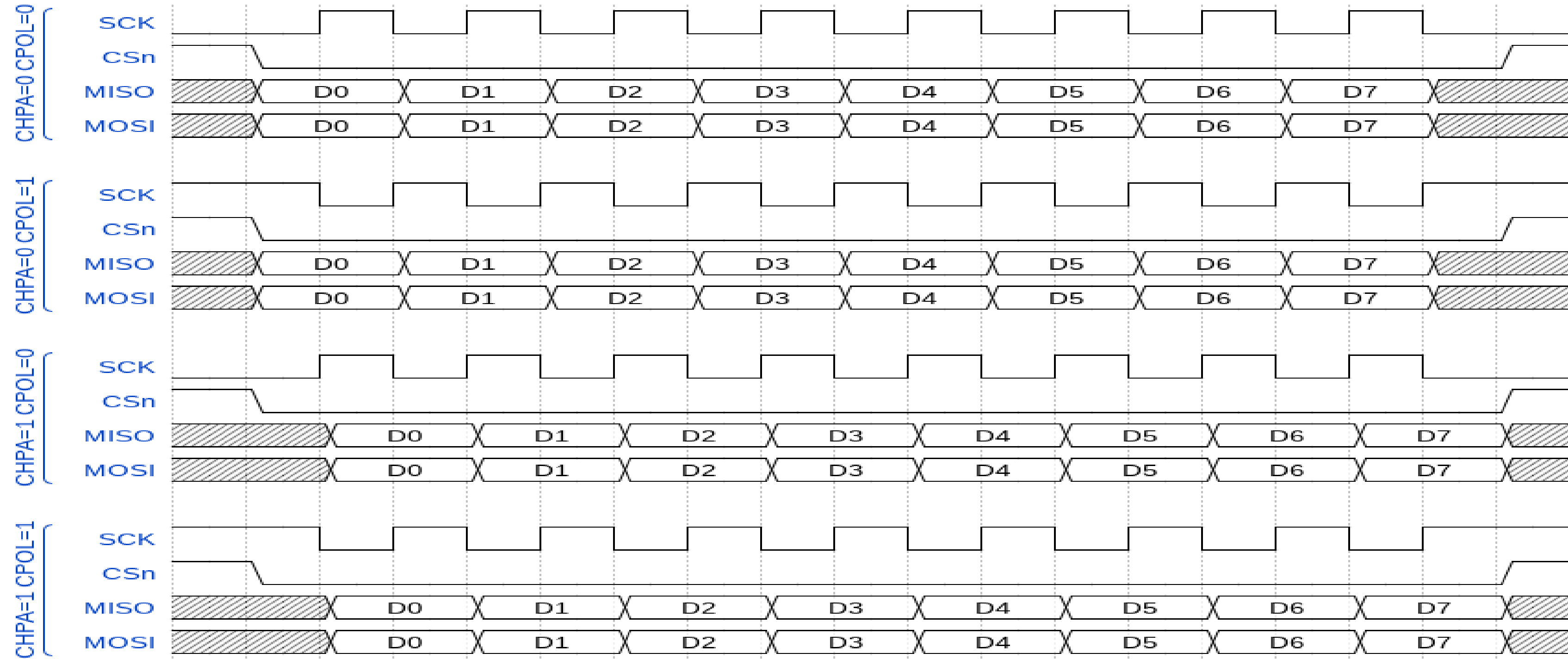
- Analog converters (ADC, DAC, CDC)
- Sensors (inertial, temperature, pressure)
- Serial LCD
- Serial Flash
- Touchscreen controllers
- FPGA programming interface

SPI Modes

- SPI Mode is typically represented by (CPOL, CPHA) tuple
  - CPOL – clock polarity
    - 0 = clock idles low
    - 1 = clock idles high
  - CPHA – clock phase
    - 0 = data latched on falling clock edge, output on rising
    - 1 = data latched on rising clock edge, output on falling
- Mode (0, 0) and (1, 1) are most commonly used.
- Sometimes listed in encoded form 0-3.

