

katix.org

Kate Alhola
kate@katix.org

Katix Embedded Linux SPI subsystem

Driver subsystem description
03/10/04

1

GPL
Public

Katix Embedded Linux SPI Subsystem description

Change history

Version	Date	Author	Comments
0.0.1	03/10/04	Kate Alhola	initial draft

katix.org

Kate Alhola
kate@katix.org

Katix Embedded Linux SPI subsystem
Driver subsystem description
03/10/04

2
GPL
Public

Table of Contents

Introduction.....3
 Terms, acronyms and abbreviations.....3

Overview

Introduction

Linux SPI subsystem is designed to making generic programming framework for various peripherals with SPI interface. It is derived from I2C subsystem with lot of SPI specific additions. Design is divided in four layers, cpu architecture independent target driver and spi subsystem, spi controller device specific spi algorithm layers and board specific spi adapter layer. Above SPI subsystem layer there is various target drivers that interfaces to upper level subsystems depending of their type. As example SPI-MMC driver interfaces to MMC subsystem and SPI-touch screen interfaces to HID subsystem.

SPI subsystem contains derived codes from Original unfinished SPI subsystem from **Hamey Hicks**, I2C subsystem from **Simon G. Vogl**, **Denx** PPC linux from **Wolfgang Denk**

License

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA. This program is licensed under the terms of the GNU General Public License version 2. This program is licensed "as is" without any warranty of any kind, whether express or implied.

Terms, acronyms and abbreviations

Term	Explanation
SPI	Serial Peripheral Interface
I2C	Inter IC bus
PSC	MPC5200 Programmable Serial Controller
MMC	Multimedia Card
HID	Human Interface Device

katix.org

Katix Embedded Linux SPI subsystem 4

Kate Alhola

Driver subsystem description

GPL

kate@katix.org

03/10/04

Public

Term	Explanation
SPI Client	SPI target driver utilizing SPI subsystem
SPI Driver	SPI interface driver
SPI Adapter	SPI adapter that contains SPI Interface and chip select logic
SPI target	SPI connected peripheral like A/D converter

