MCP23sxx microchip gpio extender in the IMX kernel

The microchip MXP23sXX is a gpio extender – gpio-controller driver

It support both I2C and SPi for 8 and 16 bits

The file name in the kernel is

gpio-mcp23s08.c

But as mentioned in the beginning of the file, it support four flavors of the chip.

/\*\*

\* MCP types supported by driver

\*/

#define MCP\_TYPE\_S08 0

#define MCP\_TYPE\_S17 1

#define MCP\_TYPE\_008 2

#define MCP\_TYPE\_017 3

The driver is configured first from the device tree , the information in the device tree is used in the driver to configure the devices.

This documents will focus on the SPI flavor of MCP23S17 - the 16 bit gpio chip.

I am saying gpio expander but the kernel call it gpio-controller

To understand better the gpio-controller one need to read chapter 15 in the **book**

Chapter 15. GPIO Controller Drivers – gpio\_chip





I don’t like the name gpio-controller , gpio-expander driver is much more make sense.

So now we know that the gpio-mcp23s08.c in the kernel is a gpio-controller, gpio expander driver, let’s see how it actually build.

There are two ways to write the driver.

One, like we see in the Ethernet, write a SPI or I2C calls according to the datasheet.

The second way is the Linux way.

Why?

When a driver is build in the Arm Linux it is written in a way that:

1. It can be configured by the device tree.

Passing few properties , spi-present-mask , chip select , irq

We can pass parameters to the driver without compile the kernel and change the kernel,

The device driver support 4 chips and can support all N chips on one bus, so the number of chips and the type of the chip are all controlled via the device tree.

1. It expose a standard interface , using the sysfs interface.

It is explain in the book:

Sysfs interface for GPIO controller On successful **gpiochip\_add(),** a directory entry with a path like **/sys/class/gpio/gpiochipX**/ will be created, where X is the GPIO controller base (controller providing GPIOs starting at #X), having the following attributes: base, whose value is same as X, and which corresponds to gpio\_chip.base (if assigned statically), and being the first GPIO managed by this chip. label, which is provided for diagnostics (not always unique). ngpio, which tells how many GPIOs this controller provides (N to N + ngpio - 1). This is the same as defined in gpio\_chip.ngpios. All of the preceding attributes are read-only.

So we can access to the chip from use space , if is echo command or c functions like any normal gpio.

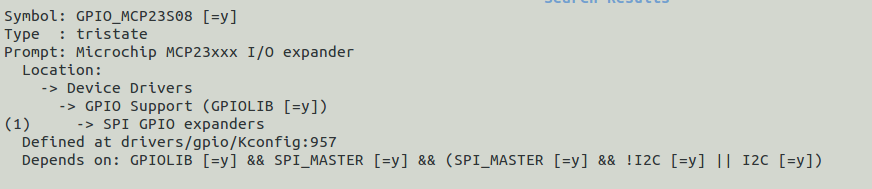
This is the entry in the device tree of the SPI 1

Note, it cannot hold both SPI dev interface and gpio controller.

It can be only one interface:

Note:

We need to add the driver inside the kernel, search for MCP23xx



Here we define two interface just for showing the option.

&ecspi1 {

pinctrl-names = "default";

pinctrl-0 = <&pinctrl\_ecspi1>;

fsl,spi-num-chipselects = <4>;

cs-gpios = <

&gpio5 17 GPIO\_ACTIVE\_LOW

&gpio3 19 GPIO\_ACTIVE\_LOW

&gpio3 24 GPIO\_ACTIVE\_LOW

&gpio3 25 GPIO\_ACTIVE\_LOW

>;

status = "okay";

spidev0: spi@0 {

compatible = "spidev";

reg = <0>;

spi-max-frequency = <54000000>;

};

gpiom1: gpio@2 {

compatible = "microchip,mcp23s17";

#gpio-cells = <2>;

gpio-controller;

reg = <2>;

microchip,spi-present-mask = <0x03>;

spi-max-frequency = <10000000>;

};

};

I think the spidev can work with the gpio-controller ( gpio –expander ) because the spi will be loaded by the kernel and the host can use it , the gpio-controller can use it

Of course not together but they can.

**Spi preset mask**

This is the code from the kernel driver:

for (addr = 0; addr < ARRAY\_SIZE(pdata->chip); addr++) {

if (!(spi\_present\_mask & (1 << addr)))

continue;

chips--;

data->mcp[addr] = &data->chip[chips];

data->mcp[addr]->irq = spi->irq;

status = mcp23s08\_probe\_one(data->mcp[addr], &spi->dev, spi,

0x40 | (addr << 1), type, pdata,

addr);

The driver wants to probe devices, it loop address from zero to array size - let’s say 8.

For address 0 we should put spi\_present\_mask 1

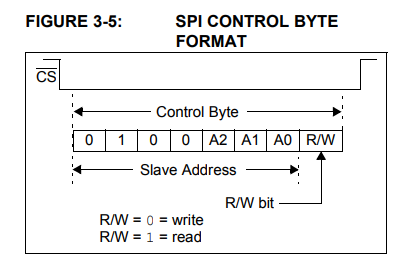
For address 1 we should put spi\_present\_mask 2

For address 2 we should put spi\_present\_mask 4

So for three devices the present mask should be 7.

