Devices

At the lowest level, the I/O system communicates with the hardware (fig.6.1).

Older Unix systems used a simple static interface.

More complicated hardware in newer systems has made it desirable to build help systems to offload the device drivers.

FreeBSD have two new subsystems to support the following services:

- Disk memory partitions and RAID - Handled by the GEOM layer
- General help routines as for example setting up of DMA channels (fig 7.2)
  - → The CAM layer gives general support for several hardware types
  - → The ATA layer supports ATA disk memories.

The PC I/O Architecture

- The I/O architecture for a typical Intel based PC is shown in fig. 7.1
- The PC has several busses connected with bridges
- *Northbridge* is closest to the processor and connects to main memory, graphics bus and to the *southbridge*
- The *Southbridge* connects to the following busses:
  - → PCI - Peripheral Component Interconnect. The PCI-X and PCI Express are newer busses, intended to replace the PCI standard.
  - → ATA - Advanced Technology Attachment bus. An old standard for cheap disk drives. Is about to be replaced with serial ATA. SCSI is primarily used by servers.
  - → Firewire (IEEE 1394) - Primarily used by cameras.
  - → CardBus (PCMCIA) - Used to connect extra equipment to laptop computers.
  - → PIC - Programmable Interrupt Controller. Maps the the device interrupt to IRQ values in older computers. Usually replaced with APIC or LAPIC.
  - → ACPI - Advanced Configuration and Power Interface. A standard for PC power management. Replaces the older APM (Advanced Power Management).
  - → Super I/O - Chip with interface lo legacy PS2 keyboard and mouse ports as well as support for the AC97 sound standard.

Device Naming

- Unix use special files in the **/dev** directory to provide access to hardware devices.
- Internally the system uses *major* and *minor device numbers* to name the devices.
- In older systems, the special files in the /dev directory were created by the system administrator.
- FreeBSD 5.2 use DEVFS (DEvice File System). As devices are discovered either at boot time or while the system is running, their names appear in the **/dev** filesystem.
- When a device disappears, its entry in **/dev** disappears.

Device File System

- The static assignment of device numbers and special files had some problems:
  - → The /dev directory contained a large number of special files that did not represent hardware that existed in the system.
  - → When new hardware was added to the system, the system administrator had to create special files for the device with the correct device numbers.
- A problem with DEVFS is that the device files are given standard access permissions that do not always suit a specific system.
  - → To solve this problem, DEVFS has a configuration file where the system administrator can specify the access permissions for specific device files.
  - → In Linux there is a newer and more flexible system, UDEV, with the same goals as DEVFS.

The GEOM Layer

Located between DEVFS and the device drivers.

The GEOM layer provides a modular transformation framework for disk-I/O requests.

Transformations in GEOM include the following:

- Simple address calculations needed for disk partitioning
- Aggregation of disks to provide RAID
- Encryption of data
- I/O optimizations such as disk sorting

The GEOM Layer - Terminology

GEOM is an architecture rather than an implementation

Object oriented design

Transformation A particular way to modify I/O requests

Class Implements a specific transformation. An example is MBR (Master Boot Record). An instance of a *class* is called a *geom*. Typically there is a *geom* of *class* MBR for each disk memory (It splits the disk in 4 partitions).

Fig. 7.3 shows a sample GEOM configuration:

- At the bottom is a geom that communicates with the CAM layer and produces the disk da0.
- Above da0 is stacked a MBR geom that interprets the MBR label found in the first sector of the disk to provide two partitions da0s1 and da0s2.
- These partitions have DEVFS consumers that export them as /dev/da0s1 and /dev/da0s2

GEOM - Implementation

- The FreeBSD GEOM implementation uses two threads to manage a stack of *geoms*.
- The *g_down* thread process requests moving from the top of the stack down to the driver.
- The *g_up* thread process results returning from the driver.
- This single-threaded implementation imposes the following restrictions on a geom:
  - ➝ It may not block
  - ➝ It may not do time consuming calculations
- The commands that may pass through the GEOM stack are *read*, *write* and *delete*.

ATA

- Because CAM was designed for SCSI disks and have a structure that do not suit ATA disks, the ATA disks are handled by its own subsystem.
- An ATA controller have two channels each of which support a master unit and a slave unit.
- The common parts of the ATA layer handles the parts of the disk memory interface that are not specific to a particular hardware unit (fig. 7.8).
- The filesystem code passes a read/write request down to the GEOM layer by calling the *adstrategy* routine.

adstrategy calls *bioq_disksort* which puts the request on the disc queue of the specified ATA disk. Then *ata_start* is called.

ata_start manages the channel queues. If needed, the data transports will alternate between master and slave. If the channel is free, a hardware specific driver routine (*ad_start*) is called to move the operation from the bio queue to the correct channel queue. Then *ata_begin_transaction* is called to start the data transport.

ata_begin_transaction Sets up a DMA channel if needed and starts the data transport by writing to the units control register.

