

Re: [RFC][PATCH] SPI subsystem

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2005-09/5162.html>

From: Mark Underwood (*basicmark_at_yahoo.com*)

Date: 09/16/05

Date: Fri, 16 Sep 2005 18:55:36 +0100 (BST)

To: David Brownell <david-b@pacbell.net>, linux-kernel@vger.kernel.org

----- David Brownell <david-b@pacbell.net> wrote:

> > *ASCII art time :)
>
> ASCII also works for English and C, but this is good
> too. ;)
>
>
>
> > *SPI device drivers| Core | Adapter driver
>
> Or in my terminology, with no core/midlayer
> expectation:
>
> *SPI Master | SPI Master
> Protocol Driver | Controller Driver
>
> > ----- msg ||
> > | EEPROM |-----> |- |
> > ----- |||
> > ----- msg || ----- | msg -----
> > | ETHERNET |-----> |-->|Queue|->|----->|xfer|
> > ----- || ----- | -----
> > ----- msg || ^ ||
> > | FLASH |-----> |- -----|<----- Next/done
> > ----- ||
>
> > *And the workqueue usage you're describing (to manage
> that queue)
> is what I'd call just one of many implementation
> strategies; one
> that wouldn't be universally appropriate.
>
> > You don't NEED a separate "core" to hold the queue;
> the driver
> can just as easily manage a list_head itself.****

- >
- > – A PIO driver running with IRQs blocked wouldn't
- > need a queue.
- > (And that might be fine for low volume usage,
- > since it might
- > take all of an hour to write and debug.)
- >
- > – A DMA driver might build the queue out of DMA
- > descriptors, so
- > that the hardware would process many
- > transactions back-to-back
- > without needing intermediate attention from the
- > CPU. (Great
- > for high volume usage, but likely takes more
- > than an hour.)
- >
- > You don't NEED to allocate a task ("workqueue") to
- > handle the
- > movement of requests from the queue to the hardware.
- >
- > – Drivers are often written so that the "next"
- > transaction is
- > started from an IRQ handler ... it's a lot less
- > I/O latency
- > than if you need the workqueue's task to
- > schedule before that
- > transaction can start, and better throughput
- > too.
- >
- > That's three more implementation strategies ... ones
- > that don't need
- > or want to have a "workqueue" task, and can easily
- > manage a list_head.
- > But you want them all to not just have a workqueue,
- > but not be able
- > to work unless they interact with it!! Why?
- >

Thank you! I have seen the light. Yes, I can see now that the workqueue should really be in my adapter driver not the core. The only thing about having a work queue in the core is that it means adapter drivers don't have to do the sleep for blocking transfers, but I guess that's just laziness creeping in ;).

Thinking about it, for blocking transfers the core could call the adapters transfer routine and then start a wait on completion. When the message has been sent the adapter would finish the completion and the call to the core would then return (I think this is how the mmc core layer does it). How do you feel about

that suggestion?

How would you feel about having a list head for messages in the adapter structure? I think every adapter driver would at least need this.

>
>
> > *If you don't have a queue of some sort how do you
> > handle the fact that one or more devices will want
> > to
> > send asynchronous messages through one adapter
> > while
> > that adapter is still busy transferring a previous
> > message?*
>
> *You seem to have been replying to someone else's
> posts! The
> examples I gave `_included_` queues. Queues that
> advanced using
> IRQs, and didn't need to schedule a workqueue's task
> before
> starting another transaction.*
>

Sorry I must have had a blind spot :(.

>
> > *OK. But you also need to see what else is in the
> > cs
> > table, namely the default level of the cs's. The
> > issue
> > which we have to solve is that all the cs's have
> > to be
> > put into their non-active state before any
> > transfer
> > can be done.*
>
> *If you observed the code I sent by, you'll observe
> comments about that
> issue in the declaration of both "spi_device" and
> "spi_board_info".*
>
> *There are other protocol tweaks to be dealt with
> over time too. As one
> person commented off-line, there are multiple
> protocols to fit into this
> API (SPI, MicroWire, SSI, SSP, and more) that differ
> in only minor ways.
> The controller driver may need to know about some of
> those tweaks in*

Linux-Kernel: Re: [RFC][PATCH] SPI subsystem

> *order to talk properly with a given chip.*

>

Yes, at some time I was thinking about how you might create a serial bus subsystem that would even incorporate I2C :O.

