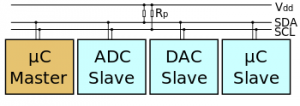
# I2C Protocol

I²C (Inter-Integrated Circuit) is a master slave, single-ended, serial bus used for attaching low-speed peripherals to computer motherboards and embedded systems. I²C uses only two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL), pulled up with resistors. Typical voltages used are +5 V or +3.3 V. The I²C reference design has a 7-bit or a 10-bit address space. Common I²C bus speeds are the 100 kbit/s standard mode, 10 kbit/s low-speed mode, 400 kbit/s Fast mode, 1 Mbit/s Fast mode and 3.4 Mbit/s High Speed mode. The maximum number of nodes is limited by the address space, and also by the total bus capacitance of 400 pF, which restricts practical communication distances.

[](http://invo-tronics.com/wp-content/uploads/2014/09/i2c.png)A sample schematic with one master (a microcontroller), three slave nodes (an ADC, a DAC, and a microcontroller), and pull-up resistors Rp.

The I2C bus has two roles for nodes:

Master and Slave:

* Master node — node that generates the clock and initiates communication with slaves
* Slave node — node that receives the clock and responds when addressed by the master

There may be four potential modes of operation for a given bus device, although most devices only use a single role and its two modes:

* master transmit — master node is sending data to a slave
* master receive — master node is receiving data from a slave
* slave transmit — slave node is sending data to the master
* slave receive — slave node is receiving data from the master

The master is initially in master transmit mode by sending a start bit followed by the 7-bit address of the slave it wishes to communicate with, which is finally followed by a single bit representing whether it wishes to write(0) to or read(1) from the slave.

If the slave exists on the bus then it will respond with an ACK bit (active low for acknowledged) for that address. The master then continues in either transmit or receive mode (according to the read/write bit it sent), and the slave continues in its complementary mode (receive or transmit, respectively).

The address and the data bytes are sent most significant bit first. The start bit is indicated by a high-to-low transition of SDA with SCL high; the stop bit is indicated by a low-to-high transition of SDA with SCL high. All other transitions of SDA take place with SCL low.

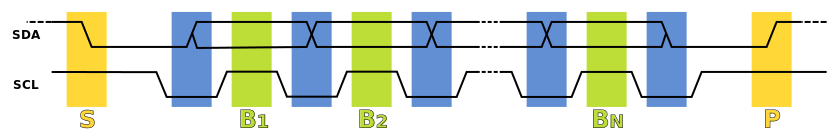
If the master wishes to write to the slave then it repeatedly sends a byte with the slave sending an ACK bit. (In this situation, the master is in master transmit mode and the slave is in slave receive mode.)

If the master wishes to read from the slave then it repeatedly receives a byte from the slave, the master sending an ACK bit after every byte but the last one. (In this situation, the master is in master receive mode and the slave is in slave transmit mode.)  
The master then either ends transmission with a stop bit, or it may send another START bit if it wishes to retain control of the bus for another transfer (a “combined message”).

**Key symbols**  
S (1 bit) : Start bit  
P (1 bit) : Stop bit  
R/W (1 bit) : Read/Write bit. Rd equals 1, Wr equals 0.  
A, NA (1 bit) : Accept and reverse accept bit.  
Addr (7 bits): I2C 7 bit address. Can be expanded as usual to get a 10 bit I2C address.  
Comm (8 bits): Command byte, a data byte which often selects a register on the device.  
Data (8 bits): A plain data byte.  
Count (8 bits): A data byte containing the length of a block operation.  
[..]: Data sent by I2C device, as opposed to data sent by the host adapter.

**Simple send transaction**  
This corresponds to i2c\_master\_send.  
S Addr W [A] Data [A] Data [A] … [A] Data [A] P

**Simple receive transaction**  
This corresponds to i2c\_master\_recv.  
S Addr R [A] [Data] A [Data] A … A [Data] NA P

[](http://invo-tronics.com/wp-content/uploads/2014/09/i2c-data-transfer.png)

1. Data Transfer is initiated with a START bit (S) signaled by SDA being pulled low while SCL stays high.
2. SDA sets the 1st data bit level while keeping SCL low (during blue bar time.)
3. The data is sampled (received) when SCL rises (green) for the first bit (B1).
4. This process repeats, SDA transitioning while SCL is low, and the data being read while SCL is high (B2, Bn).
5. A STOP bit (P) is signaled when SDA is pulled high while SCL is high.

In order to avoid false marker detection, SDA is changed on the SCL falling edge and is sampled and captured on the rising edge of SCL.

At the physical layer, both SCL and SDA lines are of open-drain design, thus, pull-up resistors are needed. Pulling the line to ground is considered a logical zero while letting the line float is a logical one. This is used as a channel access method. High speed systems (and some others) also add a current source pull up, at least on SCL; this accommodates higher bus capacitance and enables faster rise times. An important consequence of this is that multiple nodes may be driving the lines simultaneously. If any node is driving the line low, it will be low. Nodes that are trying to transmit a logical one (i.e. letting the line float high) can see this, and thereby know that another node is active at the same time.

When used on SCL, this is called clock stretching and gives slaves a flow control mechanism. When used on SDA, this is called arbitration and ensures there is only one transmitter at a time.

When idle, both lines are high. To start a transaction, SDA is pulled low while SCL remains high. Releasing SDA to float high again would be a stop marker, signaling the end of a bus transaction. Although legal, this is typically pointless immediately after a start, so the next step is to pull SCL low.

Except for the start and stop signals, the SDA line only changes while the clock is low; transmitting a data bit consists of pulsing the clock line high while holding the data line steady at the desired level.

While SCL is low, the transmitter (initially the master) sets SDA to the desired value and (after a small delay to let the value propagate) lets SCL float high. The master then waits for SCL to actually go high; this will be delayed by the finite rise-time of the SCL signal (the RC time constant of the pull-up resistor and the parasitic capacitance of the bus), and may be additionally delayed by a slave’s clock stretching.

Once SCL is high, the master waits a minimum time (4 μs for standard speed I²C) to ensure the receiver has seen the bit, then pulls it low again. This completes transmission of one bit.

After every 8 data bits in one direction, an “acknowledge” bit is transmitted in the other direction. The transmitter and receiver switch roles for one bit and the erstwhile receiver transmits a single 0 bit (ACK) back. If the transmitter sees a 1 bit (NACK) instead, it learns that:

* (If master transmitting to slave) The slave is unable to accept the data. No such slave, command not understood, or unable to accept any more data.
* (If slave transmitting to master) The master wishes the transfer to stop after this data byte.

During the acknowledgment, SCL is always controlled by the master.

After the acknowledge bit, the master may do one of three things:

* Prepare to transfer another byte of data: the transmitter set SDA, and the master pulses SCL high.
* Send a “Stop”: Set SDA low, let SCL go high, then let SDA go high. This releases the I²C bus.
* Send a “Repeated start”: Set SDA high, let SCL go high, and pull SDA low again. This starts a new I²C bus transaction without releasing the bus.

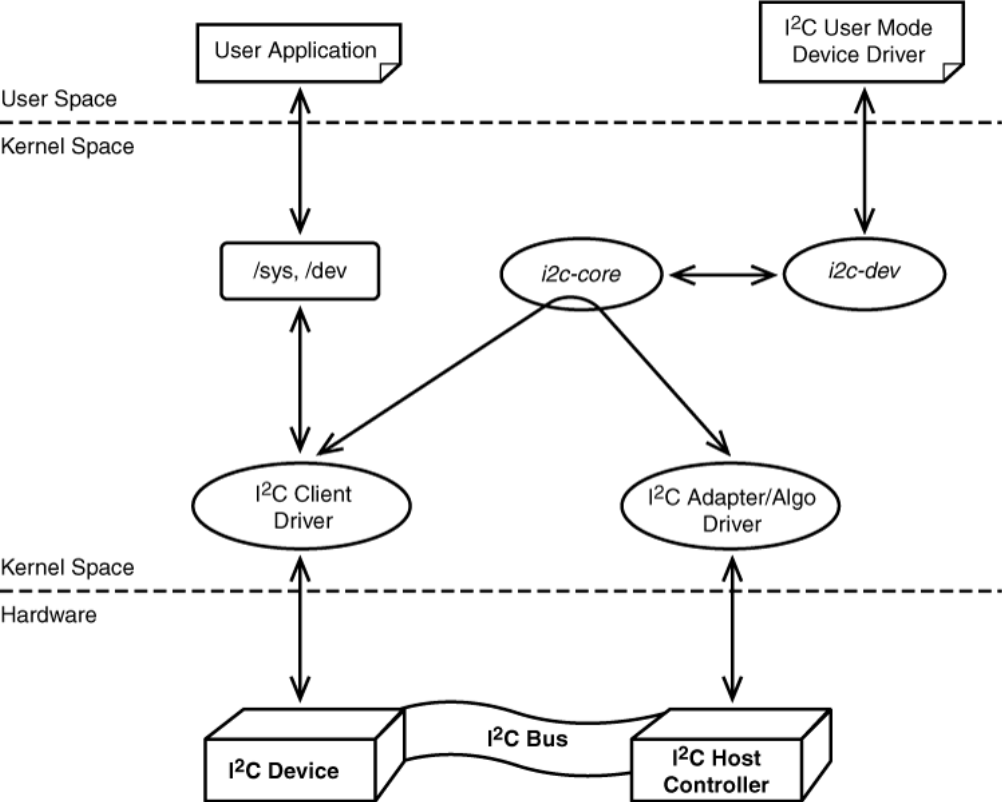
One of the more significant features of the I²C protocol is clock stretching. An addressed slave device may hold the clock line (SCL) low after receiving (or sending) a byte, indicating that it is not yet ready to process more data. The master that is communicating with the slave may not finish the transmission of the current bit, but must wait until the clock line actually goes high. If the slave is clock stretching, the clock line will still be low (because the connections are open-drain). The same is true if a second, slower, master tries to drive the clock at the same time. (If there is more than one master, all but one of them will normally lose arbitration.)

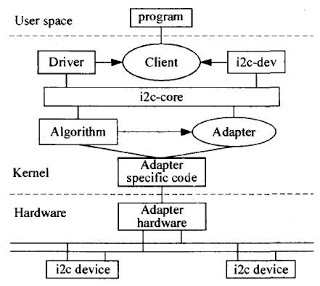
The master must wait until it observes the clock line going high, and an additional minimum time (4 μs for standard 100 kbit/s I²C) before pulling the clock low again.  
Although the master may also hold the SCL line low for as long as it desires, the term “clock stretching” is normally used only when slaves do it. Although in theory any clock pulse may be stretched, generally it is the intervals before or after the acknowledgment bit which are used. For example, if the slave is a microcontroller, its I²C interface could stretch the clock after each byte, until the software decides whether to send a positive acknowledgment or a NACK.

Clock stretching is the only time in I²C where the slave drives SCL. Many slaves do not need to clock stretch and thus treat SCL as strictly an input with no circuitry to drive it.  
I²C is appropriate for peripherals where simplicity and low manufacturing cost are more important than speed. Common applications of the I²C bus are:

* Reading configuration data from EEPROMs on SDRAM, DDR SDRAM, DDR2 SDRAM memory sticks (DIMM) and other stacked PC boards
* Accessing low speed DACs and ADCs.
* Changing contrast, hue, and color balance settings in monitors (Display Data Channel).
* Changing sound volume in intelligent speakers.
* Controlling OLED/LCD displays, like in a cellphone.
* Reading real-time clocks.
* Turning on and turning off the power supply of system components.

**Linux I2C subsystem:**





The kernel I2C subsystem is divided into Buses and Devices, and then further buses into Algorithms and Adapters, and devices into Drivers and Clients.

**Algorithms**  
An Algorithm performs the reading and writing of I2C messages to the hardware, this may involve bit banging GPIO lines or writing to an I2C controller chip. An Algorithm is represented by the structure i2c\_algorithm and allows you to define function pointers to functions that can write I2C messages (master\_xfer) or SMBus messages (smbus\_xfer).