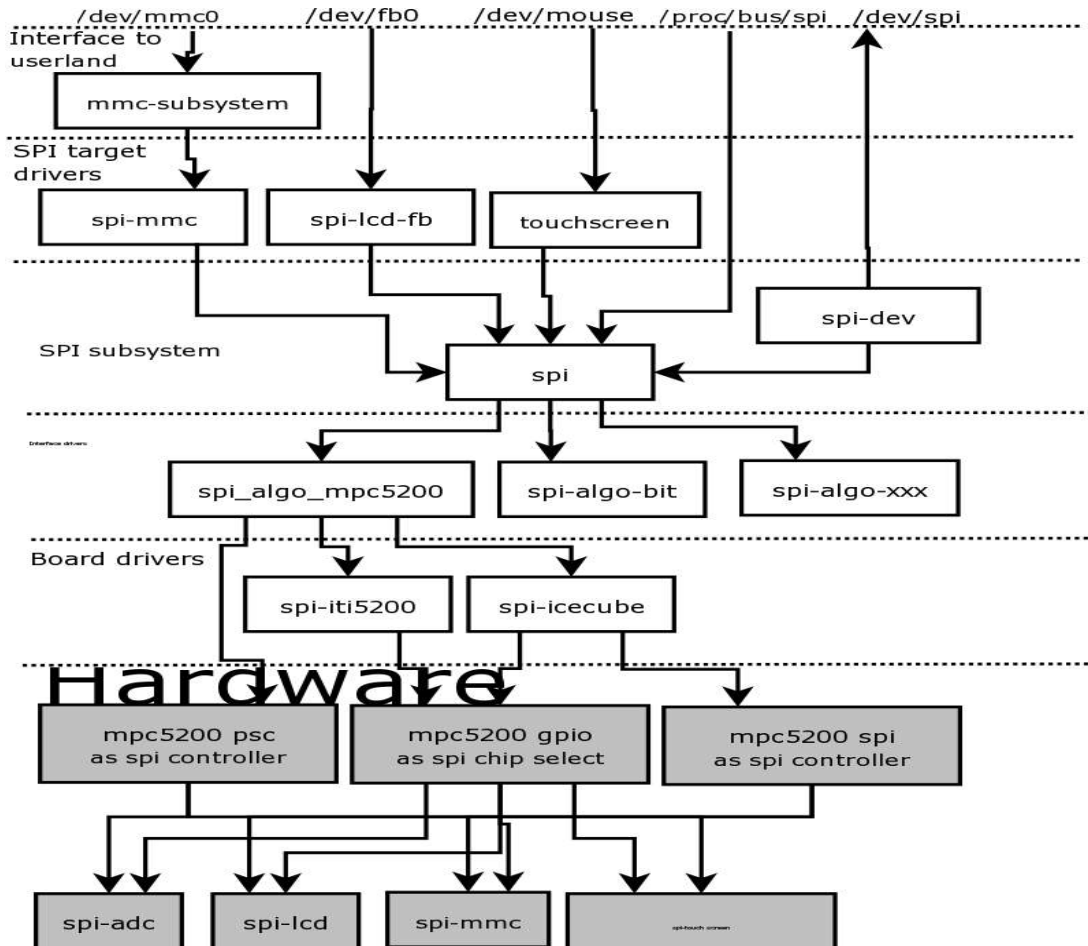
References

Ref ID, Document Name, Author, Version	Location / Weblink
----------------------------------------	--------------------

