THIS SPEC IS OBSOLETE

Spec No: 001-98566

Spec Title: AN98566 - LINUX CONFIGURATION FOR CYPRESS SPI

Replaced by: NONE

Linux Configuration for Cypress SPI

Author: Cypress

AN98566 compares the software driver stacks used for parallel NOR and SPI flash devices in Linux and discusses the requirements needed in order to get Cypress SPI devices up and running.

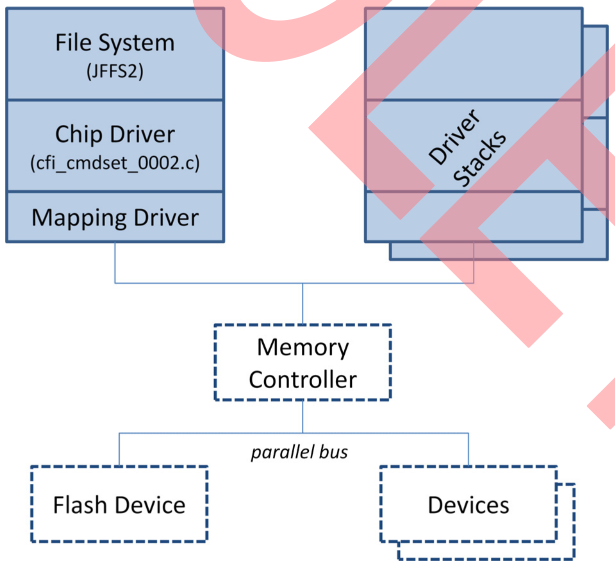
1 Introduction

Many customers have been employing parallel NOR flash for years and are now moving towards serial (SPI) flash. This requires not only to replace the underlying hardware but also to apply software changes addressing e.g. the different command set used by SPI flash devices. This document compares the software driver stacks used for parallel NOR and SPI flash devices in Linux. It shows their differences and discusses the requirements needed in order to get Cypress SPI devices up and running.

2 Driver Stacks for Parallel and Serial Cypress Flash Devices

The driver stack for parallel NOR flash under Linux is pretty simple. Basically, it includes three layers: a mapping driver that is responsible to make the flash device accessible, a chip driver that implements the vendor specific command set, and finally a flash file system such as JFFS2 on top of the stack that offers the standard file system API to user space applications. Figure 1 shows a block diagram of the driver infrastructure.

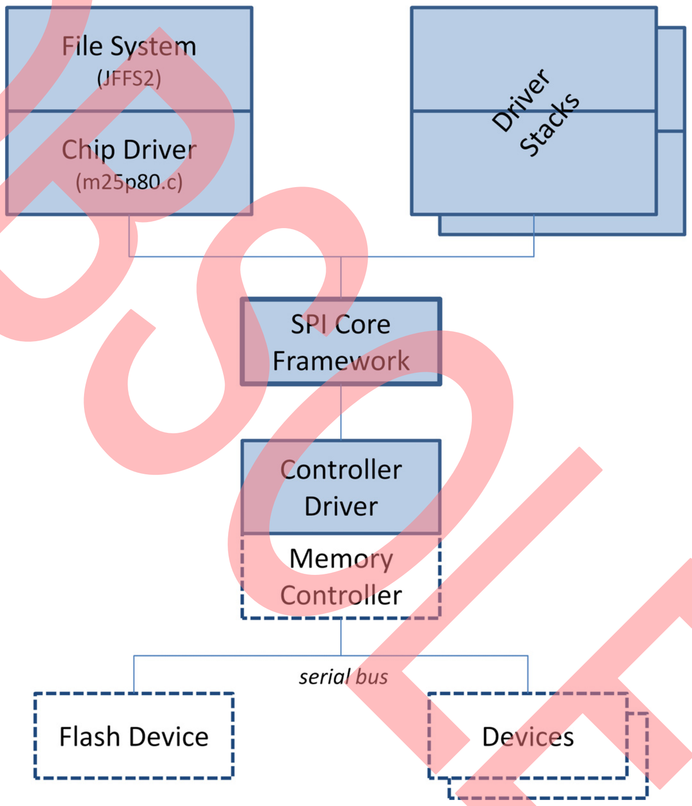
Figure 1. Driver Infrastructure for Parallel Flash Devices



For new system designs, only the mapping driver has to be changed. The Linux kernel sources already include a generic mapping driver (physmap.c) that needs just three configuration parameters (physical flash base address, size, and bus width) before it can be used. In any case, mapping drivers for parallel NOR flash are very simple and lightweight. Most of them just map the physical I/O memory of the entire device into the (virtual) kernel address space at initialization time. Afterwards, they forward all read and write word requests directly to the mapped memory window. If multiple devices are attached to the same bus or controller they all get their own independent driver stack. Proper synchronization of the bus accesses is guaranteed by the memory controller, i.e. it is fully implemented in hardware.