**Adapters**  
An Adapter represents a bus which is used to tie up a particular I2C/SMBus with an algorithm and bus number. It is represented by the structure i2c\_adapter. If you imagine a system where there are many I2C buses – perhaps two controlled by a controller chip and one bit-banged – then you would expect to see 3 instances of an i2c\_adapter and 2 instances of an i2c\_algorithm.

**Clients**  
A Client represents a chip (slave) on the I2C/SMBus such as a Touchscreen, RTC or ADC-DAC peripheral and is represented by a structure i2c\_client. This includes various members such as chip address, name and pointers to the adapter and driver.

**Drivers**  
A driver, represented by a structure i2c\_driver represents the device driver for a particular class of I2C/SMBus slave devices. The structure contains a bunch of function pointers like probe and remove pointers.

A device driver is a piece of software designed to direct control a specific hardware resource using an hardware-independent well defined interface.

The I2C core layer and its attendant benefits is an example of how Linux helps portability. For instance, enabling I2C on a new platform requires only to design the hardware-dependent components, namely the bus driver and the client drivers, whereas the core layer needs not to be changed.

**Initializing and probing of I2C Client Driver**

During initialization the driver registers itself with the I2C core. This is achieved by populating a structure i2c\_driver and passing it as argument to the function i2c\_add\_driver(), as shown in (Listing-1). The structure i2c\_driver holds pointers to the probe and remove functions that are executed respectively on device probing and when the device is removed. The id\_table member of the structure i2c\_driver informs the I2C framework about which slave devices are supported by the driver. In this case the only chip supported is named lis3lv02d. The names of the supported devices are important for binding, as explained next.

1 /\* Device and driver names \*/  
2 # define DEVICE\_NAME “lis3lv02d”  
3  
4/\* I2C client structure \*/  
5 static struct i2c\_device\_id lis3lv02d\_idtable [] = {  
6 { DEVICE\_NAME , 0 },  
7 {}  
8 };  
9 MODULE\_DEVICE\_TABLE (i2c , lis3lv02d\_idtable );  
10  
11 static struct i2c\_driver lis3lv02d\_driver = {  
12 . driver = {  
13 . name = DRIVER\_NAME  
14 },  
15 . probe = lis3lv02d\_probe ,  
16 . remove = \_\_devexit\_p ( lis3lv02d\_remove ),  
17 . id\_table = lis3lv02d\_idtable ,  
18 };  
19  
20 /\* Module init \*/  
21 static int \_\_init lis3lv02d\_init ( void )  
22 {  
23 return i2c\_add\_driver (& lis3lv02d\_driver );  
24 }

***Listing-1: Registration of the LIS3LV02DL driver.***

The binding process consists of associating a device with a driver that can control it. In embedded systems where the number of the I2C bus and the devices connected to it are known for a fact, it is possible to declare in advance the I2C slaves which live on the bus. This is typically done in the board setup file (Listing-2).

The mysoc\_platform\_init function is executed on board startup and, among other tasks, registers the I2C slave devices by invoking the i2c\_register\_board\_info function with arguments that specify the number of the bus (zero in this case) and the devices connected with it. This is done through an array of structure i2c\_board\_info(), each item of which specifies the device name and the device address, with the former that must match with the name registered by the driver in order for binding to succeed. In this case structure i2c\_board\_info holds only one item which corresponds to the LIS3LV02DL inertial sensor.

Since the sensor’s chip has an interrupt line tied to the cpu, the irq member is also specified with the respective IRQ number. By means of another member called platform\_data it is possible to define custom data for the driver.

1 /\* I2C devices \*/  
2 static struct i2c\_board\_info mysoc\_i2c\_devices [] = {  
3 {  
4 I2C\_BOARD\_INFO (” lis3lv02d “, 0 x1D ),  
5 . irq = MYSOC\_GPIO\_TO\_IRQ (82) ,  
6 /\* No platform data : use driver defaults \*/  
7 },  
8 };  
9  
10 static void \_\_init mysoc\_platform\_init ( void )  
11 {  
12 …  
13 /\* Register I2C devices on bus #0 \*/  
14 i2c\_register\_board\_info (0, mysoc\_i2c\_devices ,  
15 ARRAY\_SIZE ( mysoc\_i2c\_devices ));  
16 …  
17 }

***Listing 2: Registration of the I2C devices.***

During boot the kernel looks for any I2C driver that has registered a matching device name, that is “lis3lv02d”. Upon finding such a driver, the kernel invokes its probe() function passing a pointer to the LIS3LV02DL device as a parameter. This process is called probing. The probe function is responsible for the per-device initialization, that is initializing hardware, allocating resources, and registering the device with any appropriate subsystem.

More in detail, the LIS3LV02DL probe function takes the following actions:

1. Allocate memory for lis3lv02d\_priv private data structure.
2. Load the device settings.
3. Identify the LIS3LV02DL chip.
4. Configure the device hardware.
5. Create the per-device sysfs nodes.
6. If the free-fall feature is enabled, request the interrupt and register the IRQ for the free-fall detection.
7. If the device polling feature is enabled, register the device with the input subsystem.

On successful completion of all the above steps, meaning a successful probing, the device is bound to the driver.

**Initializing and probing of I2C Bus Driver**

Initializing and probing the dummy MYSOC I2C bus driver is performed in a similar way as for LIS3LV02DL client driver, with the major difference being that the former uses a platform bus. The platform bus requires that any I2C adapter (or equivalently controller), which is controlled by the bus driver, be registered using a platform\_device structure. This structure represents the bus adapter and provides information such as the device name, the device resources and the adapter number, to the bus driver.

Usually the registration of the I2C adapters with the platform bus is performed by the board initialization file, as the information needed are highly board specific. Listing-3 shows the part relevant to this matter.

1 /\* first bus : i2c0 \*/  
2 static struct platform\_device mysoc\_i2c\_dev0 = {  
3 .name = “mysoc\_i2c “,  
4 .id = 0,  
5 . resource = &mysoc\_i2c\_resources [0] ,  
6 . num\_resources = 2,  
7 . dev = {  
8 . platform\_data = &mysoc\_i2c\_dev0\_data ,  
9 },  
10 };  
11  
12 /\* second bus: i2c1 \*/  
13 static struct platform\_device mysoc\_i2c\_dev1 = {  
14 . name = “mysoc\_i2c”,  
15 .id = 1,  
16 . resource = &mysoc\_i2c\_resources [2] ,  
17 . num\_resources = 2,  
18 /\* No platform data : use driver defaults \*/  
19 };  
20  
21 static int \_\_init mysoc\_i2c\_init ( void )  
22 {  
23 …  
24 platform\_device\_register (&mysoc\_i2c\_dev0 );  
25 …  
26 platform\_device\_register (&mysoc\_i2c\_dev1 );  
27 …  
28 }  
29 arch\_initcall ( mysoc\_i2c\_init );

***Listing 3: Registration of the I2C platform device with the platform bus.***

On the driver’s side, the registration with the platform bus is achieved by populating a structure platform\_driver and passing it to the macro module\_platform\_driver() as argument (Listing-4). The platform bus simply compares the driver.name member against the name of each device, as defined in the platform\_device data structure (Listing 3); if they are the same the device matches the driver.

1 # define DRIVER\_NAME “mysoc\_i2c”  
2  
3static struct platform\_driver mysoc\_i2c\_driver = {  
4 . probe = mysoc\_i2c\_probe ,  
5 . remove = \_\_devexit\_p ( mysoc\_i2c\_remove ),  
6 . driver = {  
7 . name = DRIVER\_NAME ,  
8 . owner = THIS\_MODULE ,  
9 .pm = &mysoc\_i2c\_pm\_ops ,  
10 },  
11 };  
12 module\_platform\_driver ( mysoc\_i2c\_driver );

***Listing 4: Registration of the I2C platform driver with the platform bus.***

As usual, binding a device to a driver involves calling the driver’s probe() function passing a pointer to the device as a parameter.

The sequence of operations performed on probing are the following:

1. Get the device resource definitions.
2. Allocate the appropriate memory and remap it to a virtual address for being accessed by the kernel.
3. Load the device settings.
4. Configure the device hardware.
5. Register with the power management system.
6. Create the per-device sysfs nodes.
7. Request the interrupt and register the IRQ.
8. Set up the struct i2c\_adapter and register the adapter with the I2C core.

On successful completion of above steps the driver is bounded to the devices representing the two mysoc I2C controllers.

In the Linux I2C subsystem an I2C bus driver consists of an adapter driver and an algorithm driver. This division is to improve the software reuse and to allow portability. An algorithm driver is intended to contain general code that can be used for a whole class of I2C adapters, while each specific adapter driver either depends on one algorithm driver, or includes its own implementation.

However, while having a generic algorithm that works for multiple adapters is suitable for many cases, in embedded systems, where each I2C bus adapter has its own way of interfacing with the processor and the bus, it is usual to develop the adapter driver together with its corresponding algorithm driver. The bus driver registers with the I2C subsystem by using a structure i2c\_adapter that is instantiated and initialized by the I2C platform driver’s probe() function, as shown in Listing 5.

The i2c\_adapter structure’s algo member is set up to point to a structure i2c\_algorithm which in turn holds two pointers:

* master\_xfer points to the function that implements the actual I2C transmit and receive algorithm.
* functionality points to a function that returns the features supported by the I2C adapter.

To communicate with a client the I2C subsystem offers two class of functions: one for I2C plain communication which includes i2c\_master\_send(), i2c\_master\_recv() and i2c\_transfer(), and a second one that uses SMBus commands. However, whichever method is used, the data transfer relies on the bus driver’s function pointed to by master\_xfer, as the I2C core ultimately calls this function for the actual transfer to take place.

1 /\* I2C algorithm structure \*/  
2 static struct i2c\_algorithm mysoc\_i2c\_algo = {  
3 . master\_xfer = mysoc\_i2c\_xfer ,  
4 . functionality = mysoc\_i2c\_func ,  
5 };  
6  
7/\* Probe function \*/  
8 static int \_\_devinit mysoc\_i2c\_probe  
9 ( struct platform\_device \* pdev )  
10 {  
11 …  
12 adap = kzalloc ( sizeof ( struct i2c\_adapter ),  
13 GFP\_KERNEL );  
14 …  
15 adap -> algo = &mysoc\_i2c\_algo ;  
16 …  
17 err = i2c\_add\_numbered\_adapter ( adap );  
18 …  
19 }

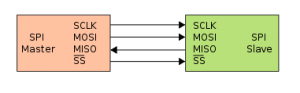
***Listing 5: Registration of the I2C adapter.***

As shown in listing 5, mysoc\_i2c\_xfer() is the transfer function installed by the I2C bus driver. This function receives an array of messages as argument and processes them in sequence by calling mysoc\_i2c\_xfer\_rd() or mysoc\_i2c\_xfer\_wr() depending on whether the message being processed is marked for read or write. Once all messages have been sent mysoc\_i2c\_xfer() successfully returns, otherwise, upon detecting a communication error, aborts the transmission and returns an appropriate error code.

<https://pt.slideshare.net/varunmahajan06/i2c-subsystem-in-linux2624/7>

# SPI Protocol

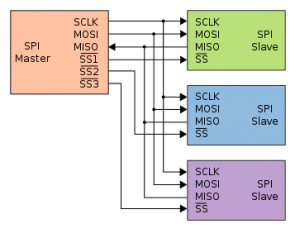
<https://hackaday.com/2016/07/01/what-could-go-wrong-spi/>

[](http://invo-tronics.com/wp-content/uploads/2014/09/SPI_single_slave.png)The Serial Peripheral Interface or SPI Bus is a four wire master/slave full duplex synchronous bus named by Motorola. Multiple slave devices can be hooked by utilizing chip select lines. It is used for short distance, single master communication, for example in embedded systems, sensors, and SD cards.