>

> > *If devices are added into a running SPI*

> > *adapter (I mean registration of devices not*

> *physical*

> > *plugging) then how do you know what their idle*

> *state*

> > *is at the beginning (when the first SPI device*

> *does a*

> > *transfer?*

>

> *Until someone has to hook up hardware that acts*

> *atypically, it's the*

> *normal SPI convention: chipselect is "active low".*

> *When someone*

> *needs that, I've already identified where to record*

> *"active high".*

>

>

> > *To me the only solution seems to be to pass*

> > *the idle state of all the devices that will be on*

> *that*

> > *device (be they hardwire, plugged or what ever)*

> *when*

> > *then adapter gets registered which is why I put it*

> *in*

> > *as part of platform data.*

>

> *Much like I did with "spi_board_info", but on a*

> *per-chip basis. It's*

> *possible to hook an "active low" chip to one*

> *chipselect while another*

> *uses "active high" signaling. It's just an*

> *inverter. :)*

>

>

>

> > > *I can certainly understand how, say, Philips*

> *might*

> > > *want to support*

> > > *evaluation boards in PC environments. It can*

> *be*

> > > *easier to debug on*

> > > *PCs; not everyone has JTAG tools; and so on.*

> > >

Re: [RFC][PATCH] SPI subsystem

> >
> > *That's my thinking. Plus I was planning on writing*
> > *a*
> > *parallel port bit-banging adapter as a sweetnear*
> > *for*
> > *the PC Linux folk and later a SPI MMC driver so*
> > *your*
> > *PC could have a MMC slot :) (even old 386's &*
> > *486's*
> > *which don't have USB and thus no card readers!).*
>
> *Yes, I expect an SPI bitbanger will be useful to*
> *some folk.*
> *As for MMC ... it'll be interesting to watch that*
> *play out;*
> *won't the mmc_block code need to change?*
>

I don't know, I would hope not. If the mmc core is completely generic then I think I should only have to write a driver like mmci and not have to change the mmc_block or mmc core layers.

>
> *I attach the latest snapshot of my code. You'll*
> *notice two changes:*
> *(a) suspend/resume calls; this code seems pretty*
> *complete except for*
> *the protocol tweaking options; (b) a new method*
> *that'll be handy*
> *when doing things like hotplugging a card with an*
> *SPI controller*
> *and a few soldered-on SPI devices. (Like that USB*
> *prototype.)*

OK. I'll have a look at this and send another reply.

Mark

>
> - *Dave*
>
>
> =====
>
> *This is the start of a small SPI framework that*
> *started fresh, so that*
> *doesn't perpetuate the "i2c driver model mess".*
>
> - *It's still less than 2KB of ".text" (ARM,*
> *modules enabled). If there*