Architecture

Subsystem level architecture

SPI Subsystem diagram



Interfaces

Interfaces used

LinuxKernel API

Interfaces provided

/proc filesystem

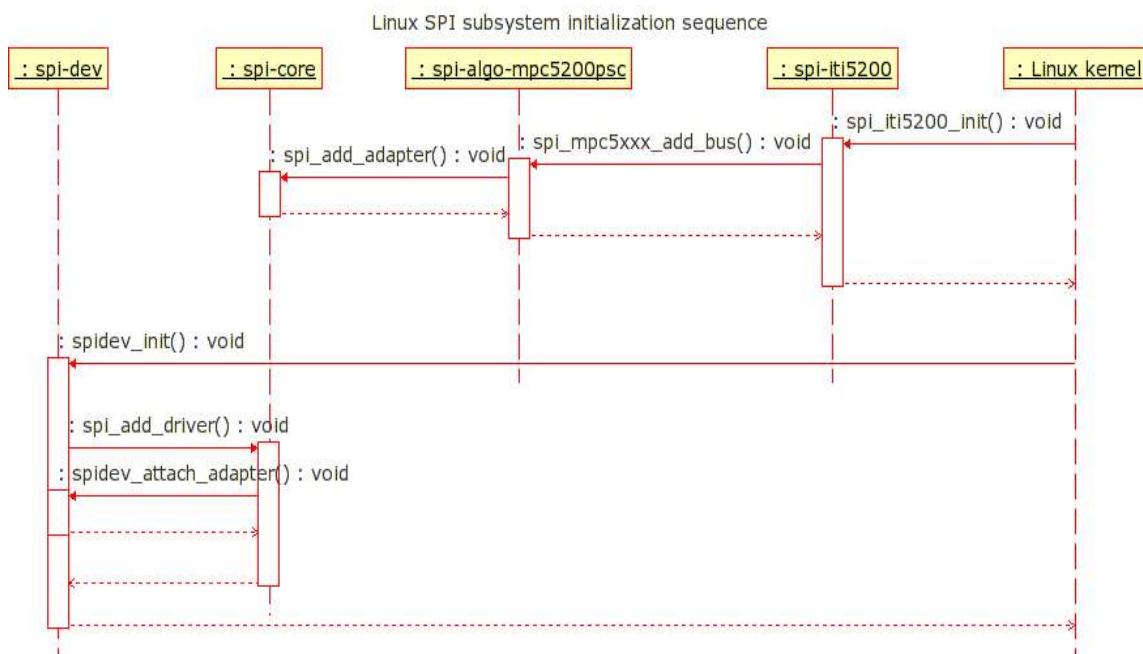
/dev/spi

SPI subsystem API

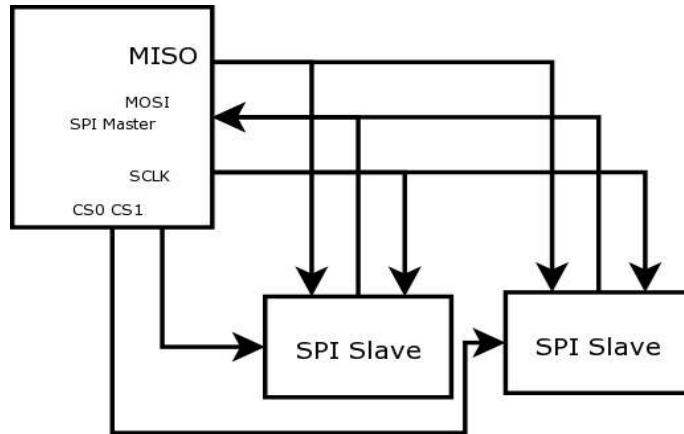
Dependencies

Scenarios

Linux SPI subsystem initialization sequence



Principles of operation



SPI (synchronous Peripheral Interface) is bi-directional full duplex data transfer with separate bit and frame synchronization signals.

Data is transferred via two unidirectional data lines called MISO (Master In Slave Out) and MOSI (Master Out Slave In). Mosi is sometimes called in master SDO (Serial Data Out) and MISO called as SDI (Serial DataIn). MOSI is always driven by master and is always input in slaves. MISO is always input in master and it is three state signal in slaves. Only slave addressed with its representative CS signal asserted will drive MISO . SPI transfer has always read (MISO) and write (MOSI) signals an always operate as simultaneous full duplex data transfer. I case we have simple Write only devices like simple digital/analog converters (DAC) that have only MOSI signal, the master still samples MISO signal and reads meaningless data that should be discarded. Similary with input only devices, master still transfers data out from MOSI signal but it is ignored or not connected to slave device.

Bit clocking and synchronization is provided with SCLK signal. SCLK is NOT free running clock, it is gated clock and it is gated with data transfer. SCLK is always driven by master.

Transferred frames are synchronized with CS (Chip Select signals). CS lines are always driven by master. Normally there is least one but not always just one separate CS signal per target device. Chip select signals does double function. It works as target device select issued by master and also works as frame synchronization signal. I case that master needs to send successive transfers to same slave, it will indicate it releasing and asserting CS line between frames.

Chip Select handling

SPI greatly differs from other peripheral busses like I2C, USB, Single wire etc by target selection logic. Other busses use target address transferred at beginning of the frame.SPI uses dedicated Chip Select lines that functions as both target select and frame synchronization lines. Every target has it's dedicated unique CS line that indicates that frame is intended for it and where frame begins and ends.

Even different boards utilizing same processor (or SPI interface) circuit, the schip select lines may be implemented completely different way. For this reason the actual SPI interface driver (spi-algo-xxxx) is separated from board interface drivr like spi-

iti5200). SPI target drivers (spi clients) are completely independent to lower SPI subsystem layer implementation as they are in I2C subsystem. Difference to I2C is that every spi client must be able to attach it's target and it's dedicated chip select line. As we know, same type target (like tsc2301 or MMC card) may be connected in different boards to completely different chip select lines. Also, in SPI there is no hardware method to identify target type.

To met these design criterias following chip select handling is implemented to Linux SPI subsystem.

Every SPI board driver has chip select assigment table **struct spi_cs** that contains names and assigned numbers of CS lines. SPI subsystem spi-core offers functions **spi_get_named_cs** and **spi_get_cs** for clients to locate target chips and thei chip select lines.

Chip select behavior

There is also a lot of difference chip select behavior betwen targets. Simples way is basic Spi method like used in following tsc2301 example. In basic method there is one low active chip select that is asserted in beginning of frame and is released in end of frame. More complicated handling is in following MMC and LCD examples. With MMC there is before actual frame, initial clock pulses without CS asserted and then command, wait and data phases all without releasing CS between them. MMC need that first dummy frame is given without asserting CS and then command frame is given without releasing chip select after it. Then undetermined number of status read frames when CS remains asserted and finally data transfer phase. The CS is released after all these phases.

LCD (for displaytech and similar LCD displays) has similar behavior that it also has multiple phases with CS asserted but differs from MMC that it has two chip select lines, one is actual chip select and other is command/data select.

Application examples

Simple user level program with direct access to SPI target via /dev/spi interface.

In this example we first form spi message **struct spi_msg** that contains data to be transferred and read results, target chip select address and transfer mode read (**SPI_M_RD**) and/or write (**SPI_M_WR**). Spi_msg can contain multiple messages with all own chip select signaling, data buffers, read/write mode etc. As a special case devices that have non-standard chip-select behavior can use zero-length messages just for handling chip select signal. As example some output only shift registers does not have normal chip select but instead have one that should be pulsed after data transfer. In this example we are using normal SPI behavior. We send data out tcad contains command to target device and read it's response.

struct spi_rdwr_ioctl_data is used to pass **spi_msg** as a parameter to spi ioctl command.