The bus is composed of two data pins, one clock pin, and one chip select pin:

* SCLK – Serial Peripheral Interface Clock Signal (generated by the master) (also referred to as SCK)
* MOSI – Master Out Slave In data (output from the master)
* MISO – Master In Slave Out (output from the slave)
* CS – Chip Select (also referred to as Slave Select (SS))

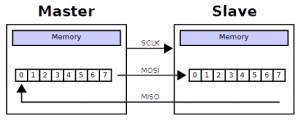
The SPI bus can operate with a single master device and with one or more slave devices. If a single slave device is used, the SS pin may be fixed to logic low if the slave permits it. Some slaves require a falling edge of the chip select signal to initiate an action, an example is the Maxim MAX1242 ADC, which starts conversion on a high→low transition.

[](http://invo-tronics.com/wp-content/uploads/2014/09/SPI_three_slaves.png)With multiple slave devices, an independent SS signal is required from the master for each slave device. Most slave devices have tri-state outputs so their MISO signal becomes high impedance (logically disconnected) when the device is not selected. Devices without tri-state outputs cannot share SPI bus segments with other devices; only one such slave could talk to the master, and only its chip select could be activated.

Each slave may operate at different clock frequencies as well as different clock polarities and clock phases with respect to the data. The permutations of polarities and phases are referred to as SPI modes. Before beginning the communication, the bus master first configures the clock and the modes. The master then transmits the logic 0 for the desired chip over the chip select line. If a waiting period is required (such as for analog-to-digital conversion), then the master must wait for at least that period of time before starting to issue clock cycles.

During each SPI clock cycle, a full duplex data transmission occurs:

* the master sends a bit on the MOSI line;the slave reads it from that same line
* the slave sends a bit on the MISO line; the master reads it from that same line

[](http://invo-tronics.com/wp-content/uploads/2014/09/SPI_8-bit_circular_transfer.png)Transmissions normally involve two shift registers of some given word size, such as eight bits, one in the master and one in the slave; they are connected in a ring. Data is usually shifted out with the most significant bit first, while shifting a new least significant bit into the same register. After that register has been shifted out, the master and slave have exchanged register values. Then each device takes that value and does something with it, such as writing it to memory. If there is more data to exchange, the shift registers are loaded with new data and the process repeats.

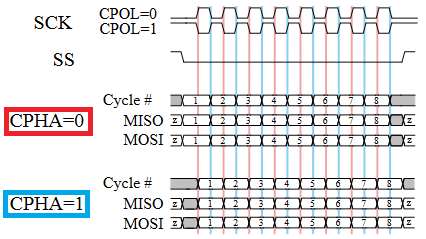
Transmissions may involve any number of clock cycles. When there is no more data to be transmitted, the master stops toggling its clock. Normally, it then deselects the slave. Transmissions often consist of 8-bit words. A master can initiate multiple such transmissions if it wishes/needs. However, other word sizes are also common, such as 16-bit words for touchscreen controllers or audio codecs or 12-bit words for many digital-to-analog or analog-to-digital converters. Every slave on the bus that has not been activated using its chip select line must disregard the input clock and MOSI signals, and must not drive MISO. The master must select only one slave at a time.

The timing diagram is shown below. The timing is further described below and applies to both the master and the slave device.  
At CPOL=0 the base value of the clock is zero

* For CPHA=0, data are captured on clock’s rising edge and data is propagated on a falling edge.
* For CPHA=1, data are captured on clock’s falling edge and data is propagated on a rising edge.

At CPOL=1 the base value of the clock is one (inversion of CPOL=0)

* For CPHA=0, data are captured on clock’s falling edge and data is propagated on a rising edge.
* For CPHA=1, data are captured on clock’s rising edge and data is propagated on a falling edge.

[](http://invo-tronics.com/wp-content/uploads/2014/09/SPI_timing_diagram.png)The data must be stable for a half cycle before the first clock cycle. The MOSI and MISO signals are usually stable (at their reception points) for the half cycle until the next clock transition. SPI master and slave devices may well sample data at different points in that half cycle.

The full-duplex capability makes SPI very simple and efficient for single master/single slave applications. Some devices use the full-duplex mode to implement an efficient, swift data stream for applications such as digital audio, digital signal processing, or telecommunications channels, but most off-the-shelf chips stick to half-duplex request/response protocols.

SPI is used to talk to a variety of peripherals, such as

* Sensors: temperature, pressure, ADC, touchscreens, video game controllers
* Control devices: audio codecs, digital potentiometers, DAC
* Camera lenses: Canon EF lens mount
* Communications: Ethernet, USB, USART, CAN, IEEE 802.15.4, IEEE 802.11, handheld video games
* Memory: flash and EEPROM
* Real-time clocks
* LCD, sometimes even for managing image data
* Any MMC or SD card (including SDIO variant)

# SPI Driver for Linux Based Embedded System

Today Linux is the operating system choice for a wide range of special-purpose electronic devices known as embedded systems. An embedded system is specifically designed to perform a set of designated activities, and it generally uses custom, heterogeneous processors. This makes Linux a flexible operating system capable of running on a variety of architectures, such as ARM and many others.

Linux has highly modular architecture and it facilitates the porting and a lot of efforts are required to build new kernel components to fully support the target platform. A big part of these efforts are in developing the low-level interfaces commonly referred to as device drivers. A device driver is a piece of software designed to direct control a specific hardware resource using an hardware-independent well defined interface.

The kernel SPI subsystem is divided into Controller Driver and Protocol Drivers.

**Controller Driver**  
A controller driver is represented by the structure spi\_master. The driver for an SPI controller manages access to spi slave devices through a queue of spi\_message transactions, copying data between CPU memory and an SPI slave device. For each such message it queues, it calls the message’s completion function when the transaction completes.

**Protocol Driver**  
A Protocol driver is represented by the structure spi\_driver, they pass messages through the controller driver to communicate with a Slave or Master device on the other side of an SPI link. For example one protocol driver might talk to the MTD layer to export data to file systems stored on SPI flash like Data Flash; and others might control audio interfaces, present touchscreen sensors as input interfaces, or monitor temperature and voltage levels during industrial processing. And those might all be sharing the same controller driver.

**Initializing and probing of SPI Controller Driver**

For embedded System-on-Chip (SOC) based boards, SPI master controllers connect to their drivers using some non SPI bus, such as the platform bus. Initializing and probing SPI controller driver is performed using the platform bus. During the initial stage of platform driver registration stage of probe() in that code includes calling spi\_alloc\_master which allocated the structure spi\_master in the kernel and during final stage calling spi\_register\_master() to hook up to this SPI bus glue.

SPI controller’s will usually be platform devices, and the controller may need some platform\_data in order to operate properly. The “struct platform\_device” will include resources like the physical address of the controller’s first register and its IRQ.

Platforms will often abstract the “register SPI controller” operation,maybe coupling it with code to initialize pin configurations in the board initialization files. This is because most SOCs have several SPI-capable controllers, and only the ones actually usable on a given board should normally be set up and registered.

1 /\* spi bus : spi0 \*/  
2 static struct platform\_device mysoc\_spi\_dev0 = {  
3 .name = “mysoc\_spi”,  
4 .id = 0,  
5 . resource = &mysoc\_spi\_resources [0] ,  
6 . num\_resources = 2,  
7 . dev = {  
8 . platform\_data = &mysoc\_spi\_dev0\_data ,  
9 },  
10 };  
11  
21 static int \_\_init mysoc\_spi\_init ( void )  
22 {  
23 …  
24 platform\_device\_register (&mysoc\_spi\_dev0 );  
25 …  
28 }  
29

**Listing 1: Registration of the SPI platform device with the platform bus.**

On the driver’s side, the registration with the platform bus is achieved by populating a structure platform\_driver and passing it to the macro module\_platform\_driver() as argument (Listing-2). The platform bus simply compares the driver.name member against the name of each device, as defined in the platform\_device data structure (Listing 1); if they are the same the device matches the platform driver.

1 # define DRIVER\_NAME “mysoc\_spi”  
2  
3 static struct platform\_driver mysoc\_spi\_driver = {  
4 .probe = mysoc\_spi\_probe ,  
5 .remove = mysoc\_spi\_remove,  
6 .driver = {  
7 .name = DRIVER\_NAME ,  
8 .owner = THIS\_MODULE ,  
9 .pm = &mysoc\_spi\_pm\_ops ,  
10 },  
11 };  
12 module\_platform\_driver ( mysoc\_spi\_driver );

**Listing 2: Registration of the SPI platform driver with the platform bus.**

As usual, binding a device to a driver involves calling the driver’s probe() function passing a pointer to the device as a parameter. The SPI controller driver registers with the SPI subsystem by using a structure spi\_master that is instantiated and initialized by the SPI platform driver’s probe() function, as shown in Listing 3.

The sequence of operations performed on probing are the following:

1. Get the device resource definitions.
2. Allocate the appropriate memory and remap it to a virtual address for being accessed by the kernel.
3. Load the device settings.
4. Configure the device hardware.
5. Register with the power management system.
6. Create the per-device sysfs nodes.
7. Request the interrupt and register the IRQ.
8. Set up the struct spi\_master and register the master controller driver with the SPI core.

On successful completion of above steps the driver is bounded to the devices representing the mysoc SPI controllers.

1 /\* Probe function \*/  
2 static int mysoc\_spi\_probe ( struct platform\_device \* pdev )  
3 {  
4 …  
5 struct spi\_master \*master;  
6 …  
7 /\* Allocate master \*/  
8 master = spi\_alloc\_master(&pdev->dev, sizeof(struct spi\_master));  
9 …  
10 /\* Register with the SPI framework \*/  
11 status = spi\_register\_master(master);  
12 …  
14 }

**Listing 3: Registration of the SPI Controller Driver.**

Then it will scan the platform data to find the SPI devices connected to this SPI bus. The function scan\_boardinfo() scans the platform data, and call spi\_new\_device() to create SPI device data structure, and set up struct spi\_device based on the platform information. Then it calls the master’s setup() method to further initialize, link the struct spi\_device with struct spi\_master, and add the SPI device to the system. To this point, the SPI master and SPI device are created and added to the system. But it still can’t communicate with the SPI device as no specific driver is installed yet.

**Initializing and probing of SPI Protocol Driver**

SPI Protocol driver’s deal with the spi chip attached to the SPI controller. These drivers are responsible to send/receive to/from the device. These device drivers expose user-level API (like spidev does) or kernel-level API that can be used by another subsystem. For example, the touch controller chip ADS7846, which is connected to the SPI bus, provides a touch interface and connects to input subsystem to generate input events.

SPI board information is part of the machine-depended code that performs registration of SPI devices with the SPI subsystem. Because SPI devices are usually hardwired to the board and rarely have an ability to enumerate them, they have to be hardcoded in machine board file in the Linux kernel. The board-dependent code does the registration by calling the function spi\_register\_board\_info. It takes two parameters: list of devices connected and the size of this list (Listing-4).

1 /\* SPI Device \*/  
2 static struct ads7846\_platform\_data ads\_info = {  
3 .vref\_delay\_usecs = 100,  
4 .x\_plate\_ohms = 580,  
5 .y\_plate\_ohms = 410,  
6 };  
7 static struct spi\_board\_info mysoc\_spi\_devices [] \_\_initdata = {  
8 {  
9 .modalias = “ads7846”,  
10 .platform\_data = &ads\_info,  
11 .mode = SPI\_MODE\_0,  
12 .irq = GPIO\_IRQ(31),  
13 .max\_speed\_hz = 120000,  
14 .bus\_num = 1,  
15 .chip\_select = 0,  
16 },  
17 };  
18 static void \_\_init mysoc\_platform\_init ( void )  
19 {  
20 …  
21 /\* Register SPI devices on bus #0 \*/  
22 spi\_register\_board\_info (mysoc\_spi\_devices, ARRAY\_SIZE ( mysoc\_spi\_devices ));  
23 …  
24 }

**Listing 4: Registration of the SPI devices.**

During initialization the driver registers itself with the SPI core. This is achieved by populating a structure spi\_driver and passing it as argument to the function spi\_register\_driver(), as shown in Listing-5.