Linux-Kernel: Re: [RFC][PATCH] SPI subsystem

> *must be more code than the drivers, that's the*
> *right size budget. :)*
>
> – *The guts use board-specific SPI device tables to*
> *build the driver*
> *model tree. (Hardware probing is rarely an*
> *option.)*
>
> – *The Kconfig should be informative about the*
> *scope and content of SPI.*
>
> – *Building more drivers into this framework likely*
> *means updating the*
> *I/O "async message" model to include protocol*
> *tweaking (like I2C).*
>
> – *No userspace API. There are several*
> *implementations to compare.*
> *Implement them like any other driver; no magic*
> *hooks!*
>
> *This should suffice for writing (or adapting) real*
> *driver code, such*
> *as for SPI master controllers (including*
> *hotpluggable ones like a*
> *parport bitbanger) or protocol masters.*
>
>
> --- /dev/null 1970-01-01 00:00:00.000000000 +0000
> +++ osk/include/linux/spi.h 2005-09-15
> 18:08:00.293267631 -0700
> @@ -0,0 +1,268 @@
> +/*
> + * Copyright (C) 2005 David Brownell
> + *
> + * 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.
> + */
> +
> +#ifndef __LINUX_SPI_H
> +#define __LINUX_SPI_H
> +
> +/*
> + * PROTOTYPE !!!
> + *
> + * The focus is on driver model support ... enough
> for SPI mastering
> + * board setups to work. The I/O model still needs
> attention, since
> + * SPI protocols seem to need to tweaking.
> + */
> +
> +
> +/*-----*/
> +
> +/*
> + * INTERFACES between SPI master drivers and
> infrastructure
> + *
> + * There are two types of master (or slave) driver:
> "controller" drivers
> + * usually work on platform devices and touch chip
> registers; "protocol"
> + * drivers work on abstract SPI devices by asking a
> controller driver to
> + * transfer the relevant data, and they shouldn't
> much care if they use
> + * one controller or another.
> + *
> + * A "struct device_driver" for an SPI device uses
> "spi_bus_type" and needs
> + * no special API wrappers (much like
> platform_bus). These drivers are
> + * bound to devices based on their names (much like
> platform_bus), and
> + * are available in dev->driver.
> + */
> +extern struct bus_type spi_bus_type;
> +
> +struct spi_device { /* this proxies the device
> through a master */
> + struct device dev;
```

Linux-Kernel: Re: [RFC][PATCH] SPI subsystem

```
> + struct spi_master *master;
> + u32 max_speed_hz;
> + u8 chip_select;
> + u8 mode;
> + #define SPI_CPHA 0x01 /* clock phase */
> + #define SPI_CPOL 0x02 /* clock polarity */
> + #define SPI_MODE_0 (0|0)
> + #define SPI_MODE_1 (0|SPI_CPHA)
> + #define SPI_MODE_2 (SPI_CPOL|0)
> + #define SPI_MODE_3 (SPI_CPOL|SPI_CPHA)
> +
> + // MAYBE ADD hooks for protocol options that
> + affect how
> + // the controller talks to its chips:
> + // - chipselect polarity (default active low)
> + // - spi wordsize (default 8 bits unsigned)
> + // - bit order (default is wordwise msb-first)
> + // - memory packing (default: 12 bit samples into
> + 16 bits, msb)
> + // - priority
> + // - delays
> + // - ...
> +
> +};
> +
> +static inline struct spi_device
> +*to_spi_device(struct device *dev)
> +{
> + return container_of(dev, struct spi_device, dev);
> +}
> +
> +
> +struct spi_message;
> +
> +struct spi_master {
> + struct class_device cdev;
> +
> + /* other than zero (== assign one dynamically),
> + bus_num is fully
> + * board-specific. usually that simplifies to
> + being SOC-specific.
> + * for example: one SOC has three SPI
> + controllers, numbered 1..3,
> + * and one board's schematics might show it using
> + SPI-2. software
> + * would normally use bus_num=2 for that
> + controller.
> + */
> + u16 bus_num;
> +
> + /* chipselects will be integral to many
```