In beginning of actual code we open **/dev/spi0** device to get access from userland program to kernel mode spi subsystem. After we have opened spi device, we can use ioctl system call to pass commands to spi subsystem. Normally when we like to use simultaneous read/write or multiple spi messages or other spi special features we need to use most flexible way, ioctl. In case we need to do just simple read without writing or write without read we can use normal read and write system calls.

```
#include <linux/spi/spi.h>

#define DEV_SPI "/dev/spi0" /* Create device with 'mknod /dev/spi0 c 228 0' */
#define LEN 10

/* Trasfer one message via SPI */
int spi_xfer(char addr, char offset, char *buf, int len)
{
    struct spi_msg msgs[] = {
        { addr: addr, flags: SPI_M_RD| SPI_M_WR, len: len, buf: buf }
    };
    struct spi_rdwr_ioctl_data data = { msgs: msgs, nmsgs: 1 };
    int fd;

    if ((fd = open(DEV_SPI, O_RDWR)) < 0) {
        perror(DEV_SPI " open");
        return -1;
    }
    if (ioctl(fd, SPI_RDWR, &data) < 0) {
        perror(DEV_SPI " ioctl");
        return -1;
    }
    close(fd);
    return 0;
}
```

TSC2301 example

Following example demonstrates how to access **iti5200** board **TSC2301** system codec ADC from user space process with /dev/spi interface. The tsc2301read and tsc2301write functions are just made to easier access tsc2301 registers. In this example all tsc2301 registers are accessed using 32 bit spi frame containing 16 bit command and 16 bit data.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R/W*		0	0	pg 1	pg 0	addr 5	addr 4	addr 3	addr 2	addr 1	addr 0	x	x	x	x	x

The tsc2301read and tsc2301write functions build 32 bit spi frame from parameters and send it using spi_xfer function to spi device. TSC2301read function sends 16 bit data part of frame as zero and then in return it returns value read from spi.

Reader should notice that no information about is this command device read or write

is given to spi_xfer function. SPI xfer passes all frames as read/write attributes to spi driver. If operation is write, the returned frame is just not used. In read the tsc2301 chip just uses first bit of command to see that operation is read and it should pass output data after command word and discards all data after read command. The dummy writedata is still mandatory to give to the device because in spi reading is only possible when bit clock pulses are given for this dummy data.

```
void tsc2301write(int addr,int pg,int reg,unsigned short val)
{
    unsigned short buf[6]={0,0,0,0,0,0};
    buf[0]=(pg<<11) | (reg << 5);
    buf[1]=val;
    hexdump((unsigned char*)buf,4);
    spi_xfer(0,(unsigned char *)buf,4);
}

int tsc2301read(int addr,int pg,int reg)
{
    unsigned short buf[6]={0,0,0,0,0,0};
    buf[0]=0x8000 | (pg<<11) | (reg << 5);
    spi_xfer(0,(unsigned char *)buf,4);
    hexdump((unsigned char *)buf,4);
    return(buf[1]);
}

#define VREF 1.2

int main(int argc, char* argv[])
{
    unsigned short buf[256]={0x8000,0x2f30,0,0,0};
    float bat1,bat2,aux1,aux2;

    tsc2301write(0,1,3,0x1000); /* bank1 reg 3 Ref int 1.2v */
    tsc2301write(0,1,0,0x2f30); /* bank1 reg 0,scan bat1,bat2,aux1,aux2,12bit,1MHz int
clock */
    bat1=VREF*tsc2301read(0,0,5)/4096*20*4; /* bat 1 - voltage divider 1/80*/
    bat2=VREF*tsc2301read(0,0,6)/4096; /* bat 2 */
    aux1=VREF*tsc2301read(0,0,7)/4096; /* aux 1 */
    aux2=VREF*tsc2301read(0,0,8)/4096; /* aux 2 */

    printf("bat1=%f bat2=%f aux1=%f aux2=%f\n",bat1,bat2,aux1,aux2);

    return 0;
}
```

Running our example . Notice that only bat1 is connected to measure board input voltage, other ADC inputs are floating.

```
bash-2.05b# /tmp/tsc2301
08 60 10 00 .`.
08 00 2f 30 ../0
ff ff 02 10 ...
ff ff 00 05 ...
ff ff 04 43 ...C
ff ff 05 9f ...
```

```
bat1=12.375000 bat2=0.001465 aux1=0.319629 aux2=0.421582
```

Frame list example

To make more optimized application there is possible to do everything in that was in previous application done with multiple SPI ioctl calls with one SPI ioctl call with multiple message list. Using single call we get only one call overhead but for result may not be as easy to read. Next example demonstrates how to do it.