1 static struct spi\_driver ads7846\_driver = {  
2 . driver = {  
3 . name = “ads7846”,  
4 .owner=THIS\_MODULE,  
5 },  
6 . probe = ads7846\_probe ,  
7 . remove = ads7846\_remove ),  
8 };  
9  
10 /\* Module init \*/  
11 static int \_\_init ads7846\_init ( void )  
12 {  
13 return spi\_register\_driver (& ads7846\_driver );  
14 }

**Listing-5: Registration of the ADS7846 driver.**

The structure spi\_driver holds pointers to the probe and remove functions that are executed respectively on device probing and when the device is removed. The names of the supported devices are important for binding.

During boot the kernel looks for any SPI driver that has registered a matching device name, that is “ads7846”. Upon finding such a driver, the kernel invokes its probe() function passing a pointer to the ADS7846 device as a parameter. This process is called probing. The probe function is responsible for the per-device initialization, that is initializing hardware, allocating resources, and registering the device with any appropriate subsystem.

More in detail, the ADS7846 probe function takes the following actions:

1. Allocate memory for spi\_transfer and spi\_message data structure.
2. Load the device settings.
3. Configure the device hardware.
4. Create the per-device sysfs nodes.
5. If the device interrupt feature is enabled, request the interrupt and register the IRQ.
6. If the device polling feature is enabled, register the device with the input subsystem.

On successful completion of all the above steps, meaning a successful probing, the device is bound to the driver.

<https://wiki.analog.com/software/linux/docs/iio/iio>

# Linux Industrial I/O Subsystem

## **IIO Overview**

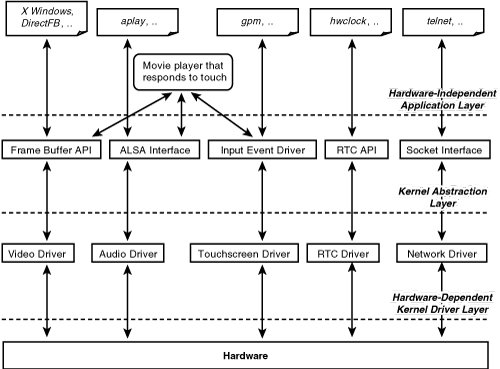
* The Industrial I/O subsystem is intended to provide support for devices that in some sense are analog to digital or digital to analog converters (ADCs, DACs).
* Devices that fall into this category are:
  + ADCs
  + Accelerometers
  + Gyros
  + IMUs
  + Capacitance to Digital Converters (CDCs)
  + Pressure Sensors
  + Color, Light and Proximity Sensors
  + Temperature Sensors
  + Magnetometers
  + DACs
  + DDS (Direct Digital Synthesis)
  + PLLs (Phase Locked Loops)
  + Variable/Programmable Gain Amplifiers (VGA, PGA)
* The overall aim is to fill the gap between the somewhat similar hwmon and input subsystems.
* Hwmon is very much directed at low sample rate sensors used in applications such as fan speed control and temperature measurement.
* Input is, as it's name suggests focused on human interaction input devices.:
  + Keyboard
  + Mouse
  + Touch Screen
  + Joystick
* In some cases there is considerable overlap between these and IIO.
* A typical device falling into the IIO category would be connected via SPI or I2C.
* However typical DMA operated devices such as ones connected to a high speed synchronous serial (McBSP, SPORT) or high speed synchronous parallel (EPI, PPI) or FPGA peripherals are also subject to this subsystem.
* Since latter ones unlike SPI or I2C are not generally abstracted by Linux bus drivers they are subject to processor platform dependent implementations.

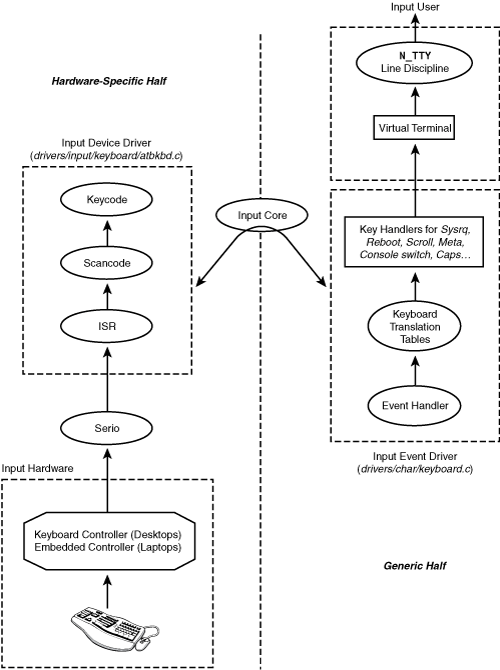
## **IIO Subsystem Overview**

<http://sagaralatgi.blogspot.sg/2013/04/linux-input-subsystem-kernels-input.html>

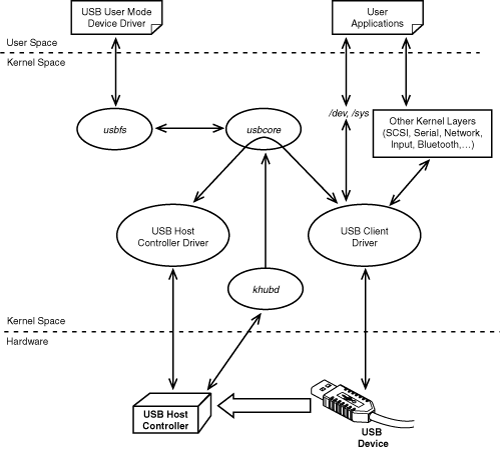
|  |  |
| --- | --- |
| http://1.bp.blogspot.com/-Ul3ETMYH6fg/UXvlZhE4RkI/AAAAAAAAADU/Yo-kEPdypi8/s1600/YTNyaWQ3ODBzOS9jL2VnbXRwNjQ5NWEzMi9yZzE1ZmkwcGdmaS4wLzFoaWc3c2M-.jpg | Provides support for using GPIO pins as IIO triggers. |
|  |  |
|  |  |

# http://www.programering.com/images/remote/ZnJvbT1jbmJsb2dzJnVybD1jR2NxNUNNMU1UTzRZVE0yRVROeVlUTXlFekwwQUROeEFqTXZJRE54WXpOejh5WnZ4bVl2MDJiajV5WnZ4bVkwbG1iajV5Y2xkV1l0bDJMdm9EYzBSSGE.jpg





**Linux USB subsystem:**



BOOT Mode:

MMC/SD

Emmc

UART

NOR flash

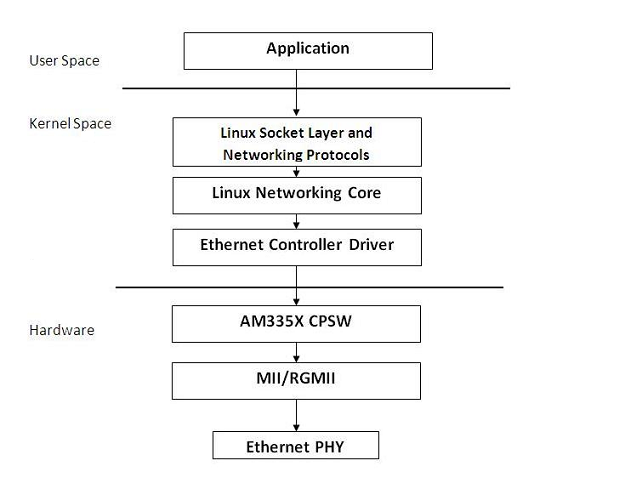
USB boot

QSPI flash

EMAC Boot

| **Peripheral Driver Support** | | | |
| --- | --- | --- | --- |
| **Peripheral** | **Description** | **Linux driver type** | **DMA usage** |
| Audio (McASP) | Audio Record and Playback | ALSA SoC | sDMA / eDMA |
| Ethernet | Ethernet Network driver | Netdev | Internal DMA |
| USB1 DWC3(DRD)-SS/HS/FS/LS | DWC3 Device & xhci host controller driver | USB HCD/DCD | USB Internal DMA |
| USB2 DWC3(DRD)-HS/FS/LS | DWC3 Device & xhci host controller driver | USB HCD/DCD | USB Internal DMA |
| eMMC/SD/MMC | Interface to MultiMedia Secure Digital cards | Block | SDMA |
| UART | Serial Communication Interface | Character | Supported |
| I2C | Inter-IC Communication | Character | Not Supported |
| DSS | Display Subsystem driver | Platform driver | Internal DMA |
| VIP | Video IP driver | V4L2 Capture | VPDMA |
| VPE | Video Processing Engine driver | V4L2 Mem to Mem | VPDMA |
| CPUFreq | Supports multiple SoC operating levels for MPU(OPPs) | NA | None |
| RTC | Realtime clock | Character | None |
| Watchdog | Watchdog Timer | Miscellaneous | None |
| SPI (/dev/spi) | Serial Peripheral Interface | Character | Not Supported |
| Touchscreen | Touchscreen driver | Input driver | None |
| NAND Flash | Flash storage system | MTD Character and Block | Not Supported |
| GLCD | Graphical LCD driver | Frame Buffer | LCDC Internal DMA |

## Ethernet Driver



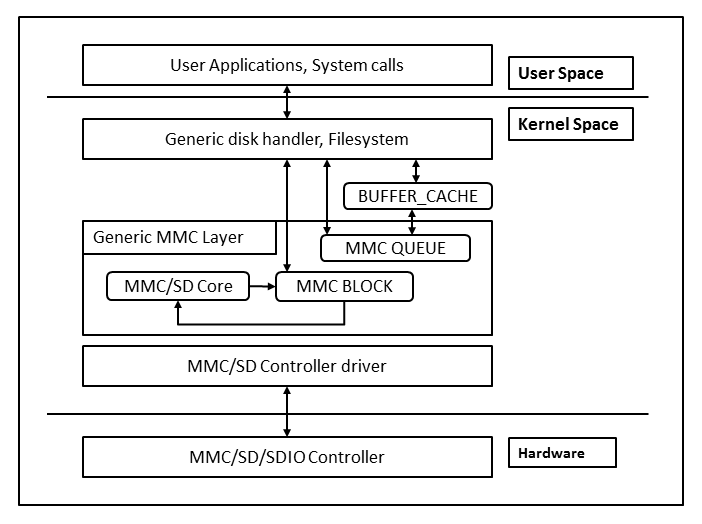
### Driver Features

The driver supports the following features:

1. 10/100/1000 Mbps mode of operation.
2. Auto negotiation.
3. Full duplex and half duplex mode of operation.
4. Linux NAPI support
5. Support for MII and RGMII interfaces to PHY
6. This driver uses Timer 5 & 6 for CPSW Interrupt Pacing. Sharing this Timer with any other module will result in Ethernet pacing not working properly.

## MMC/SD Driver

The MMC controller provides an interface to external MMC cards. The MMC driver is implemented as a block driver. Block device nodes(such as /dev/mmcblockp1, /dev/mmcblockp2) are created for user space access.