```

> controllers, while others
> + * might use board-specific GPIOs.
> + */
> + u16 num_chipselect;
> +
> + /* setup mode and clock, etc */
> + int (*setup)(struct spi_device *spi);
> +
> + /* bidirectional bulk transfers
> + * + transfer() may not sleep
> + * + to a given spi_device, message queueing is
> pure fifo
> + * + if there are multiple spi_device children,
> the i/o queue
> + * arbitration algorithm is unspecified (round
> robin, fifo,
> + * priority, reservations, preemption, etc)
> + * + the master's main job is to process its
> message queue,
> + * selecting a chip then transferring data
> + * + for now there's no remove-from-queue
> operation, or
> + * any other request management
> + */
> + int (*transfer)(struct spi_device *spi,
> + struct spi_message *mesg);
> +};
> +
> +/* the controller driver manages memory for the
> spi_master classdev,
> + * normally contained inside its associated private
> state.
> + */
> +extern int spi_register_master(struct device *host,
> + struct spi_master *spi);
> +extern void spi_unregister_master(struct spi_master
> + *spi);
> +
> +
>
+/*-----*/
> +
> +/*
> + * I/O INTERFACE between SPI controller and
> protocol drivers
> + *
> + * The interface is a queue of spi_messages, each
> transferring data
> + * between the controller and memory buffers. SPI
> protocol drivers just
> + * provide spi_messages to hardware controller

```

```

> drivers. SPI controller
> + * drivers process that queue, issuing completion
> callbacks as appropriate.
> + *
> + * The spi_messages themselves consist of a series
> of read+write transfer
> + * segments. Those segments always read the same
> number of bits as they
> + * write; but one or the other is easily ignored by
> passing a null buffer
> + * pointer. (This is unlike most types of I/O API,
> because SPI hardware
> + * is full duplex.)
> + *
> + * NOTE: Allocation of spi_transfer and
> spi_message memory is entirely
> + * up to the protocol driver, which guarantees the
> integrity of both (as
> + * well as the data buffers) for as long as the
> message is queued.
> + */
> +
> +struct spi_transfer {
> + /* it's ok if tx_buf == rx_buf (right?)
> + * for MicroWire, one buffer must be null
> + * buffers must work with dma_*map_single() calls
> + */
> + void *tx_buf, *rx_buf;
> + unsigned len;
> +};
> +
> +/* spi messages will often be stack-allocated */
> +struct spi_message {
> + struct spi_transfer *transfers;
> + unsigned n_transfer;
> +
> + struct spi_device *dev;
> +
> + /* Optionally leave this chipselect active
> afterward */
> + unsigned csrel_disable:1;
> +
> + /* completion is reported this way */
> + void (*complete)(void *context);
> + void *context;
> + unsigned actual_length;
> + int status;
> +
> + /* for optional controller driver use */
> + struct list_head queue;
> +};

```

```

> +
> +/**
> + * spi_setup -- setup SPI mode and clock rate
> + * @spi: the device whose settings are being
> modified
> + *
> + * SPI protocol drivers may need to update the
> transfer mode if the
> + * device doesn't work with the mode 0 default.
> They may likewise need
> + * to update clock rates. This function changes
> those settings,
> + * and must be called from a context that can
> sleep.
> + */
> +static inline int
> +spi_setup(struct spi_device *spi)
> +{
> + return spi->master->setup(spi);
> +}
> +
> +
> +/* synchronous SPI transfers; these may sleep
> uninterruptibly */
> +extern int spi_sync(struct spi_device *spi, struct
> spi_message *message);
> +extern int spi_w8r8(struct spi_device *spi, u8
> cmd);
> +extern int spi_w8r16(struct spi_device *spi, u8
> cmd);
> +
> +
> +/**
> + * spi_async -- asynchronous SPI transfer
> + * @spi: device with which data will be exchanged
> + * @message: describes the data transfers,
> including completion callback
> + *
> + * This call may be used in_irq in other contexts
> which can't sleep,
> + * as well as from task contexts which can sleep.
> + *
> + * The completion callback is invoked in a context
> which can't sleep.
> + * Before that invocation, the value of
> message->status is undefined.
> + * When the callback is issue, message->status
> holds either zero (to
> + * indicate complete success) or a negative error
> code.
> + */

```