For SPI devices things are more complicated, see Figure 2. File system and chip driver look similar as in the parallel case but the hardware specific portion of the stack is much different. A SPI memory controller typically comprises small buffers (FIFOs) to queue incoming and outgoing data from the bus. These buffers need to be loaded and monitored by a software driver. For example, when the read buffer overflows the controller might generate an interrupt indicating to the software driver that it should retrieve the data from its buffer before moving on.

Figure 2. Driver Infrastructure for Serial Flash Devices



For every SPI memory controller in the system there must be a dedicated controller driver. That controller driver is not directly connected to the chip driver. Instead, it is connected to a central SPI core framework that manages and routes all SPI traffic in the system. There might be multiple SPI buses in a system each with its own controller driver but there is always just one SPI core framework. All controller and chip drivers register with this framework. Furthermore, board drivers provide detailed information about the system configuration, e.g. the kind of SPI devices available in the system, their maximum supported frequency, the bus and chipselect numbers they are attached to as well as the names of the chip driver that are needed for them. Based on this data, the SPI framework binds every SPI device to the appropriate chip driver at initialization time.

3 Building Blocks

The following chapters discuss the individual building blocks and drivers for a SPI solution in more detail.

3.1 Flash File Systems

Due to the standardized MTD interface provided by the chip driver to upper layers, the flash file system drivers usually remain unchanged. The FFS drivers do not need to know many details about the underlying hardware and driver infrastructure. All relevant parameters such as sector size and geometry can be dynamically retrieved via the standardized MTD interface.

When configuring the kernel make sure that the preferred FFS is enabled, i.e. that the kernel option

CONFIG_JFFS2_FS

or similar is set.

3.2 Chip Driver

Support for Cypress SPI flash devices comes with the MTD chip driver m25p80.c. This driver implements the device specific command set, i.e. the protocol or language needed to talk to the physical device. It includes support for a whole series of compatible SPI flash devices from various vendors. Its code base has been growing continuously in the past with an increasing number of features as well as number of supported devices. Direct support for Cypress SPI devices was first added with kernel version 2.6.27. Since then, the number of directly supported Cypress OPNs (mainly FL-A and FL-P) has increased to nine for kernel version 2.6.34.

If needed, new OPNs can easily be added. For example, the following device entries describe the supported Cypress FL129P devices (256 kB and 64 kB sector versions):

```
{ "s25fl129p0", INFO(0x012018, 0x4d00, 256 * 1024,  64, 0) },
{ "s25fl129p1", INFO(0x012018, 0x4d01,  64 * 1024, 256, 0) },
```

The structure of device entries is pretty simple. Data sets start with a unique string or name followed by the standard and extended device IDs, sector size in bytes, number of sectors and finally optional flags.

Note that m25p80.c currently supports uniform sector layouts only. Furthermore, it supports single I/O only as well as no suspend/resume. In order to enable this driver, the kernel configuration macros

```
CONFIG_MTD_M25P80
CONFIG_MTD
CONFIG_MTD_CHAR
CONFIG_MTD_BLOCK
```

need to be set at least. Basic MTD support is needed so that the chip driver can register its MTD objects with the kernel offering an API to upper driver layers as well as the MTD user utilities (e.g. to format the flash device for a file system or to program an image). The MTD user utilities can be used without changes together with this chip driver.

3.3 SPI Core Framework

The SPI core framework serves as a kind of switchboard for all SPI traffic in the system. Drivers register with the framework and use a message based protocol to issue and process SPI transactions. Its API is architecture independent and is defined in include/linux/spi/spi.h.

One of the most important tasks of the SPI framework is to bind SPI chip drivers to the corresponding controller drivers and physical devices. Usually this happens via a spi_board_info struct that is initialized and provided by a board driver during system initialization. This struct provides the name of the appropriate chip driver, the maximum supported clock speed of the device, the bus and chipset numbers the device is attached to as well as some other parameters. The following example describes a SPI flash device that is attached to bus number 1, uses chip select 0 and should be managed by the m25p80 chip driver:

```
static struct spi_board_info spsn_pci_board_info = {
    .modalias = "m25p80",
    .max_speed_hz = 40000000,
    .bus_num = 1,
    .chip_select = 0
};
```

Usually, the struct is registered by the board driver via spi_register_board_info() together with other platform resources. Afterwards, the SPI core framework calls the probing function of the chip driver, i.e. m25p80.c, to check for the presence of a physical device.