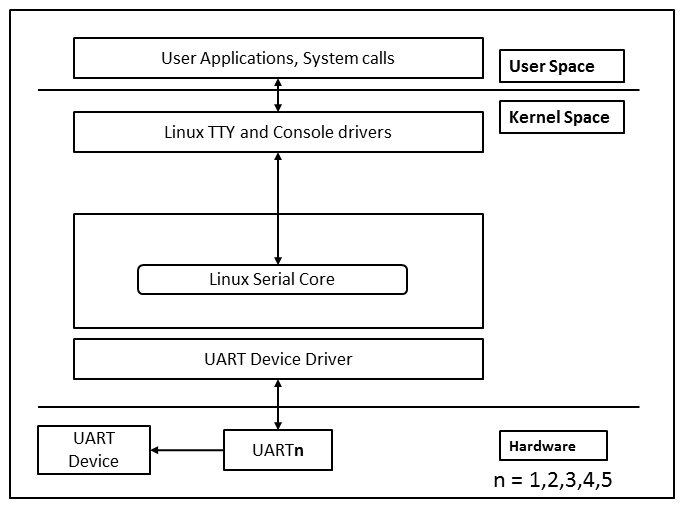
### Driver Features

The driver supports the following features:

1. MMC/SD native protocol command/response set
2. Single/multiple block data transfers
3. Linux file system and generic MMC layer abstract details of block devices (MMC)
4. High-speed (SDv1.1) and High Capacity (SDv2.0) cards
5. Support for 4 bit modes
6. Support for card detect and Write protect features
7. DMA and polled mode for data transfer operations

## UART Driver

The UART driver is implemented as a serial driver, and can be accessed from user space as /dev/ttyOX(X=0-5)



### Driver Features

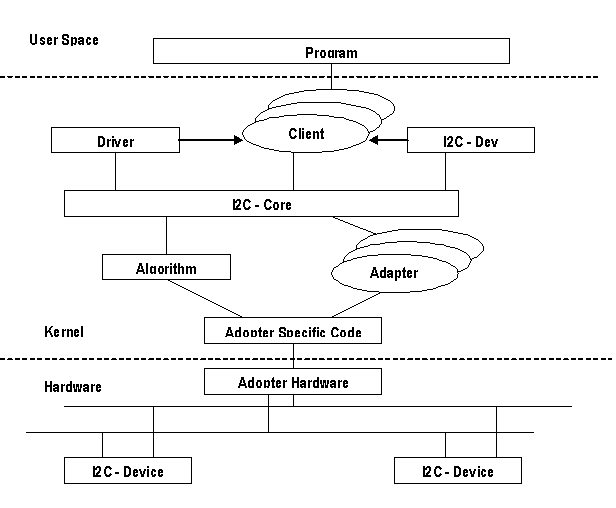
* Both Peripheral mode and DMA mode are supported

### Features Not Supported

* Hardware Flow Control due to board limitation
* UART DMA mode supported but not validated at baud rates greater than 115200 due to HW limitation mentioned above (Flow control).
* Baud rate of 12 Mbps was not validated.

## I2C Driver

The I2C peripheral is compliant with the Philips Semiconductor I2C-bus specification version 2.1. The I2C driver is implemented as a serial driver. The I2C driver can be accessed from the user space as /dev/i2c/0.



### Driver Features

The driver supports the following features:

1. 7-bit addressing mode
2. Fast mode
3. Interrupt mode

### Features Not Supported

1. 7-bit and 10-bit addressing combined format is not supported
2. DMA mode is not supported

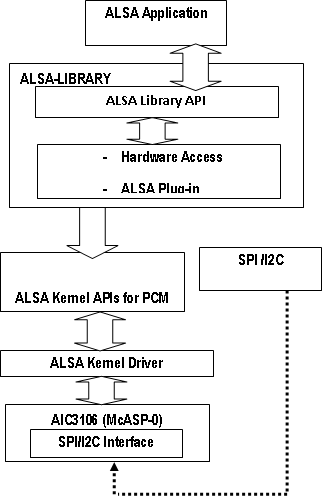
## ALSA SoC Audio Driver

AM335x Audio driver complies to the Advanced Linux Sound Architecture (ALSA) System on Chip (SoC) framework (ASoC).

The ASoC framework splits an embedded audio system into three components:

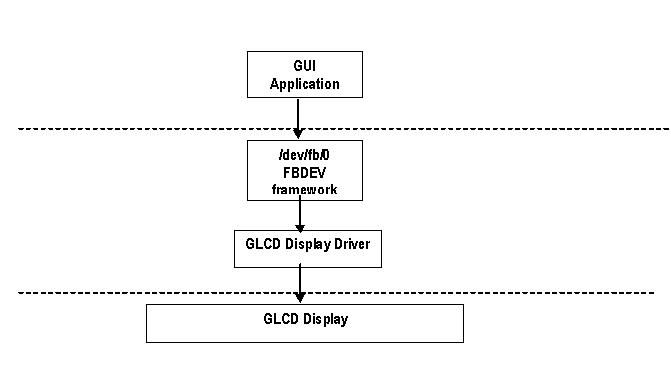
* **Codec driver:** The codec driver is generic and hardware independent code that configures the audio codec to provide audio capture and playback. It should contain no code that is specific to the target platform or machine.
* **Platform driver:** The platform driver can be divided into audio DMA and SoC Digital Audio Interface (DAI) configuration and control. The platform driver only targets the SoC CPU and must have no board specific code.
* **Machine driver:** The ASoC machine (or board) driver is the code that glues together the platform and codec drivers. It can contain codec and platform specific code. It registers the audio subsystem with the kernel as a platform device.

Following architecture diagram shows all the components and the interactions among them.



## Graphical LCD (GLCD) Driver

GLCD driver is based on Fbdev framework and it reuses da8xx-fb driver as LCDC IP is upgraded version of that found on OMAP-L138 SoC.



### Driver Features

The driver supports the following features:

1. Supports WVGA display through Fbdev framework.
2. Supports display of ARGB:8888 images at max resolution 2048\*2048, but pixel clock can be driven Max 150MHz.
3. Supports getting and setting the variable screen information.
4. Supports retrieving the fixed screen information.

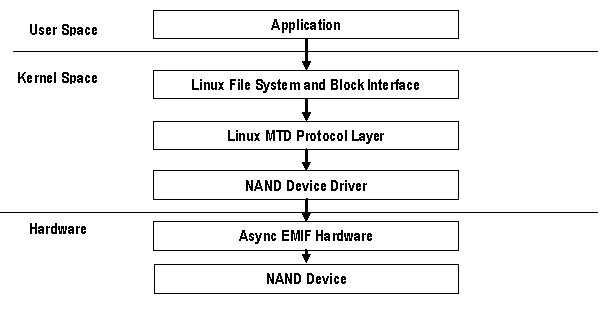
### Features Not Supported

1. Brightness and color control ioctls not supported.

## NAND Driver

The NAND driver is implemented as a character and block driver, compliant with the Linux MTD subsystem interface. It supports various NAND Flash chips (see file *drivers/mtd/nand/nand\_ids.c* in Linux kernel sources). The NAND driver creates the device nodes for user space access (/dev/mtdblock0, /dev/mtdblock1, /dev/mtd0,/dev/mtd1 and so on.).

This figure illustrates the stack diagram of NAND flash driver in Linux.



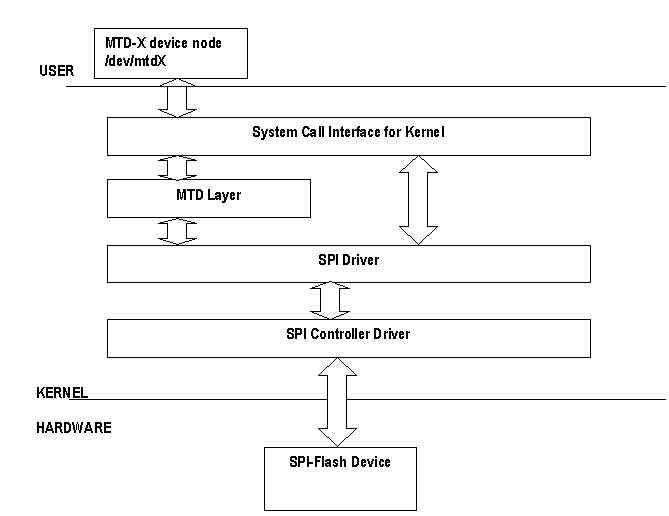
### Driver Features

The driver supports the following features:

1. JFFS2 file system support
2. Supports Read, Write and Erase
3. Bad Block Management
4. Polled Prefetch mode of transfer (enabled by default)
5. SLC NAND

## SPI Flash Driver

SPI Flash driver is implemented as block driver and compliant with standard MTD driver. It supports various flash devices. The SPI driver creates device node for user space access (example, /dev/mtd1)



### Driver Features

The driver supports the following features:

1. PIO Mode of Operation are supported.

### Features Not Supported

1. DMA Mode of Operation not supported.

## Touchscreen Driver

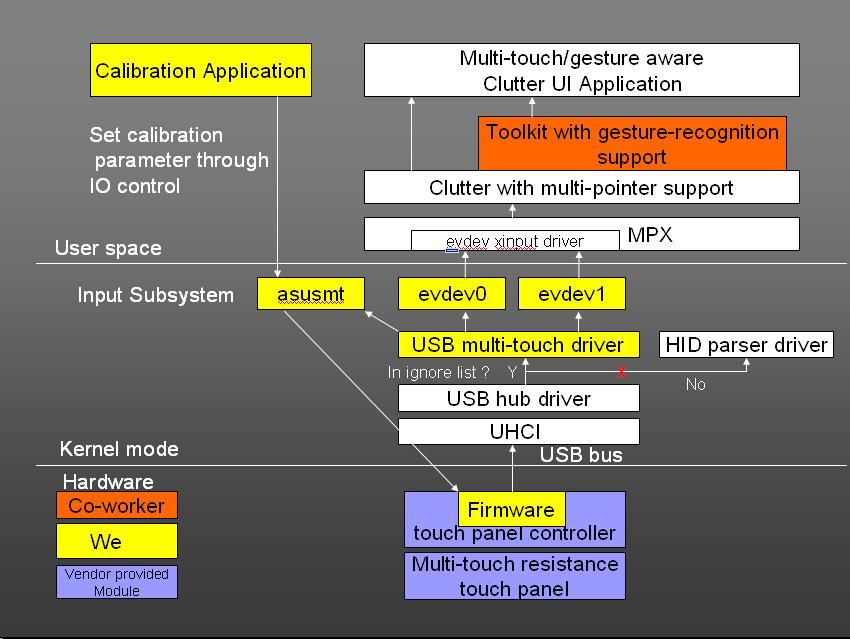
### Introduction

Touchscreen controller on AM335x is an 8 channel general purpose ADC, with optional support for interleaving Touch Screen conversions for a 4-wire, 5-wire, or 8-wire resistive panel. A resistive touchscreen operates by applying a volatge across a resistive network and measuring the change in resistance at a given point on the matrix where the screen is touched by an input (stylus or finger). The change in the resistance ratio marks the location on the touchscreen.

### Driver Features

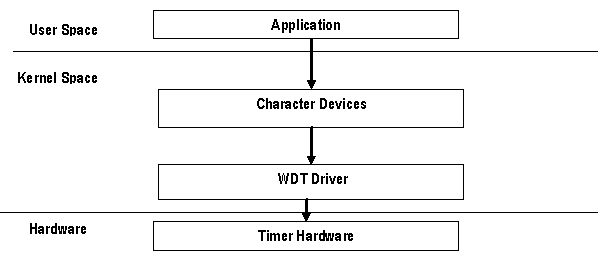
The driver supports the following features:

* Resistive touchscreen
* Hardware synchronized one shot mode
* Programmable Open delay/Sampling delay before sampling each channel
* Programmable averaging of input samples - 16
* Support for servicing FIFOs via CPU.
* ADC sampled data is 12-bit wide.



## Watchdog(WDT) Driver

AM335x has a 32-bit watchdog timer which can be used to reset the hardware in case of a software fault. Once the /dev/watchdog is opened, it will reboot the system unless a user space daemon resets the timer at regular intervals within a certain timeout period. If watchdog device node is closed before timeout also reboot won't happen. The WDT driver is registered as a misc device. Default timeout of this driver is 60 seconds.

[](http://processors.wiki.ti.com/index.php/File:Wdt.png)

### Driver Features

The driver supports the following features:

1. Supports IOCTLs to set/get the timeout value, ping the watchdog & query the watchdog structure info.
2. Driver can be built as a loadable module and inserted dynamically.