```

> +static inline int
> +spi_async(struct spi_device *spi, struct
> spi_message *message)
> +{
> + message->dev = spi;
> + return spi->master->transfer(spi, message);
> +}
> +
> +
+/*-----*/
> +
> +/*
> + * INTERFACE between board init code and SPI
> infrastructure.
> + *
> + * No SPI driver ever sees these SPI device table
> segments, but
> + * it's how the SPI core (or adapters that get
> hotplugged) grows
> + * the driver model tree.
> + */
> +
> +/* board-specific information about each SPI device
> */
> +struct spi_board_info {
> + /* the device name and module name are coupled,
> like platform_bus.
> + *
> + * platform_data has things like IRQ assignments
> (just pass the int
> + * that goes to request_irq), related GPIOs, and
> so on.
> + */
> + char modalias[KOBJ_NAME_LEN];
> + void *platform_data;
> +
> + /* slower signaling on noisy or low voltage boards
> */
> + u32 max_speed_hz;
> +
> + /* bus_num is board specific and matches the
> bus_num of some
> + * spi_master that will probably be registered
> later.
> + *
> + * chip_select reflects how this chip is wired to
> that master;
> + * it's less than num_chipselect.
> + */
> + u16 bus_num;
> + u16 chip_select;

```

```

> +
> + /* ... may need additional spi_device chip config
> data here.
> + * avoid stuff protocol drivers can set; but
> include stuff
> + * needed to behave without being bound to a
> driver:
> + * - chipselect polarity
> + * - quirks like clock rate mattering when not
> selected
> + */
> +};
> +
> +extern int
> +spi_register_board_info(struct spi_board_info const
> *info, unsigned n);
> +
> +extern struct spi_device *
> +spi_new_device(struct spi_master *, struct
> spi_board_info *);
> +
> +static inline void
> +spi_unregister_device(struct spi_device *spi)
> +{
> + if (spi)
> + device_unregister(&spi->dev);
> +}
> +
> +
> +#endif /* __LINUX_SPI_H */
> ---- /dev/null 1970-01-01 00:00:00.000000000 +0000
> +++ osk/drivers/spi/spi.c 2005-09-15
> 17:22:46.104517401 -0700
> @@ -0,0 +1,482 @@
> +/* spi.c -- prototype of SPI init/core code
> + *
> + * Copyright (C) 2005 David Brownell
> + *
> + * 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.
> + */
> +
> +#include <linux/autoconf.h>
> +#include <linux/kernel.h>
> +#include <linux/device.h>
> +#include <linux/init.h>
> +#include <linux/spi.h>
> +
> +
> +* SPI bustype and spi_master class are registered
> during early boot,
> + * usually after board init code provided the SPI
> device tables, and
> + * are available when driver init code needs them.
> + *
> + * Drivers for SPI devices are like those for
> platform bus devices:
> + * (a) no bus-specific API wrappers (== needless
> bloat here)
> + * (b) matched to devices using device names
> + * (c) should support "native" suspend and resume
> methods
> + */
> +static void spidev_release(struct device *dev)
> +{
> + const struct spi_device *spi = to_spi_device(dev);
> +
> + class_device_put(&spi->master->cdev);
> + kfree(dev);
> +}
> +
> +// no, we probably don't need to list speed and
> mode in sysfs...
> +
> +static ssize_t
> +modalias_show(struct device *dev, struct
> device_attribute *a, char *buf)
> +{
> + const char *modalias = strchr(dev->bus_id, '-') +
> 1;
> +
> + return snprintf(buf, BUS_ID_SIZE + 1, "%s\n",
```