SPI Modes

e-ale



Linux SPI Subsystem

First developed in early 2000s (2.6 ERA) based on the work of several key developers in including:

- David Brownell
- Russell King
- Dmitry Pervushin
- Stephen Street
- Mark Underwood
- Andrew Victor
- Vitaly Wool

Linux SPI Subsystem

Past maintainers of the Linux SPI subsystem:

- David Brownell
- Grant Likely

Current maintainer:

- Mark Brown

e-ale

Linux SPI Mailinglist

List: linux-spi; (subscribe / unsubscribe)
Info:

This is the mailing list for the Linux SPI subsystem.

Archives:
http://marc.info/?l=linux-spi
Footer:

To unsubscribe from this list: send the line "unsubscribe linux-spi" in the body of a message to majordomo@vger.kernel.org
More majordomo info at http://vger.kernel.org/majordomo-info.html

e-ale

Controller Drivers

- Controller drivers are used to abstract and drive transactions on an SPI master.
- The host SPI peripheral registers are accessed by callbacks provided to the SPI core driver. (drivers/spi/spi.c)
- struct spi_controller

Controller Drivers

- Allocate a controller
  - **spi_alloc_master()**

- Set controller fields and methods
  - mode_bits - flags e.g. **SPI_CPOL, SPI_CPHA, SPI_NO_CS, SPI_CS_HIGH, SPI_RX_QUAD, SPI_LOOP**
  - **.setup()** - configure SPI parameters
  - **.cleanup()** - prepare for driver removal
  - **.transfer_one_message()/.transfer_one()** - dispatch one msg/transfer (mutually exclusive)

- Register a controller
  - **spi_register_master()**

Controller Devicetree Binding

The SPI controller node requires the following properties:
- compatible - Name of SPI bus controller following generic names recommended practice.

In master mode, the SPI controller node requires the following additional properties:
- #address-cells - number of cells required to define a chip select address on the SPI bus.
- #size-cells - should be zero.

Optional properties (master mode only):
- cs-gpios - gpios chip select.
- num-cs - total number of chipselects.

So if for example the controller has 2 CS lines, and the cs-gpios property looks like this:

```
cs-gpios = <&gpio1 0 0>, <0>, <&gpio1 1 0>, <&gpio1 2 0>;
```

Controller Devicetree Binding

Example:

```
spi1: spi@481a0000 {
        compatible = "ti,omap4-mcspi";
        #address-cells = <1>;
        #size-cells = <0>;
        reg = <0x481a0000 0x400>;
        interrupts = <125>;
        ti,spi-num-cs = <2>;
        ti,hwmods = "spi1";
        dmas = <&edma 42 0
                &edma 43 0
                &edma 44 0
                &edma 45 0>;
        dma-names = "tx0", "rx0", "tx1", "rx1";
        status = "disabled";
};
```

Protocol Drivers

- For each SPI slave you intend on accessing, you have a protocol driver. SPI protocol drivers can be found in many Linux driver subsystems (iio, input, mtd).
- Messages and transfers are used to communicate to slave devices via the SPI core and are directed to the respective controller driver transparently.
- A **struct spi_device** is passed to the probe and remove functions to pass information about the host.

Protocol Drivers

- Transfers
  - A single operation between master and slave
  - RX and TX buffers pointers are supplied
  - Option chip select behavior and delays
- Messages
  - Atomic sequence of transfers
  - Argument to SPI subsystem read/write APIs

e-ale

struct spi_device

```
struct spi_device {
	struct device dev;
	struct spi_controller * controller;
	struct spi_controller * master;
	u32 max_speed_hz;
	u8 chip_select;
	u8 bits_per_word;
	u16 mode;
	int irq;
	void * controller_state;
	void * controller_data;
	char modalias;
	int cs_gpio;
	struct spi_statistics statistics;
};
```

Controller side proxy for an SPI slave device. Passed to the probe and remove functions with values based on the host configuration.