## USB Driver

### MUSB OTG controller

#### Description

The MUSB driver is implemented on top of Mentor OTG IP version 2.0 which supports all the speeds (High, Full and Low (host mode only)). On AM335x, MUSB uses CPPI 4.1 DMA for all the transfers.

#### Driver Features

The driver supports the following features:

**Host Mode**

1. Human Interface Class (HID)
2. Mass Storage Class (MSC)
3. Hub Class
4. USB Video Class (UVC)
5. USB Audio Class (UAC)
6. USB CDC HOST (USBNET)

**Gadget mode**

1. Mass Storage Class (MSC)
2. USB Networking - RNDIS/CDC

#### Features Not Supported

OTG

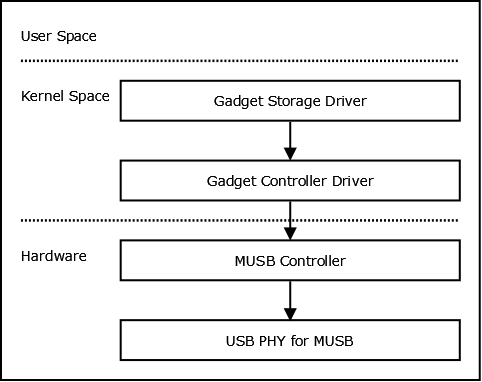
### USB Mass Storage Class Host Driver

#### Driver Features

The driver supports the following features:

1. DMA mode
2. PIO mode

### USB Mass Storage Class Slave Driver



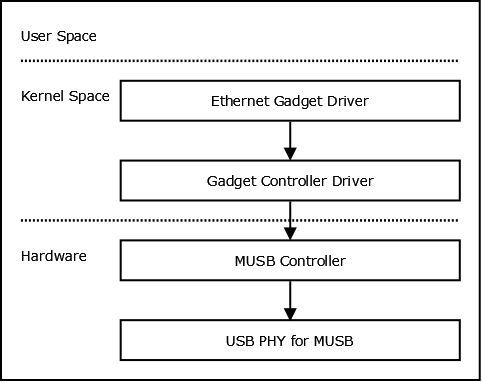
#### Driver Features

The driver supports the following features:

1. DMA mode
2. PIO mode
3. File backed storage driver was tested with SD media as the storage medium

### USB CDC/RNDIS Slave Driver

The CDC RNDIS gadget driver that is used to send standard Ethernet frames using USB. The driver will create an Ethernet device by the name usb0.



#### Driver Features

The driver supports the following features:

1. DMA mode
2. PIO mode
3. 10/100 Mbps speed.

### USB Human Interface Device (HID) Driver

The event sub system creates /dev/input/event\* devices with the help of mdev.

#### Driver Features

The driver supports the following features:

1. DMA mode
2. PIO mode
3. USB Mouse and Keyboards that conform to the USB HID specifications

### USB Isochronous Driver

USB camera, speaker and mic uses isochronouse transfers. USB Video Class (UVC) is used by most of the USB cameras to capture image.

#### Driver Features

The driver supports the following features:

1. DMA mode
2. PIO mode
3. Support for USB Audio and video class(UVC class)

### USB OTG Driver

MUSB controller on DaVinci supports USB On The Go (OTG). OTG protocol enables runtime role switch between USB host and device. This is achived using Session Request Protocol (SRP) and Host Negotiation Protocol (HNP). OTG driver is tested with OPT (OTG Protocol Tester).

## EDMA Driver

The EDMA controller handles all data transfers between the level-two (L2) cache/memory controller and the device peripherals. On AM335x EDMA has has one instance of Channel controller. Each EDMA instance supports up to 32-dma channels and 8 QDMA channels. The EDMA consists of a scalable Parameter RAM (PaRAM) that supports flexible ping-pong, circular buffering, channel-chaining, auto-reloading, and memory protection. The EDMA allows movement of data to/from any addressable memory spaces, including internal memory (L2 SRAM), peripherals, and external memory.

The EDMA driver exposes only the kernel level API's. This driver is used as a utility by other drivers for data transfer.

### Driver Features

The driver supports the following features:

1. Request and Free DMA channel
2. Programs DMA channel
3. Start and Synchronize with DMA transfers
4. Provides DMA transaction completion callback to applications
5. Multiple instances of EDMA driver on a single processor

### Features Not Supported

1. QDMA is not supported.
2. Reservation of resources (channels and PaRAMs) for usage from other masters is not supported.

<http://techvolve.blogspot.sg/2014/05/linux-log-messaging-architeture.html>

Linux log messaging Architeture:

sysctl -a | grep printk  and cat /proc/sys/kernel/printk

  KERN\_EMERG   -  system is unusable. set to 0

  KERN\_ALERT   -  action must be taken immediately. set to 1

  KERN\_CRIT    -  critical conditions. set to 2

  KERN\_ERR     -  error conditions. set to 3

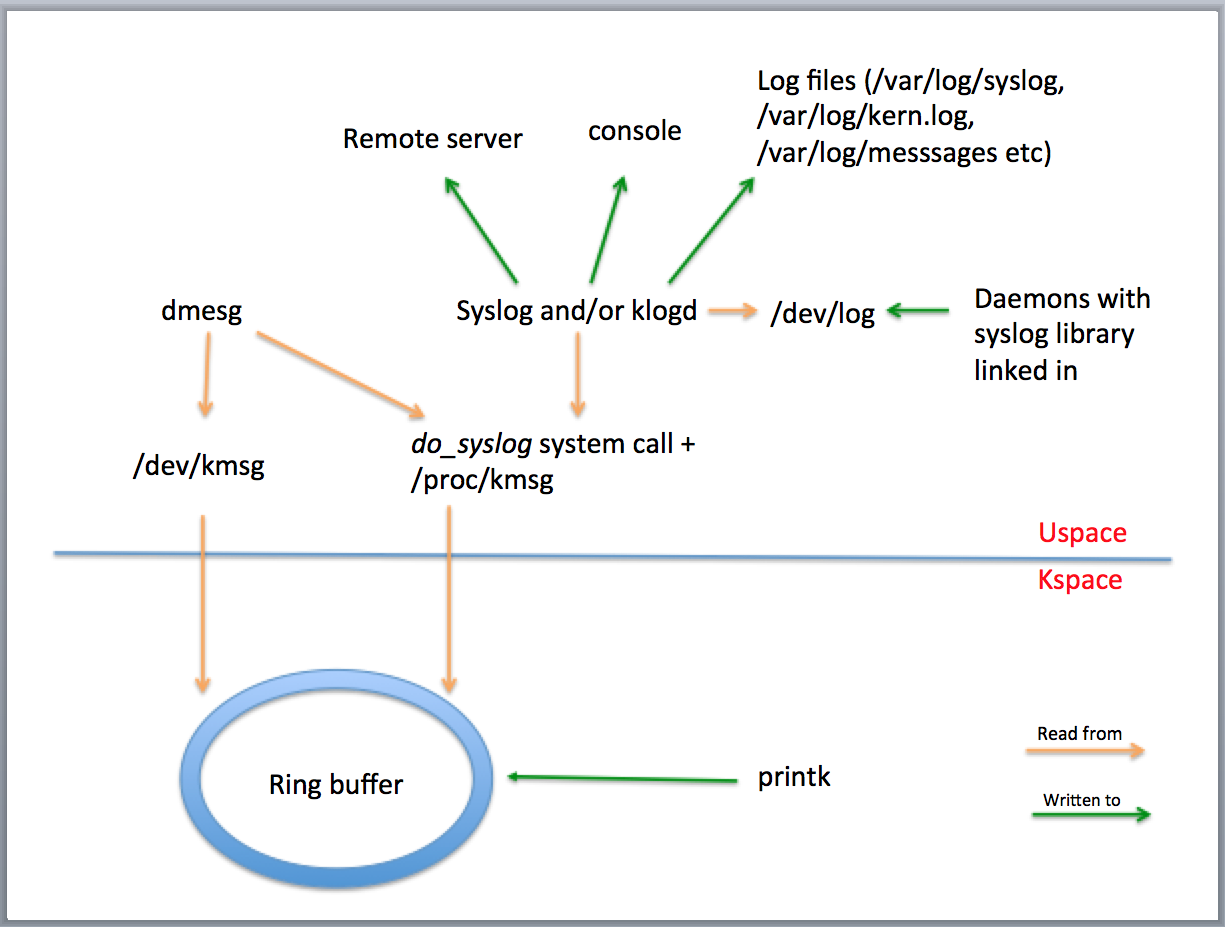
  KERN\_WARNING -  warning conditions. set to 4

  KERN\_NOTICE  -  normal but significant condition. set to 5

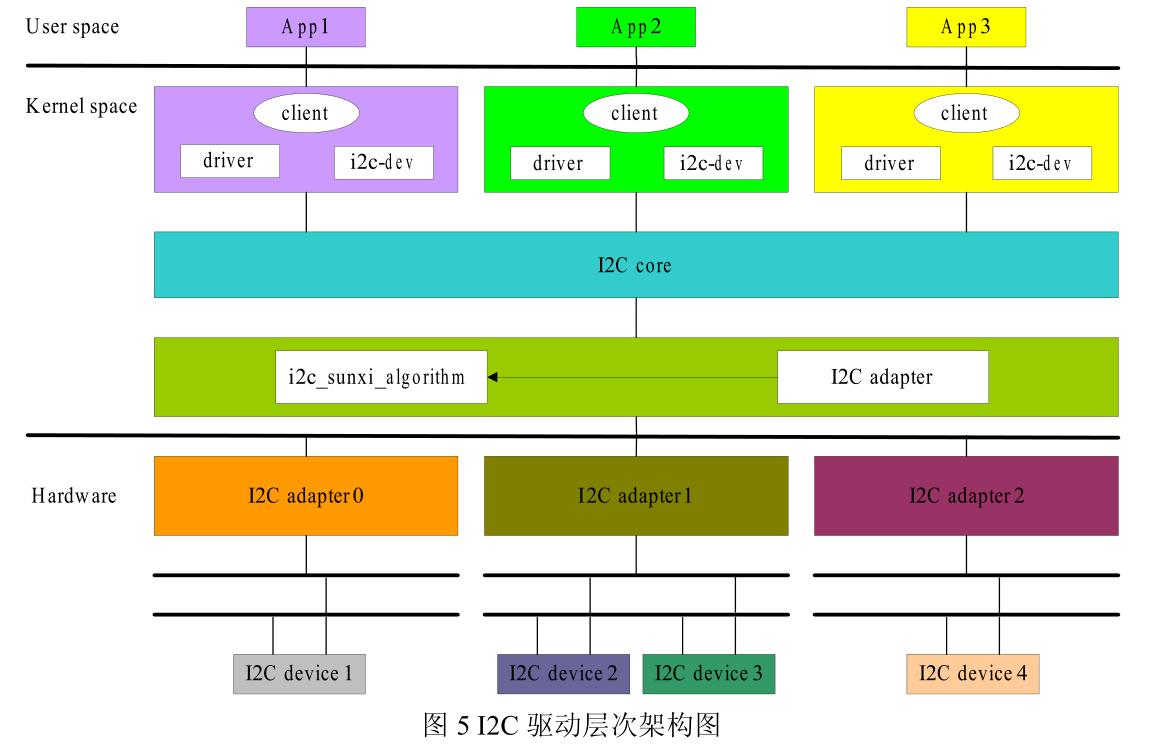
  KERN\_INFO    -  informational. set to 6

  KERN\_DEBUG   -  debug-level messages. set to 7

  KERN\_DEFAULT -  the default kernel loglevel. set to [CONFIG\_DEFAULT\_MESSAGE\_LOGLEVEL](http://lxr.linux.no/linux+v3.13.5/lib/Kconfig.debug#L18)



<http://prog3.com/sbdm/blog/ghostyu/article/details/8094049>



**Linux device Model:** [**http://linuxburps.blogspot.in/2013/12/linux-device-model.html**](http://linuxburps.blogspot.in/2013/12/linux-device-model.html)

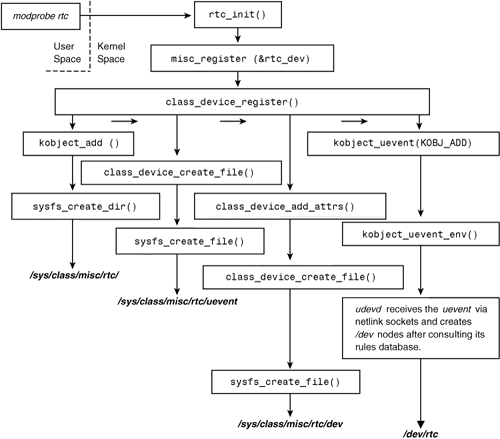
**Device Driver bus class subsystem kojects kset ktypes**

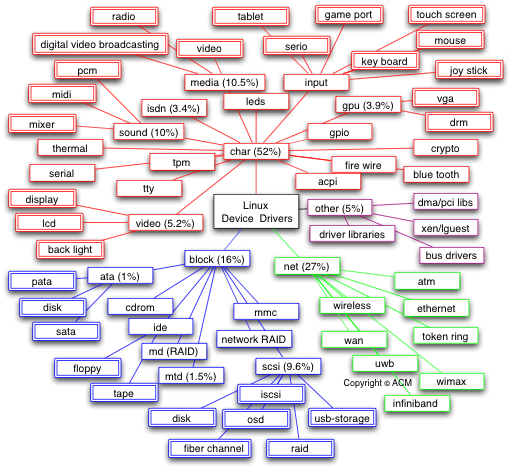
Some of the important structures defined by the device model core are given below.