```

> modalias);
> +}
> +
> +#ifdef DEBUG
> +
> +static ssize_t
> +maxspeed_show(struct device *dev, struct
> device_attribute *a, char *buf)
> +{
> + const struct spi_device *spi = to_spi_device(dev);
> +
> + return sprintf(buf, "%u\n", spi->max_speed_hz);
> +}
> +
> +static ssize_t
> +mode_show(struct device *dev, struct
> device_attribute *a, char *buf)
> +{
> + struct spi_device *spi = to_spi_device(dev);
> +
> + return sprintf(buf, "%u\n", spi->mode);
> +}
> +
> +#endif /* DEBUG */
> +
> +static struct device_attribute spi_dev_attrs[] = {
> + __ATTR_RO(modalias),
> +#ifdef DEBUG
> + __ATTR_RO(maxspeed),
> + __ATTR_RO(mode),
> +#endif /* DEBUG */
> + __ATTR_NULL,
> +};
> +
> +static int spi_match_device(struct device *dev,
> struct device_driver *drv)
> +{
> + const char *modalias = strchr(dev->bus_id, '-') +
> 1;
> +
> + return strncmp(modalias, drv->name, BUS_ID_SIZE)
> == 0;
> +}
> +
> +static int spi_hotplug(struct device *dev, char
> **envp, int num_envp,
> + char *buffer, int buffer_size)
> +{
> + const char *modalias = strchr(dev->bus_id, '-') +
> 1;
> +

```

Linux-Kernel: Re: [RFC][PATCH] SPI subsystem

```
> + envp[0] = buffer;
> + snprintf(buffer, buffer_size, "MODALIAS=%s",
> modalias);
> + envp[1] = NULL;
> + return 0;
> +}
> +
> +/* suspend/resume in "struct device_driver" don't
> really need that
> + * strange third parameter, so we just make it a
> constant and expect
> + * drivers to ignore it.
> + */
> +static int spi_suspend(struct device *dev,
> pm_message_t message)
> +{
> + if (dev->driver && dev->driver->suspend)
> + return dev->driver->suspend(dev, message,
> SUSPEND_POWER_DOWN);
> + else
> + return 0;
> +}
> +
> +static int spi_resume(struct device *dev)
> +{
> + if (dev->driver && dev->driver->resume)
> + return dev->driver->resume(dev, RESUME_POWER_ON);
> + else
> + return 0;
> +}
> +
> +struct bus_type spi_bus_type = {
> + .name = "spi",
> + .dev_attrs = spi_dev_attrs,
> + .match = spi_match_device,
> + .hotplug = spi_hotplug,
> + .suspend = spi_suspend,
> + .resume = spi_resume,
> +};
> +EXPORT_SYMBOL_GPL(spi_bus_type);
> +
> +static struct class spi_master_class = {
> + .name = "spi_master",
> + .owner = THIS_MODULE,
> +};
> +
> +
> +
> +/*-----*/
> +
> +/* SPI devices should normally not be created by
```

```

> SPI device drivers; that
> + * would make them board-specific. Similarly with
> SPI master drivers.
> + * Device registration normally goes into like
> arch/.../mach.../board-YYY.c
> + * with other information about mainboard devices.
> + */
> +
> +struct boardinfo {
> + struct list_head list;
> + unsigned n_board_info;
> + struct spi_board_info board_info[0];
> +};
> +
> +static LIST_HEAD(board_list);
> +static DECLARE_MUTEX(board_lock);
> +
> +#define kzalloc(n, flags) kcalloc(1,(n),(flags))
> +
> +static int __init_or_module
> +check_child(struct device *dev, void *data)
> +{
> + const struct spi_device *spi =
> + to_spi_device(dev);
> + const struct spi_board_info *chip = data;
> +
> + return (spi->chip_select == chip->chip_select);
> +}
> +
> +
> +/* On typical mainboards, this is purely internal;
> + and it's not needed
> + * after board init creates the hard-wired devices.
> + Some development
> + * platforms may not be able to use
> + spi_register_board_info though, and
> + * this is exported so that for example a USB or
> + parport based adapter
> + * driver could add devices.
> + */
> +struct spi_device *__init_or_module
> +spi_new_device(struct spi_master *master, struct
> +spi_board_info *chip)
> +{
> + struct spi_device *proxy;
> + struct device *dev = master->cdev.dev;
> + int status;
> +
> + /* NOTE: caller did any chip->bus_num checks
> + necessary */
> +

```