ATA - cont.

The channel queues are managed by the data structure *ata_request.*

Then a data transport is completed the interrupt routine, *ata_interrupt,* is called.

ata_interrupt updates the request with the command completion status, releases the channel and returns the request to the ATA layer by calling *ata_finish*.

ata_finish puts the completed I/O request on the ATA work queue to be run by the *g_up* thread and returns from the interrupt.

ata_completed is called by the *g_up* thread. It checks error codes returned by the hardware and calls the driver specific callback routine (*ad_done* for disk memories). If needed it then calls *ata_start* to start the next data transport.

ad_done updates the bio structure and wakes the filesystem by calling *biodone*().

Autoconfiguration

- The early Unix systems used static device configuration by manually editing the kernel sources to call the correct driver.
- Autoconfiguration is a procedure carried out by the system to recognize and enable the hardware devices present in the system.
- Originally FreeBSD used the autoconfiguration code from 4.2BSD.
- The current FreeBSD uses a new autoconfiguration system called *newbus*.
- A design goal for newbus was to provide a stable ABI (Application Binary Interface) for the drivers.
- Unfortunately the other interfaces to the drivers do not provide a stable ABI.

Autoconfiguration - cont.

- The autoconfiguration works by systematically probing the possible I/O busses on the machine.
- For each bus it is decided which devices are present on the bus.
- Because the busses may contain bridges to other busses, the procedure must be recursively repeated.
- The autoconfiguration must take care of some problems.
  - ➝ Some devices must exist for the system to work, but other devices may be connected while the system is running.
  - ➝ Devices may be present in different numbers and on different addresses.
  - ➝ Devices on newer busses are usually self-identifying which means that the device have a status register that can be read to determine its identity but some older non-self-identifying busses are still in use.
- To address these problems, FreeBSD support both static configuration at kernel compile time and dynamic configuration at boot time or later.
- Both FreeBSD and Linux also support loadable device drivers, so called *kernel modules*.

Autoconfiguration - cont.

- Address information for non-self-identifying busses are given in the file */boot/devices.hint*.
- This file and other configuration information is used by /usr/sbin/config to build the static configuration.
- Autoconfiguration is done early during boot by calling the architecture-dependent routine *config()*.
- The *config()* routine configures root0, which is the top node in the configuration tree. Directly below root0 in I386 systems is nexus that represents the northbridge.
- Then the architecture independent routine *root_bus_configure()* is called which calls *device_probe_and_attach(dev)* for every device that is a child of *root0*.
- This routine makes recursive calls until all devices are configured.
- The device drivers that support autoconfiguration have to register a set of subroutines that are later called by *device_probe_and_attach().*

Autoconfiguration - cont.

Different routines are used depending on if the bus is self-identifying or not:

1. *device_identify* is used for older busses like ISA-bus, that only can identify devices using hints. Tests each possible location at which a device might be present.
2. *device_probe* is used for newer busses. Requires that the bus specifies registers or addresses that can be read to identify connected devices.

Sometimes a device can be handled by several different drivers. In this case the best driver should be selected.

Autoconfiguration - Device Drivers

The device driver need to register the following subroutines for the autoconfiguration:

device_probe tests if the device is present. Returns a return value that indicates how good the driver matches the found device.

device_attach is called for the driver who's *device_probe* returned the best match for the device. Initializes the hardware and allocates an entry in devfs. Also allocates needed hardware resources.

device_identify is used instead of *device_probe* for busses who's devices only can be found using hints.

device_detach is called if a device (for example a USB-memory) is removed from the system.

device_shutdown is called when the system is shut down.

device_suspend is called by power-management routines before the machine enters a power-save state.

device_resume is called at return a suspend state.

Autoconfiguration - Data Structures

The autoconfiguration uses the data structures *devclass*, *device* and *driver* (fig. 7.10).

devclass represents a bus. It has a pointer (*driver*) to a list of device drivers. This list is initialized by advanced magic early in the boot process before autoconfiguration has started. The content in the list is determined at kernel compile time.

driver represents a device driver. The driver's *probe* and *attach* routines are called via this data structure during autoconfiguration.

device represents a hardware device. After autoconfiguration it points to the driver that is selected for the device.

The result of the autoconfiguration process is a tree of device structures (fig. 7.12 and fig. 7.13).

Resource Management

- The device drivers must also take care of allocating hardware resources as interrupt-request lines (irq), I/O ports and memory addresses for local memory.
- Newbus specifies a framework for management of such resources.
- The drivers have to register the subroutines specified in table 7.2.
- Low level drivers do not have the global knowledge of resource utilization needed to allocate for example irq lines.
- To solve this problem they may register a generic bypass routine (*bus_alloc_resource*) that calls the corresponding routine registered by their parent.
- This is repeated until a level is reached that has enough information to do the allocation.
- In most cases an allocation is possible only at the highest level (nexus).
- To keep track of free and busy resources general resource handlers are used (rman_xxx).