In multiple message list extensive care should be taken for CS operations between frames. If CS is not negated between frames, all frames are typically considered by devices same as one concatenated frame. If they are liked to be treated as separate frames they should be marked so that CS is negated between frames.

In some rare cases special chip select behavior is needed like with SPI LCD display where other CS is used as Command/Data selector and so needs to be toggled between command and data part when actual chip select is kept asserted. Other case is some ADC chips and shift registers that does have load strobe instead of chip select.

Internal design

SPI subsystem is divided to following main modules:

1- SPI-Core

SPI core is responsible as brooker and device interface functions between clients, drivers and adapters. When new interface is added to system,

Functional description

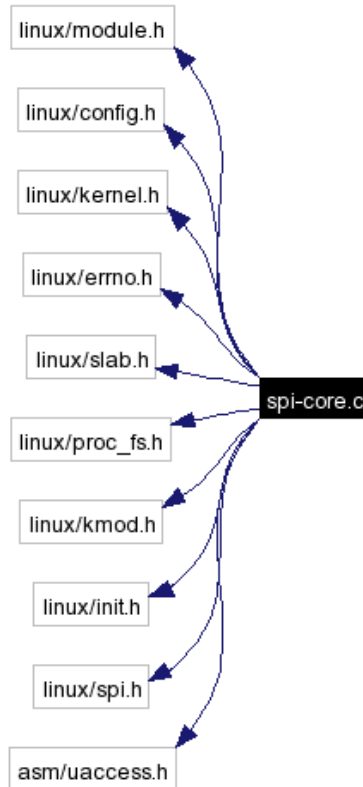
spi-core.c File Reference

SPI (Serial Periphereal Interface) subsystem core module. This module is responsible to registering and controlling spi algorithm drivers, adapters and target drivers. [More...](#)

```
#include <linux/module.h>
#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/slab.h>
#include <linux/proc_fs.h>
#include <linux/kmod.h>
#include <linux/init.h>
#include <linux/spi/spi.h>
```

```
#include <asm/uaccess.h>
```

Include dependency graph for spi-core.c:



Functions

```

int spi\_add\_adapter (struct spi_adapter *adap)
    Make the adapter available for use by clients using name adap->name. The adap->adapters list
    is initialised by this function.
int spi\_del\_adapter (struct spi_adapter *adap)
    Remove an adapter from the list of available SPI Bus adapters.
spi_adapter
    * spi\_get\_adapter (const char *name)
        Obtain a spi_adapter structure for the specified adapter. If the adapter is not currently load,
        then load it. The adapter will be locked in core until all references are released via
        spi_put_adapter.
void spi\_put\_adapter (struct spi_adapter *adap)
int spi\_add\_driver (struct spi_driver *driver)
int spi\_del\_driver (struct spi_driver *driver)
spi_driver * spi\_get\_driver (const char *name)
void spi\_put\_driver (struct spi_driver *drv)
int spi\_attach\_client (struct spi_client *client, const char *adap, const char *drv)
int spi\_detach\_client (struct spi_client *client)
int spi\_transfer (struct spi_adapter *adap, struct spi_msg msgs[], int num)
  
```

```
int spi_write (struct spi_client *client, int addr, const char *buf, int len)
int spi_read (struct spi_client *client, int addr, char *buf, int len)
int spi_adapter_id (struct spi_adapter *adap)
EXPORT_SYMBOL (spi_add_adapter)
EXPORT_SYMBOL (spi_del_adapter)
EXPORT_SYMBOL (spi_get_adapter)
EXPORT_SYMBOL (spi_put_adapter)
EXPORT_SYMBOL (spi_add_driver)
EXPORT_SYMBOL (spi_del_driver)
EXPORT_SYMBOL (spi_get_driver)
EXPORT_SYMBOL (spi_put_driver)
EXPORT_SYMBOL (spi_attach_client)
EXPORT_SYMBOL (spi_detach_client)
EXPORT_SYMBOL (spi_transfer)
EXPORT_SYMBOL (spi_write)
EXPORT_SYMBOL (spi_read)
MODULE_LICENSE ("GPL")
MODULE_AUTHOR ("Kate Alhola kate(at) katix.org")
MODULE_DESCRIPTION ("SPI-Bus main module")
```

Detailed Description

SPI (Serial Periphereal Interface) subsystem core module. This module is responsible to registering and controlling spi algorithm drivers, adapters and target drivers.