```

> + /* only one child per chipselect, ever */
> + if (device_for_each_child(dev, chip, check_child))
> + return NULL;
> +
> + if (!class_device_get(&master->cdev))
> + return NULL;
> +
> + proxy = kzalloc(sizeof *proxy, GFP_KERNEL);
> + if (!proxy) {
> + dev_err(dev, "can't alloc dev for cs%d\n",
> + chip->chip_select);
> + goto fail;
> + }
> + proxy->master = master;
> + proxy->chip_select = chip->chip_select;
> + proxy->max_speed_hz = chip->max_speed_hz;
> +
> + snprintf(proxy->dev.bus_id, sizeof
> proxy->dev.bus_id,
> + "%s.%u-%s", master->cdev.class_id,
> + chip->chip_select, chip->modalias);
> + proxy->dev.parent = dev;
> + proxy->dev.bus = &spi_bus_type;
> + proxy->dev.platform_data = chip->platform_data;
> + proxy->dev.release = spidev_release;
> +
> + /* drivers may modify this default i/o setup */
> + status = master->setup(proxy);
> + if (status < 0) {
> + dev_dbg(dev, "can't %s %s, status %d\n",
> + "setup", proxy->dev.bus_id, status);
> + goto fail;
> + }
> +
> + status = device_register(&proxy->dev);
> + if (status < 0) {
> + dev_dbg(dev, "can't %s %s, status %d\n",
> + "add", proxy->dev.bus_id, status);
> +fail:
> + class_device_put(&master->cdev);
> + kfree(proxy);
> + return NULL;
> + }
> + dev_dbg(dev, "registered child %s\n",
> proxy->dev.bus_id);
> + return proxy;
> +}
> +EXPORT_SYMBOL_GPL(spi_new_device);
> +
> +/*
> + * Board-specific early init code calls this

```

```

> (probably during arch_initcall)
> + * with segments of the SPI device table. Any
> device nodes are created later,
> + * after the relevant parent SPI controller
> (bus_num) is defined. We keep
> + * this table of devices forever, so that reloading
> a controller driver will
> + * not make Linux forget about these hard-wired
> devices.
> + *
> + * Other code can also call this, e.g. a particular
> add-on board might provide
> + * SPI devices through its expansion connector, so
> code initializing that board
> + * would naturally declare its SPI devices.
> + *
> + * The board info passed can safely be __initdata
> ... but be careful of
> + * any embedded pointers (platform_data, etc),
> they're copied as-is.
> + */
> +int __init_or_module // would be __init except for
> SPI_EXAMPLE
> +spi_register_board_info(struct spi_board_info const
> *info, unsigned n)
> +{
> + struct boardinfo *bi;
> +
> + bi = kmalloc (sizeof (*bi) + n * sizeof (*info),
> GFP_KERNEL);
> + if (!bi)
> + return -ENOMEM;
> + bi->n_board_info = n;
> + memcpy(bi->board_info, info, n * sizeof (*info));
> +
> + down(&board_lock);
> + list_add_tail(&bi->list, &board_list);
> + up(&board_lock);
> + return 0;
> +}
> +EXPORT_SYMBOL_GPL(spi_register_board_info);
> +
> +static void __init_or_module
> +scan_boardinfo(struct spi_master *master)
> +{
> + struct boardinfo *bi;
> + struct device *dev = master->cdev.dev;
> +
> + down(&board_lock);
> + list_for_each_entry(bi, &board_list, list) {
> + struct spi_board_info *chip = bi->board_info;

```