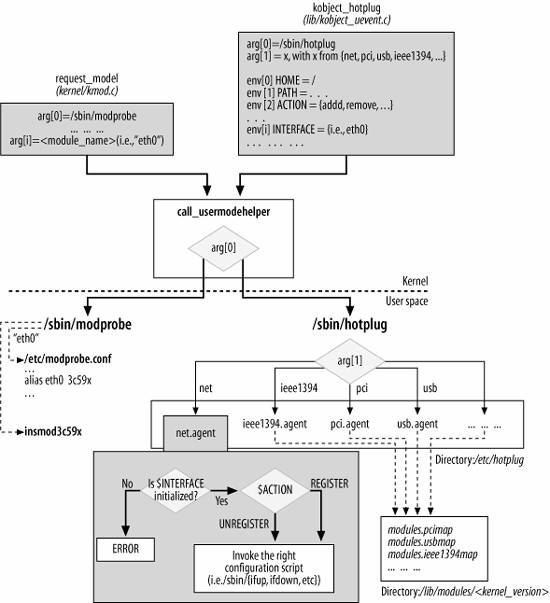
* struct bus\_type
* struct device
* struct device\_driver
* struct class

The struct bus\_type is used to represent buses like PCI, USB, I2C, etc. The struct device is used to represent devices like an Intel AC97 audio controller, an Intel PRO/100 ethernet controller, a PS/2 mouse etc. The struct device\_driver is used to represent kernel drivers that can handle specific devices. The struct class is used to represent a class of devices like sound, input, graphics, etc. no matter how they are connected to the system.

[**https://unixbhaskar.wordpress.com/2010/03/19/insight-into-gnulinux-boot-process/**](https://unixbhaskar.wordpress.com/2010/03/19/insight-into-gnulinux-boot-process/)



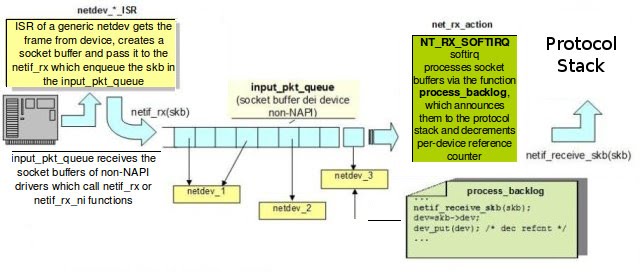




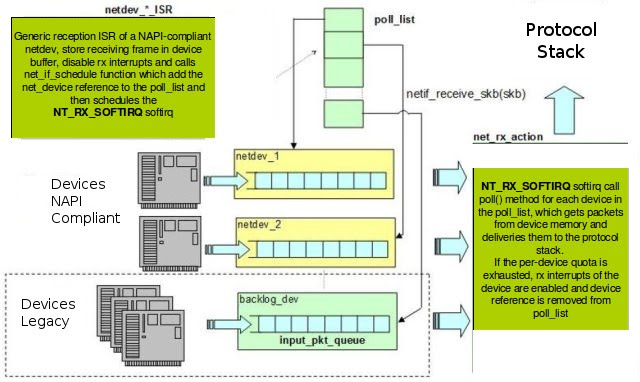
Linux networking subsystem:

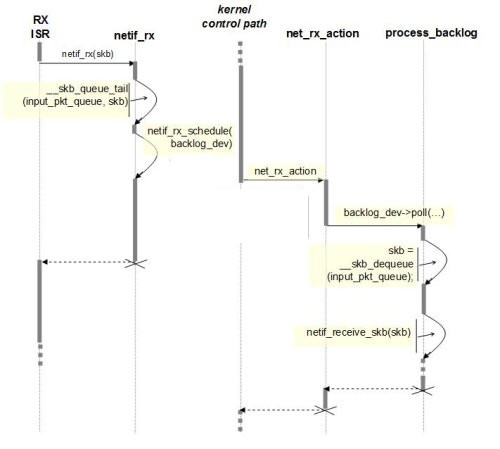
Without Napi frame reception:

<http://www.eantcal.eu/home/articles-and-publications/articles-eng/linux-napi-compliant-network-device-driver>

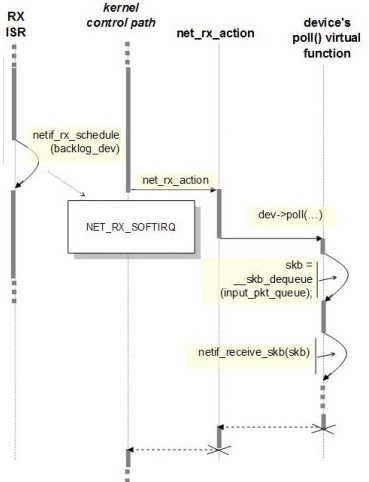


Napi frame reception:





*Sequence diagram related to non-Napi model*



*Sequence diagram related to Napi model*

[*http://the-linux-channel.the-toffee-project.org/index.php?page=3-links-linux-kernel-network-stack-and-architecture*](http://the-linux-channel.the-toffee-project.org/index.php?page=3-links-linux-kernel-network-stack-and-architecture)

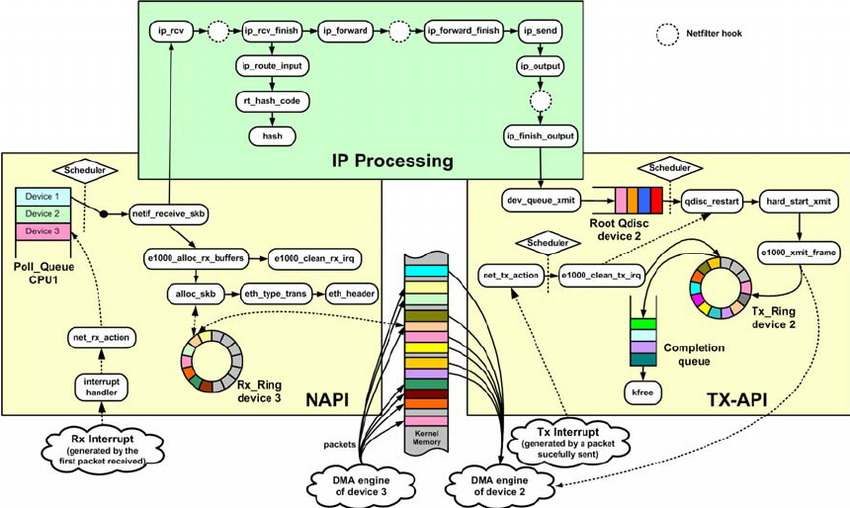
[*https://wiki.linuxfoundation.org/images/1/1c/Network\_data\_flow\_through\_kernel.png*](https://wiki.linuxfoundation.org/images/1/1c/Network_data_flow_through_kernel.png)

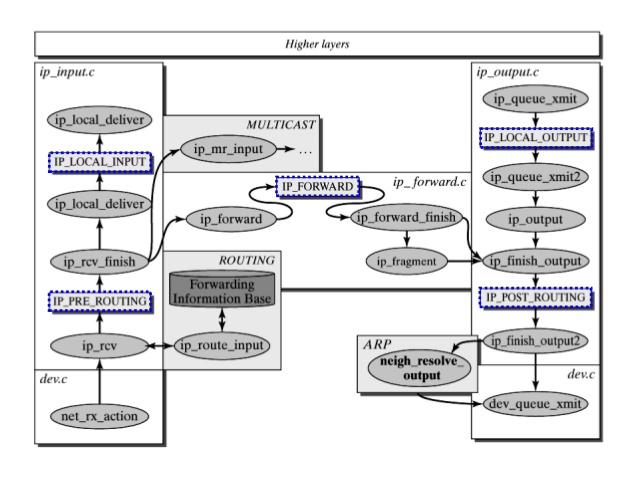
[*http://the-linux-channel.the-toffee-project.org/index.php?page=7-links-sk-buff-operations*](http://the-linux-channel.the-toffee-project.org/index.php?page=7-links-sk-buff-operations)

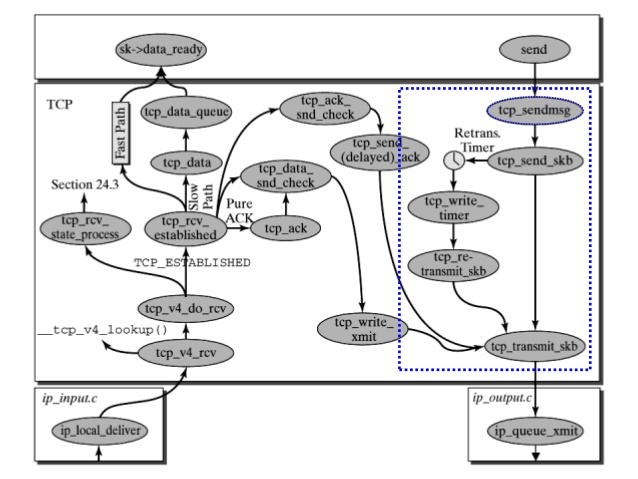
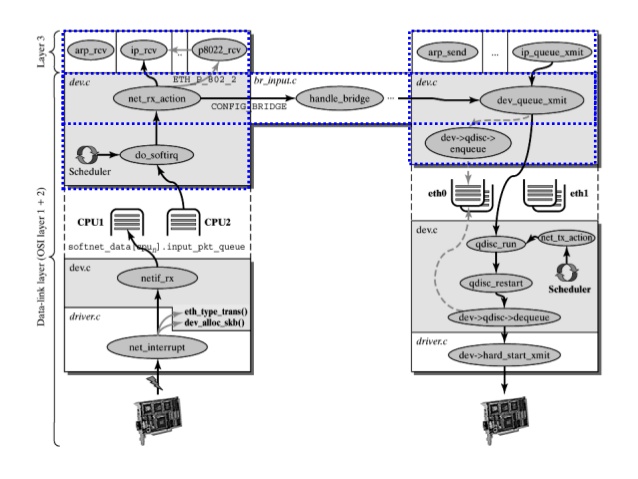
[*http://the-linux-channel.the-toffee-project.org/index.php?page=5-links-ip-tcp-udp-icmp-headers-nmap-org*](http://the-linux-channel.the-toffee-project.org/index.php?page=5-links-ip-tcp-udp-icmp-headers-nmap-org)