Function Documentation

```
int spi_adapter_id( struct spi_adapter adap )
*
```

spi_adapter_id This call returns a unique low identifier for each registered adapter

Parameters:

adap: spi_adapter

Returns:

unique id or -1 if the adapter was not registered.

```
int spi_add_adapter( struct spi_adapter adap )
*
```

Make the adapter available for use by clients using name `adap->name`. The `adap->adapters` list is initialised by this function.

`spi_add_adapter` - register a new SPI bus adapter

Parameters:

adap: `spi_adapter` structure for the registering adapter

Returns:

0;

```
int spi_add_driver( struct spi_driver driver )
```

`spi_add_driver` - register a new SPI device driver

Parameters:

driver - driver structure to make available

Make the driver available for use by clients using name `driver->name`. The `driver->drivers` list is initialised by this function.

Returns 0;

```
int spi_attach_client( struct spi_client * client,
```

```
const char * adap,
```

```
const char * drv
```

```
)
```

`spi_attach_client` - attach a client to an adapter and driver

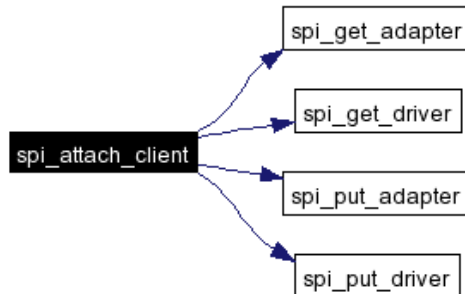
Parameters:

client: client structure to attach
adap: adapter (module) name
drv: driver (module) name

Attempt to attach a client (a user of a device driver) to a particular driver and adapter. If the specified driver or adapter aren't registered, `request_module` is used to load the relevant modules.

Returns 0 on success, or negative error code.

Here is the call graph for this function:



```
int spi_del_adapter( struct spi_adapter adap )
*
```

Remove an adapter from the list of available SPI Bus adapters.

spi_del_adapter - unregister a SPI bus adapter

Parameters:

adap: spi_adapter structure to unregister

Returns:

0;

```
int spi_del_driver( struct spi_driver driver )
*
```

spi_del_driver - unregister a SPI device driver

Parameters:

driver: driver to remove

Remove an driver from the list of available SPI Bus device drivers.

Returns 0;

```
int spi_detach_client( struct spi_client client )
*
```

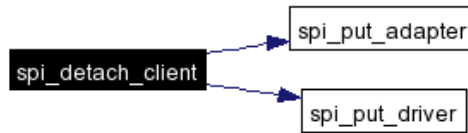
spi_detach_client - detach a client from an adapter and driver

Parameters:

client: client structure to detach

Detach the client from the adapter and driver.

Here is the call graph for this function:



```

struct          spi_adapter*( const  charname )
spi_get_adapter
                *
```

Obtain a spi_adapter structure for the specified adapter. If the adapter is not currently load, then load it. The adapter will be locked in core until all references are released via spi_put_adapter.

spi_get_adapter - get a reference to an adapter

Parameters:

name: driver name

Returns:

pointer to spi_adapter struct or NULL if no adapter found

```

struct          spi_driver*( const  charname )
spi_get_driver
                *
```

spi_get_driver - get a reference to a driver

Parameters:

name: driver name

Obtain a spi_driver structure for the specified driver. If the driver is not currently load, then load it. The driver will be locked in core until all references are released via spi_put_driver.

```

void spi_put_adapter( struct  spi_adapteradap )
                    *
```

spi_put_adapter - release a reference to an adapter

Parameters:

adap: driver to release reference

Indicate to the SPI core that you no longer require the adapter reference. The adapter module may be unloaded when there are no references to its data structure.