struct spi_device

```
#define SPI_CPHA       0x01              /* clock phase */
#define SPI_CPOL       0x02              /* clock polarity */
#define SPI_MODE_0     (0|0)             /* (original MicroWire) */
#define SPI_MODE_1     (0|SPI_CPHA)
#define SPI_MODE_2     (SPI_CPOL|0)
#define SPI_MODE_3     (SPI_CPOL|SPI_CPHA)
#define SPI_CS_HIGH    0x04              /* chipselect active high? */
#define SPI_LSB_FIRST  0x08              /* per-word bits-on-wire */
#define SPI_3WIRE      0x10              /* SI/SO signals shared */
#define SPI_LOOP       0x20              /* loopback mode */
#define SPI_NO_CS      0x40              /* 1 dev/bus, no chipselect */
#define SPI_READY      0x80              /* slave pulls low to pause */
#define SPI_TX_DUAL    0x100             /* transmit with 2 wires */
#define SPI_TX_QUAD    0x200             /* transmit with 4 wires */
#define SPI_RX_DUAL    0x400             /* receive with 2 wires */
#define SPI_RX_QUAD    0x800             /* receive with 4 wires */
```

Probe Function

```
static int myspi_probe(struct spi_device *spi)
{
	struct myspi			*chip;
	struct myspi_platform_data	*pdata, local_pdata;

	...
}
```

Probe Function

```
static int myspi_probe(struct spi_device *spi)
{
	...
	match = of_match_device(of_match_ptr(myspi_of_match), &spi->dev);
	if (match) {
		/* parse device tree options */
		pdata = &local_pdata;
		...
	}
	else {
		/* use platform data */
		pdata = &spi->dev.platform_data;
		if (!pdata)
			return -ENODEV;
	}
	...
}
```

Probe Function

```
static int myspi_probe(struct spi_device *spi)
{
	...

	/* get memory for driver's per-chip state */
	chip = devm_kzalloc(&spi->dev, sizeof *chip, GFP_KERNEL);
	if (!chip)
		return -ENOMEM;

	spi_set_drvdata(spi, chip);

	...
	return 0;
}
```

OF Device Table

Example:

```
static const struct of_device_id myspi_of_match[] = {
	{
		.compatible = "mycompany,myspi",
		.data = (void *) MYSPI_DATA,
	},
	{ },
};
MODULE_DEVICE_TABLE(of, myspi_of_match);
```

SPI Device Table

Example:

```
static const struct spi_device_id myspi_id_table[] = {
	{ "myspi", MYSPI_TYPE },
	{ },
};
MODULE_DEVICE_TABLE(spi, myspi_id_table);
```

struct spi_driver

```
struct spi_driver {
	const struct spi_device_id * id_table;
	int (* probe) (struct spi_device *spi);
	int (* remove) (struct spi_device *spi);
	void (* shutdown) (struct spi_device *spi);
	struct device_driver driver;
};
```

e-ale

struct spi_driver

Example:

```
static struct spi_driver myspi_driver = {
	.driver = {
		.name = "myspi_spi",
		.pm = &myspi_pm_ops,
		.of_match_table = of_match_ptr(myspi_of_match),
	},
	.probe = myspi_probe,
	.id_table = myspi_id_table,
};
module_spi_driver(myspi_driver);
```

Kernel APIs

- **spi_async()**
  - asynchronous message request
  - callback executed upon message complete
  - can be issued in any context
- **spi_sync()**
  - synchronous message request
  - may only be issued in a context that can sleep (i.e. not in IRQ context)
  - wrapper around spi_async()
- **spi_write()/spi_read()**
  - helper functions wrapping spi_sync()

Kernel APIs

- **spi_read_flash()**
  - Optimized call for SPI flash commands
  - Supports controllers that translate MMIO accesses into standard SPI flash commands
- **spi_message_init()**
  - Initialize empty message
- **spi_message_add_tail()**
  - Add transfers to the message’s transfer list

Slave Node Devicetree Binding

SPI slave nodes must be children of the SPI controller node.

In master mode, one or more slave nodes (up to the number of chip selects) can be present.

Required properties are:
- compatible - Name of SPI device following generic names recommended practice.
- reg - Chip select address of device.
- spi-max-frequency - Maximum SPI clocking speed of device in Hz.