```

> + unsigned n;
> +
> + for (n = bi->n_board_info; n > 0; n--, chip++) {
> + if (chip->bus_num != master->bus_num)
> + continue;
> + if (chip->chip_select >= master->num_chipselect)
> + {
> + dev_dbg(dev, "cs%d > max %d\n",
> + chip->chip_select,
> + master->num_chipselect);
> + continue;
> + }
> + (void) spi_new_device(master, chip);
> + }
> + }
> + up(&board_lock);
> + }
> +
> +
+/*-----*/
> +
> +/**
> + * spi_register_master - register SPI master
> controller
> + * @dev: the controller
> + * @master: the master, with all uninitialized
> fields zeroed
> + *
> + * This call is used only by SPI master controller
> drivers, which are the
> + * only ones directly touching chip registers.
> + *
> + * SPI master controllers connect to their drivers
> using some non-SPI bus,
> + * such as the platform bus. The final stage of
> probe() in that code
> + * includes calling spi_register_master(), with
> memory managed by that
> + * controller, to hook up to this SPI bus glue.
> + *
> + * SPI controllers use board specific (often SOC
> specific) bus numbers,
> + * and board-specific addressing for SPI devices
> combines those numbers
> + * with chip select numbers. Since SPI does not
> directly support dynamic
> + * device identification, boards need configuration
> tables tellin which
> + * chip is at which address.
> + *
> + * This must be called from context that can sleep.

```

```

> It returns zero
> + * on success, else a negative error code.
> + */
> +int __init_or_module
> +spi_register_master(struct device *dev, struct
> spi_master *master)
> +{
> + static atomic_t dyn_bus_id = ATOMIC_INIT(0);
> + int status = -ENODEV;
> +
> + if (list_empty(&board_list)) {
> + dev_dbg(dev, "spi board info is missing\n");
> + goto done;
> + }
> +
> + /* convention: dynamically assigned bus IDs count
> down from the max */
> + if (master->bus_num == 0) {
> + master->bus_num = atomic_dec_return(&dyn_bus_id);
> + dev_dbg(dev, "spi%d, dynamic bus number\n",
> master->bus_num);
> + }
> +
> + /* ELSE: verify that the ID isn't in use already
> */
> +
> + master->cdev.class = &spi_master_class;
> + master->cdev.dev = get_device(dev);
> + class_set_devdata(&master-
> controller (which
> + provides the clock and chipselect), you can
> enable that
> + controller and the protocol drivers for the SPI
> slave chips
> + that are connected.
> +
> +if SPI_MASTER
> +
> +comment "SPI Master Controller Drivers"
> +
> +config SPI_EXAMPLE
> + tristate "SPI Platform Example"
> + help
> + This just builds some sample code uses the core
> APIs to build
> + some SPI devices and plug in dummy controller
> drivers.
> +
> +# Atmel AT91rm9200 (and some other AT91 family
> chips)
> +config SPI_AT91

```

> + *tristate "AT91 as SPI master"*
> + *depends on ARCH_AT91*
> + *help*
> + *This implements SPI master mode using an SPI*
> *controller.*
> +
> +*# FIXME bitbangers need arch-specific ways to*
> *access the*
> +*# right GPIO pins, probably using platform data and*
> *maybe*
> +*# using platform-specific minidrivers.*
> +*config SPI_BITBANG*
> + *tristate "Bitbanging SPI master"*
> + *help*
> + *You can implement SPI using GPIO pins, as this*
> *driver*
> + *eventually should do.*
> +
> +*# Motorola Coldfire (m68k)*
> +*config SPI_COLDFIRE*
> + *tristate "Coldfire QSPI as SPI master"*
> + *depends on COLDFIRE*
> + *help*
> + *This implements SPI master mode using the QSPI*
> *controller.*
> +
> +*# Motorola MPC (PPC)*
> +*config SPI_MPC*
> + *tristate "MPC SPI master"*
> + *depends on PPC && MPC*
> + *help*
> + *This implements SPI master mode using the MPC*
> *SPI controller.*
> +
> +*# TI OMAP (ARM)*
> +*config SPI_OMAP*
> + *tristate "OMAP SPI controller as master"*
> + *depends on ARCH_OMAP*
> + *help*
> + *This implements SPI master mode using the*
> *dedicated SPI*
> + *controller.*
> +
> +*config SPI_OMAP_MCBSP*
> + *tristate "OMAP MCBSP as SPI master"*