You must not use the reference after calling this function.

```

void spi_put_driver ( struct  spi_driverdrv )
                    *
```

spi_put_driver - release a reference to a driver

Parameters:

drv: driver to release reference

Indicate to the SPI core that you no longer require the driver reference. The driver module may be unloaded when there are no references to its data structure.

You must not use the reference after calling this function.

```
int spi_read( struct spi_client
             *      client,
             int      addr,
             char *   buf,
             int      len
             )
```

spi_read - receive data from a device on an SPI bus

Parameters:

- client*: registered client structure
- addr*: SPI bus address
- buf*: buffer for bytes to receive
- len*: number of bytes to receive

Receive len bytes from device address addr on the SPI bus described by client to a buffer pointed to by buf.

Returns the number of bytes transferred, or negative error code.

Here is the call graph for this function:



```
int spi_transfer( struct spi_adapter
                 *      adap,
                 struct spi_msg msgs[],
                 int      num
                 )
```

spi_transfer - transfer information on an SPI bus

Parameters:

- adap*: adapter structure to perform transfer on
- msgs*: array of spi_msg structures describing transfer
- num*: number of spi_msg structures

Transfer the specified messages to/from a device on the SPI bus.

Returns number of messages successfully transferred, otherwise negative error code.

```
int spi_write( struct spi_client
              *      client,
              int      addr,
              const char * buf,
              int      len
              )
```

spi_write - send data to a device on an SPI bus

Parameters:

- client*: registered client structure
- addr*: SPI bus address
- buf*: buffer for bytes to send
- len*: number of bytes to send

Send len bytes pointed to by buf to device address addr on the SPI bus described by client.

Returns:

the number of bytes transferred, or negative error code.

Here is the call graph for this function:



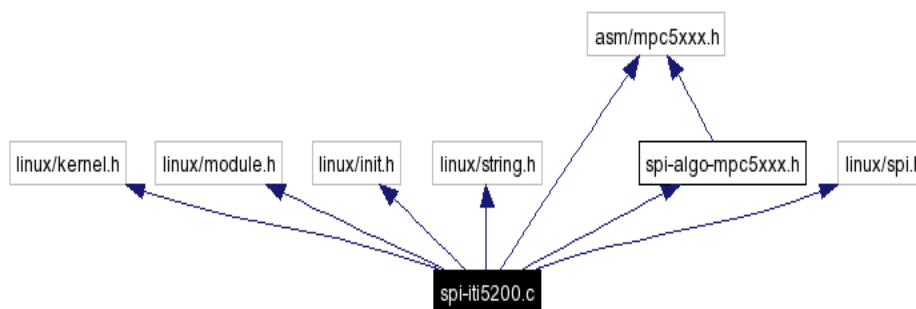
spi-iti5200.c File Reference

SPI adapter driver for iti5200 board with MPC5xxx. [More...](#)

```

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/string.h>
#include <asm/mpc5xxx.h>
#include <linux/spi/spi.h>
#include "spi-algo-mpc5xxx.h"
  
```

Include dependency graph for spi-iti5200.c:



Defines

```
#define MPC5xxx_SPI_ENABLE 1 /* Disable */
#define GPIO_PSC3_4 0x1000
#define GPIO_SINT0 1
#define GPIO_SINT1 2
```

Functions

```
int set_spi_iti5200_cs (struct spi_adapter *, int id, int val)
module_init (spi_iti5200_init)
module_exit (spi_iti5200_exit)
MODULE_LICENSE ("GPL")
MODULE_AUTHOR ("Kate Alhola kate(at) katix.org")
MODULE_DESCRIPTION ("iti5200 SPI adapter")
```

Variables

```
mpc5xxx_gpioitipower5200_gpio = (struct mpc5xxx_gpio *)
    * MPC5xxx_GPIO
mpc5xxx_gpwitipower5200_gpw = (struct mpc5xxx_gpw *)
    * MPC5xxx_GPW
```

Detailed Description

SPI adapter driver for iti5200 board with MPC5xxx.

Function Documentation

```
int
set_spi_iti5200_cs ( struct spi_adapter adap,
                   *
                   int id,
                   int val
                   )
```

set_spi_iti5200_cs set adapter spi cs line

Parameters:

adap spi_adapter structure
id chip select line id
val chip select line value

Returns:

0

katix.org

Kate Alhola
kate@katix.org

Katix Embedded Linux SPI subsystem

Driver subsystem description
03/10/04

20

GPL
Public

UML model