The device already add the 0x40 to the address.

The 0x40 is the MSB part of the chip address.



The gpio is added in the probe function

Board Support

-------------

For external GPIO controllers -- such as I2C or SPI expanders, ASICs, multi

function devices, FPGAs or CPLDs -- most often board-specific code handles

registering controller devices and ensures that their drivers know what GPIO

numbers to use with gpiochip\_add(). Their numbers often start right after

platform-specific GPIOs.

For example, board setup code could create structures identifying the range

of GPIOs that chip will expose, and passes them to each GPIO expander chip

using platform\_data. Then the chip driver's probe() routine could pass that

data to gpiochip\_add()

status = gpiochip\_add(&mcp->chip);

if (status < 0)

goto fail;

the gpio\_chip can be obeserved here:

<https://elixir.bootlin.com/linux/v4.0/source/include/linux/gpio/driver.h>

struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) {

const char \*[**label**](https://elixir.bootlin.com/linux/v4.0/ident/label);

struct device \*dev;

struct [**module**](https://elixir.bootlin.com/linux/v4.0/ident/module) \*[**owner**](https://elixir.bootlin.com/linux/v4.0/ident/owner);

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.0/ident/list_head) list;

int (\*[**request**](https://elixir.bootlin.com/linux/v4.0/ident/request))(struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) \*chip,

unsigned offset);

void (\*[**free**](https://elixir.bootlin.com/linux/v4.0/ident/free))(struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) \*chip,

unsigned offset);

int (\*get\_direction)(struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) \*chip,

unsigned offset);

int (\***[direction\_input](https://elixir.bootlin.com/linux/v4.0/ident/direction_input)**)(struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) \*chip,

unsigned offset);

int (\***[direction\_output](https://elixir.bootlin.com/linux/v4.0/ident/direction_output)**)(struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) \*chip,

unsigned offset, int value);

int (\*[**get**](https://elixir.bootlin.com/linux/v4.0/ident/get))(struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) \*chip,

unsigned offset);

void (\*[**set**](https://elixir.bootlin.com/linux/v4.0/ident/set))(struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) \*chip,

unsigned offset, int value);

void (\*set\_multiple)(struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) \*chip,

unsigned long \*mask,

unsigned long \*[**bits**](https://elixir.bootlin.com/linux/v4.0/ident/bits));

int (\***[set\_debounce](https://elixir.bootlin.com/linux/v4.0/ident/set_debounce)**)(struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) \*chip,

unsigned offset,

unsigned [**debounce**](https://elixir.bootlin.com/linux/v4.0/ident/debounce));

int (\***[to\_irq](https://elixir.bootlin.com/linux/v4.0/ident/to_irq)**)(struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) \*chip,

unsigned offset);

void (\***[dbg\_show](https://elixir.bootlin.com/linux/v4.0/ident/dbg_show)**)(struct [**seq\_file**](https://elixir.bootlin.com/linux/v4.0/ident/seq_file) \*s,

struct [**gpio\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_chip) \*chip);

int base;

u16 [**ngpio**](https://elixir.bootlin.com/linux/v4.0/ident/ngpio);

struct [**gpio\_desc**](https://elixir.bootlin.com/linux/v4.0/ident/gpio_desc) \*desc;

const char \*const \*[**names**](https://elixir.bootlin.com/linux/v4.0/ident/names);

bool can\_sleep;

bool irq\_not\_threaded;

bool exported;

#ifdef CONFIG\_GPIOLIB\_IRQCHIP

*/\**

*\* With CONFIG\_GPIOLIB\_IRQCHIP we get an irqchip inside the gpiolib*

*\* to handle IRQs for most practical cases.*

*\*/*

struct [**irq\_chip**](https://elixir.bootlin.com/linux/v4.0/ident/irq_chip) \***[irqchip](https://elixir.bootlin.com/linux/v4.0/ident/irqchip)**;

struct [**irq\_domain**](https://elixir.bootlin.com/linux/v4.0/ident/irq_domain) \*irqdomain;

unsigned int [**irq\_base**](https://elixir.bootlin.com/linux/v4.0/ident/irq_base);

[**irq\_flow\_handler\_t**](https://elixir.bootlin.com/linux/v4.0/ident/irq_flow_handler_t) [**irq\_handler**](https://elixir.bootlin.com/linux/v4.0/ident/irq_handler);

unsigned int irq\_default\_type;

#endif

The mcp23xx device creates gpio inside the kernel for us.

Using the gpio chip it point to function like direction set and more:

/sys/class/gpio/gpiochipX/

When the host will issue a direction call from sysfs or c function , a set or clear value, the call will be directed to the hookup of the gpio\_chip structure”

The question is which number the gpio is given?

From the book:

base identifies the first GPIO number handled by this chip; or, if negative during registration, the kernel will automatically (dynamically) assign one. ngpio is the number of GPIOs this controller provides, starts from base, to (base + ngpio - 1).