To include the SPI core framework in the linux kernel make sure that the configuration option

CONFIG_SPI

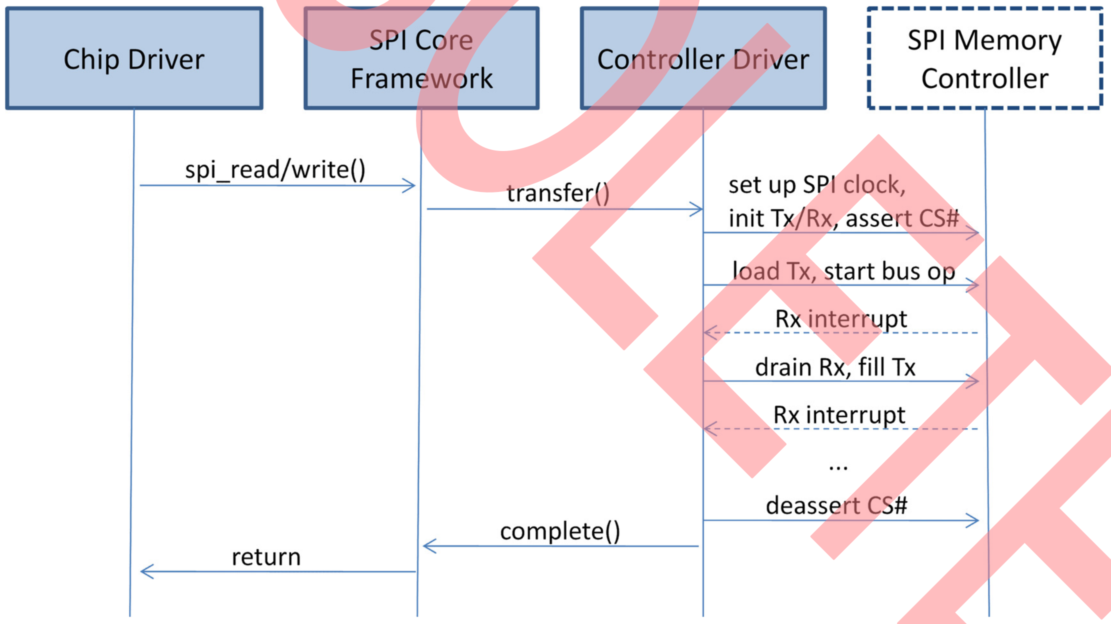
is enabled and that a spi_board_info struct is provided by the board driver.

3.4 Controller Driver

The lowest but in many cases also the most complex driver in the stack is the controller driver, i.e. the driver that operates the SPI controller hardware. Unlike with parallel NOR flash, for SPI flash there is a non-trivial software driver needed to manage the memory controller. The controller driver receives and normally also queues incoming SPI read/write requests (often spi_bitbang.c is used for the queuing purpose). Once all preceding transactions have been completed the controller driver sets the SPI bus clock to the correct value and then starts the transaction. Usually, it has to manage both the Rx and Tx buffers of the hardware controller, i.e. it keeps feeding the Tx side and at the same time drains the Rx side. In most cases this is done via interrupts where the controller issues an interrupt if either the Tx buffer is close to empty (needs to be refilled) and/or the Rx buffer is close to full (needs to be drained). If available, DMA hardware can be used at this stage to run more of these tasks in hardware, thus achieving an improved throughput and freeing some CPU cycles. Finally, once all data has been transferred, the controller driver calls the provided callback function that signals completion of the transaction to the SPI framework.

Figure 3 shows the control flow of a synchronous SPI transaction for a typical SPI controller driver and hardware.

Figure 3. Synchronous SPI Transaction



The SPI core framework provides both synchronous and asynchronous I/O functions. In the asynchronous case, control immediately returns to the chip driver. Once the SPI transaction has been completed the framework informs the chip driver via a callback. Note that the SPI flash chip driver m25p80.c uses synchronous read/write functions only.

When configuring the kernel make sure that

CONFIG_SPI_MASTER

[CONFIG_YOUR_CONTROLLER]

is set where [CONFIG_YOUR_CONTROLLER] denotes the SPI memory controller of your system. If there is no suitable controller driver included in the kernel sources you will have to develop your own. There are several controller drivers in drivers/spi that might be used as an example or starting point.

Furthermore, even if there is a controller driver included in your kernel sources it might have been developed for other, more basic SPI devices. For example, some controller drivers support fixed transaction lengths only — mainly because many SPI devices require just a few bytes to be read/written (e.g. a touch screen device). SPI flash is much more demanding in this regard. Basically, flash device transactions on the SPI bus can have any length — if supported by the controller. The m25p80.c chip driver does not split up requests by default so that e.g. read requests can have any length. If there is a hardware limit it needs to be taken care of. Basically, this means splitting up large transfers into smaller chunks.