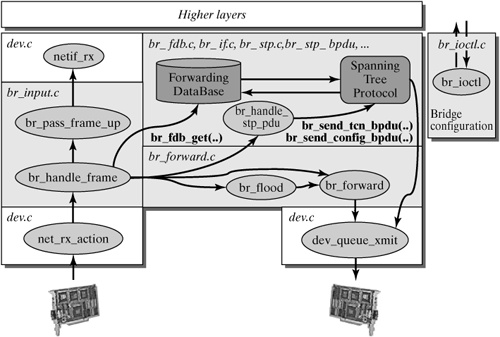
[*http://the-linux-channel.the-toffee-project.org/index.php?page=8-links-network-packet-processing-hardware-offload*](http://the-linux-channel.the-toffee-project.org/index.php?page=8-links-network-packet-processing-hardware-offload)

[*http://www.eantcal.eu/home/articles-and-publications/articles-eng/linux-napi-compliant-network-device-driver*](http://www.eantcal.eu/home/articles-and-publications/articles-eng/linux-napi-compliant-network-device-driver)

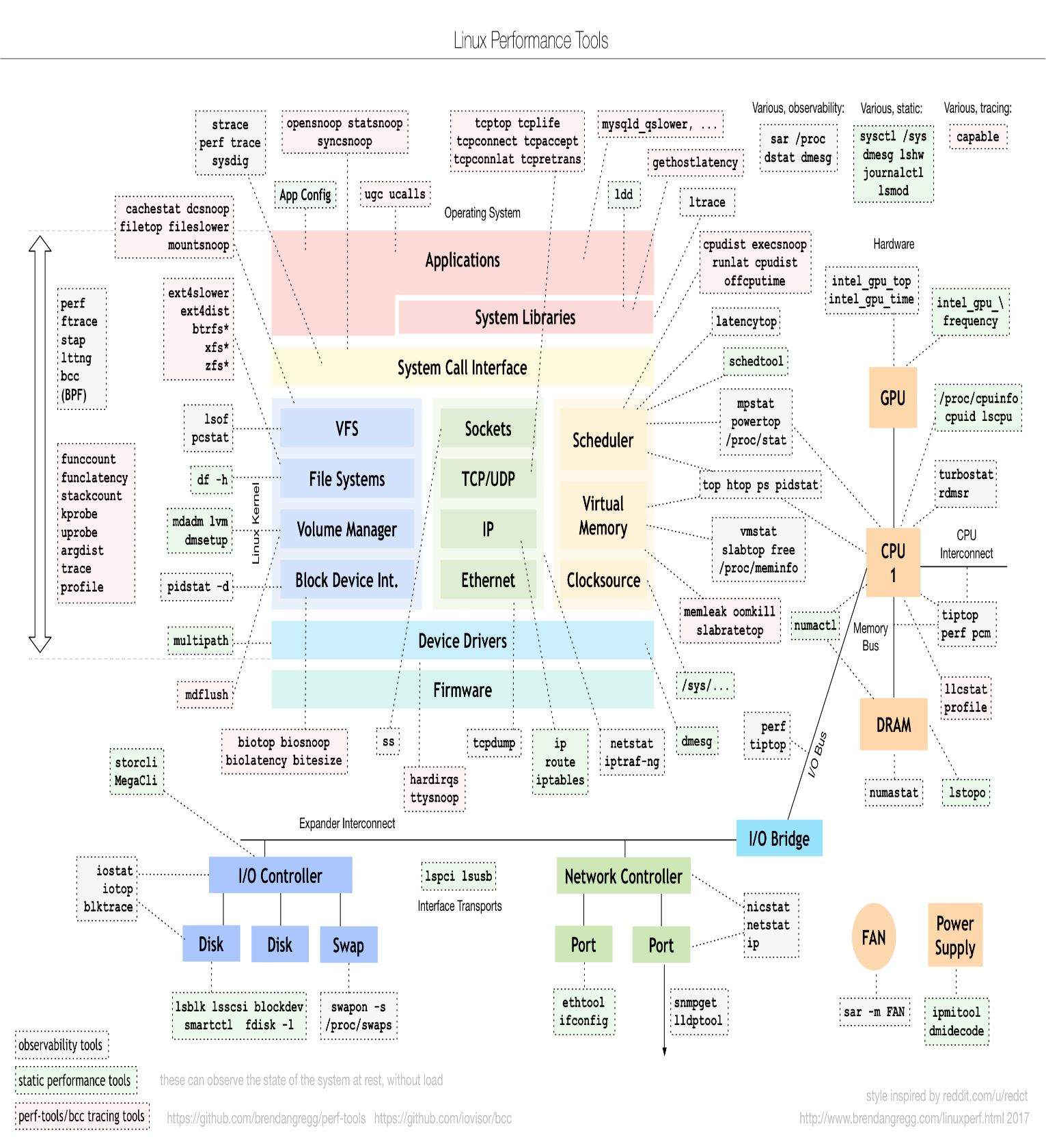








<http://the-linux-channel.the-toffee-project.org/index.php?page=11-links-linux-kernel-vs-performance-tools>



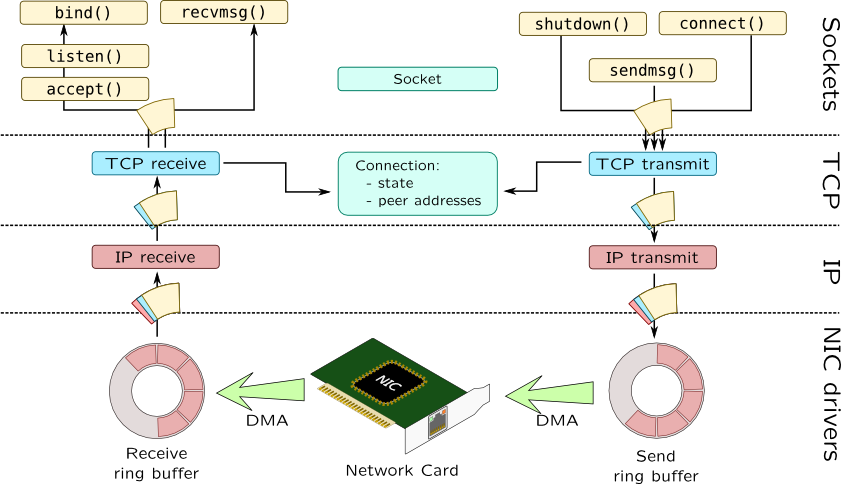
<http://apprize.info/linux/kernel/10.html>

<http://amsekharkernel.blogspot.in/2014/08/what-is-skb-in-linux-kernel-what-are.html>

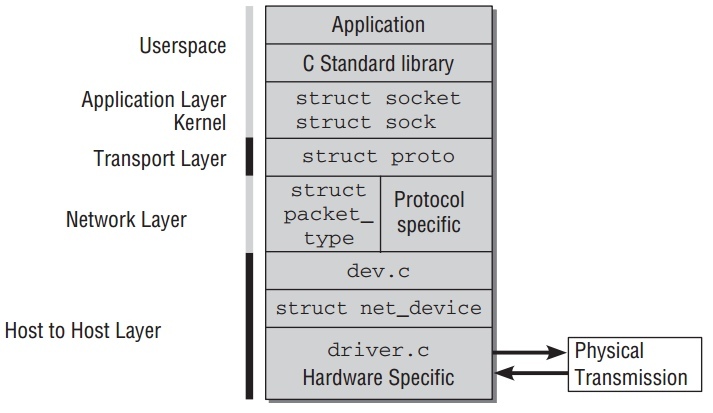
<https://myaut.github.io/dtrace-stap-book/kernel/net.html>

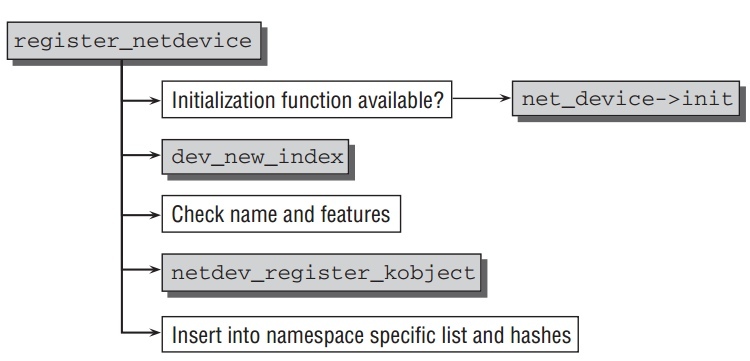
<https://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/>

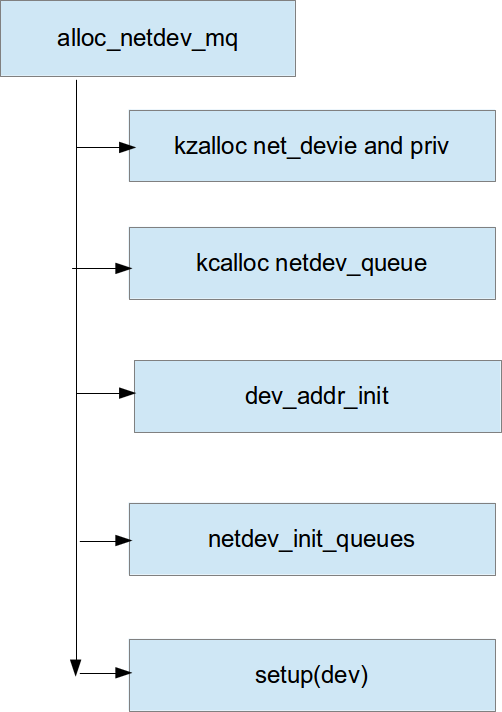
<https://wiki.linuxfoundation.org/networking/napi>



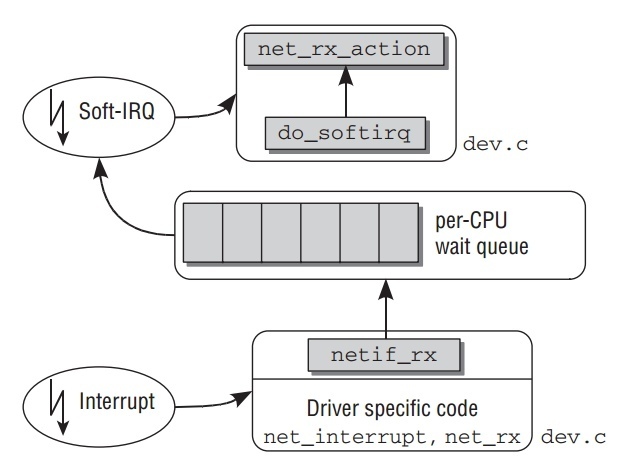


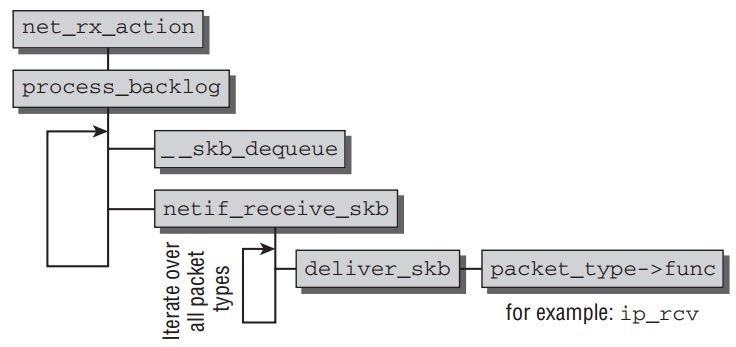






<http://liuluheng.github.io/wiki/public_html/Linux/Networks/nework-access-layer.html>





<https://www.cubrid.org/blog/understanding-tcp-ip-network-stack>

<https://www.ibm.com/developerworks/aix/library/au-aix7networkoptimize1/index.html>

<http://www.cnblogs.com/ggjucheng/archive/2012/11/01/2750217.html>