Base and ngpio are member of gpio\_chip as mentioed.

The driver uses the strcture name mcp23s08 and not mcp23XXX

Probably because it was already written for this chip and was extends later for more chip in the category.

It does support the s17 ( 16 gpio’s)

struct mcp23s08 {

u8 addr;

bool irq\_active\_high;

u16 cache[11];

u16 irq\_rise;

u16 irq\_fall;

int irq;

bool irq\_controller;

/\* lock protects the cached values \*/

struct mutex lock;

struct mutex irq\_lock;

struct irq\_domain \*irq\_domain;

**struct gpio\_chip chip;**

const struct mcp23s08\_ops \*ops;

void \*data; /\* ops specific data \*/

};

During the probe the structure is being initialized:

case MCP\_TYPE\_S17:

mcp->ops = &mcp23s17\_ops;

mcp->chip.ngpio = 16;

mcp->chip.label = "mcp23s17";

break;

the driver support read,write and read\_regs operations:

static const struct mcp23s08\_ops mcp23s17\_ops = {

.read = mcp23s17\_read,

.write = mcp23s17\_write,

.read\_regs = mcp23s17\_read\_regs,

};

Which are unique to the 23s17device.

The direction functions , for input and output

Are common to all chip

mcp->chip.direction\_input = mcp23s08\_direction\_input;

mcp->chip.direction\_output = mcp23s08\_direction\_output;

read and write

As we see, the driver implement the protocol of the read and write device via SPI.

static int mcp23s17\_read(struct mcp23s08 \*mcp, unsigned reg)

{

u8 tx[2], rx[2];

int status;

tx[0] = mcp->addr | 0x01;

tx[1] = reg << 1;

status = spi\_write\_then\_read(mcp->data, tx, sizeof(tx), rx, sizeof(rx));

return (status < 0) ? status : (rx[0] | (rx[1] << 8));

}

static int mcp23s17\_write(struct mcp23s08 \*mcp, unsigned reg, unsigned val)

{

u8 tx[4];

tx[0] = mcp->addr;

tx[1] = reg << 1;

tx[2] = val;

tx[3] = val >> 8;

return spi\_write\_then\_read(mcp->data, tx, sizeof(tx), NULL, 0);

}

This is the place we can put a printk to debug in case we have a problem.

The gpio base number:

In the kernel we can see that gpio base is initialize to -1

pdata->base = -1;

It means that the kernel will assign the base gpio and this could lead a problem in the software because it could change.

We need it constant.

We can put our own number to force the base gpio number.

SPI.

The chip uses SPI.

So how do we know that the spi is configure correct for the device to work:

We need to make sure the SPI driver is configured.

The host can use it , the gpio-controller can use everyone can use it if thery are connected to it.

The gpio\_chip , gpio\_controller, gpio\_expander use the SPI driver.

its read and write using the following functions: **spi\_write\_then\_read**

static int mcp23s17\_write(struct mcp23s08 \*mcp, unsigned reg, unsigned val)

{

u8 tx[4];

tx[0] = mcp->addr;

tx[1] = reg << 1;

tx[2] = val;

tx[3] = val >> 8;

return **spi\_write\_then\_read**(mcp->data, tx, sizeof(tx), NULL, 0);

}

**Chip Select**

The chip select is being configure from the Spi device and not from the gpio-controller driver

cs-gpios is a property of the Spi driver!

fsl,spi-num-chipselects = <4>;

cs-gpios = <

&gpio5 17 GPIO\_ACTIVE\_LOW

&gpio3 19 GPIO\_ACTIVE\_LOW

&gpio3 24 GPIO\_ACTIVE\_LOW

&gpio3 25 GPIO\_ACTIVE\_LOW

>;

In the register spi master device:

static int of\_spi\_register\_master(struct spi\_master \*master)

{

int nb, i, \*cs;

struct device\_node \*np = master->dev.of\_node;

if (!np)

return 0;

nb = of\_gpio\_named\_count(np, **"cs-gpios**");

master->num\_chipselect = max\_t(int, nb, master->num\_chipselect);

But

It look like the kernel is using the cs-gpio only for SPI access to the spi driver

spi\_transfer\_one\_message and not to the **spi\_write\_then\_read.**

The spi\_write\_then\_read is a utility function that transfer spi\_device structure by it self

struct spi\_device {

struct device dev;

struct spi\_master \*master;

u32 max\_speed\_hz;

u8 chip\_select;

u8 bits\_per\_word;

u16 mode;

int irq;

void \*controller\_state;

void \*controller\_data;

char modalias[SPI\_NAME\_SIZE];

int cs\_gpio; /\* chip select gpio \*/

So currently I don’t know if the chip select will be selected by

1. The gpio controller via the spi\_device structure
2. The cs will be selected by the spi driver because it has the device tree chip selected
3. The host will need to issue the chip select prior to write and read operation

I did not saw in the gpio expander driver where a reference to chip select

So it could be

1. That it uses the first cs In the list
2. The gpio exapnder does not support multiplex SPI chip select and only one