4 Locking and Synchronization

If multiple applications and drivers compete for the same piece of hardware then the code has to include some sort of synchronization. In the SPI flash case there are synchronization points at two levels.

The higher level point is inside the chip driver. For every registered flash device m25p80.c allocates a mutex to synchronize concurrent requests from higher levels. The mutex is locked for the entire duration of any MTD operation (read/write/erase). If other drivers or applications want to access the flash device at the same time they will have to wait until the operation has completed and the mutex is released. This locking scheme implements a coarse granularity compared to the locking scheme used by the parallel NOR flash drivers. Parallel NOR flash drivers can suspend an ongoing operation for incoming high priority requests, thus temporarily unlocking the device. For example the AMD/Cypress driver cfi_cmdset_0002.c suspends an ongoing erase if it gets a read request. After the erase has been suspended it serves the read request and then resumes the erase. This is currently not supported by the SPI chip driver.

Next to the protocol level locking there is a low level synchronization point in the controller driver to synchronize concurrent requests to multiple SPI devices on the same bus. If two chip drivers issue requests at the same time to two devices on the same SPI bus then the controller driver needs to guarantee that one request waits until the other has finished. This kind of synchronization is usually implemented by the queuing mechanism within the controller driver. The common queue represents a unique resource that is protected via some sort of locking. Only the driver that owns the lock can add new requests to the queue, others will have to wait until the lock is released. Once all requests have been queued the controller driver does not need to apply further synchronization means. It can simply take the next request from the queue head, serve it and continue this way until all requests are served, i.e. until the queue is empty.

5 Example

Let's consider a simple example showing how to add SPI support for a FL128P under Linux 2.6.31, based on a Cypress PCI card. Cypress PCI cards are universal development and debugging tools that are available from Cypress Hardware Development Tools.

In this case, our board driver is a simple Linux PCI driver. Its initialization or probing function is automatically called by the kernel when the given PCI vendor and device IDs are detected during the PCI bus scan at system start-up. The PCI probing function serves as a good location to initialize the SPI infrastructure as well, i.e. to provide the required spi_board_info struct. The following piece of code shows how this initialization is done in detail.

```
#include <linux/spi/spi.h>
...
static struct spi_board_info spsn_pci_board_info = {
   .modalias = "m25p80",
   .max_speed_hz = 40000000,
   .bus_num = 0,
   .chip_select = 0
};

static int spsn_pci_probe(struct pci_dev *dev, const struct pci_device_id *id)
{
   ...
```

```
//  Register SPI master controller driver and device

board.smaster = spi_alloc_master( &dev->dev, 0);
if (board.smaster == NULL)
   { printk( "ERROR: spi_alloc_master failed!\n"); return -ENODEV; }

board.smaster->bus_num = 0;
board.smaster->num_chipselect = 1;
board.smaster->setup    = spsn_pci_spi_setup;
board.smaster->transfer = spsn_pci_spi_transfer;
board.smaster->cleanup  = spsn_pci_spi_cleanup;

if (spi_register_master( board.smaster))
   { printk ("ERROR spi_register_master() failed!\n");
      spi_master_put( board.smaster); return -1; }

if (spi_new_device( board.smaster, &spsn_pci_board_info) == NULL)
   { printk ("ERROR spi_new_device() failed!\n");
       spi_master_put( board.smaster); return -1; }

printk( "spi master controller and device registered\n");
...
```

Next to the registration of the spi_board_info struct the code also allocates, initializes, and registers a master controller driver. The exact implementation of this part is beyond the scope of this document.

Nevertheless, the piece of code shows how the different components nicely play together. After the SPI device has been registered via spi_new_device(), the SPI core framework loads the m25p80 kernel module, binds the chip driver to the controller driver and finally calls the probe() function of m25p80.c. If a physical device is present, it will be detected by the chip driver and a MTD object will be registered.

Of course, the kernel configuration options mentioned above have to be set as well. The m25p80.c driver can remain unchanged in this case since the FL128P is already supported by default.

6 Questions and Support

If you have further questions or support requests please direct them to the Cypress Engineering Solutions portal at http://www.spansion.com/Support/SES/Pages/ask_spansion.html.

Document History Page

Document Title: AN98566 - Linux Configuration for Cypress SPI Document Number: 001-98566				
Rev.	**ECN No.**	**Orig. of Change**	**Submission Date**	**Description of Change**
**	-	-	08/09/2010	Initial version
*A	4928144	MSWI	09/21/2015	Updated in Cypress template
*B	5844295	AESATMP8	08/04/2017	Updated logo and Copyright.
*C	6035383	GEHO	01/19/2018	Obsolete the document, as the contents are old and no longer needed.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Video | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709