Slave Node Devicetree Binding

All slave nodes can contain the following optional properties:
- spi-cpol - Empty property indicating device requires inverse clock polarity (CPOL) mode.
- spi-cpha - Empty property indicating device requires shifted clock phase (CPHA) mode.
- spi-cs-high - Empty property indicating device requires chip select active high.
- spi-3wire - Empty property indicating device requires 3-wire mode.
- spi-lsb-first - Empty property indicating device requires LSB first mode.

- spi-tx-bus-width - The bus width that is used for MOSI. Defaults to 1 if not present.
- spi-rx-bus-width - The bus width that is used for MISO. Defaults to 1 if not present.

- spi-rx-delay-us - Microsecond delay after a read transfer.
- spi-tx-delay-us - Microsecond delay after a write transfer.

Slave Node Devicetree Binding

Example:

```
&spi1 {
	#address-cells = <1>;
	#size-cells = <0>;
	status = "okay";

	pinctrl-names = "default";
	pinctrl-0 = <&spi1_pins>;
	myspi@0 {
		compatible = "mycompany,myspi";
		spi-max-frequency = <2000000>;
		spi-cpha;
		...
		reg = <0>;
	};
	...
};
```

Platform Registration

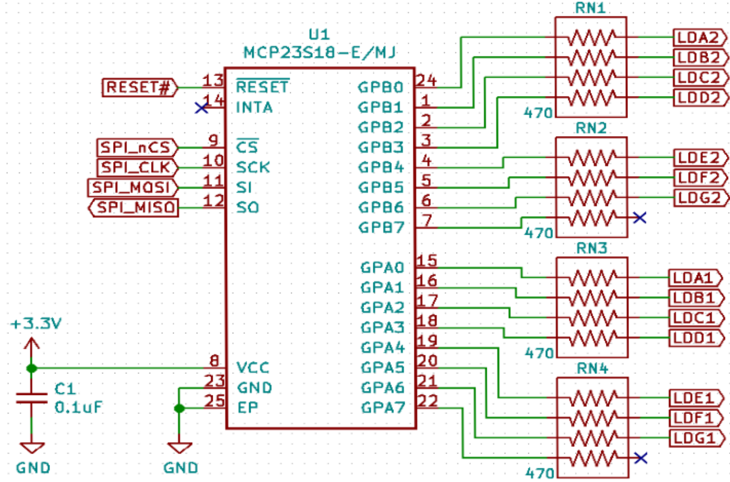
```
struct spi_board_info {
	char modalias;
	const void * platform_data;
	const struct property_entry * properties;
	void * controller_data;
	int irq;
	u32 max_speed_hz;
	u16 bus_num;
	u16 chip_select;
	u16 mode;
};
```

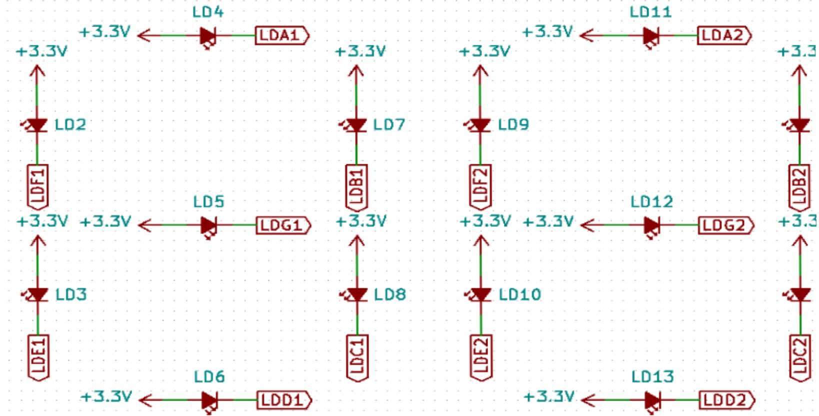
Platform Registration

Example:

```
static struct spi_board_info myspi_board_info[] = {
        {
                .modalias = "myspi",
                .platform_data = &myspi_info,
                .irq = MYIRQ,
                .max_speed_hz = 2000000,
                .chip_select = 2,
                …..
        },
};
```

Baconbits SPI Hardware





EMULATED 7 SEGMENTS

Questions?

Thank you!