Device Drivers

Hardware Software CoDesign

January 2012

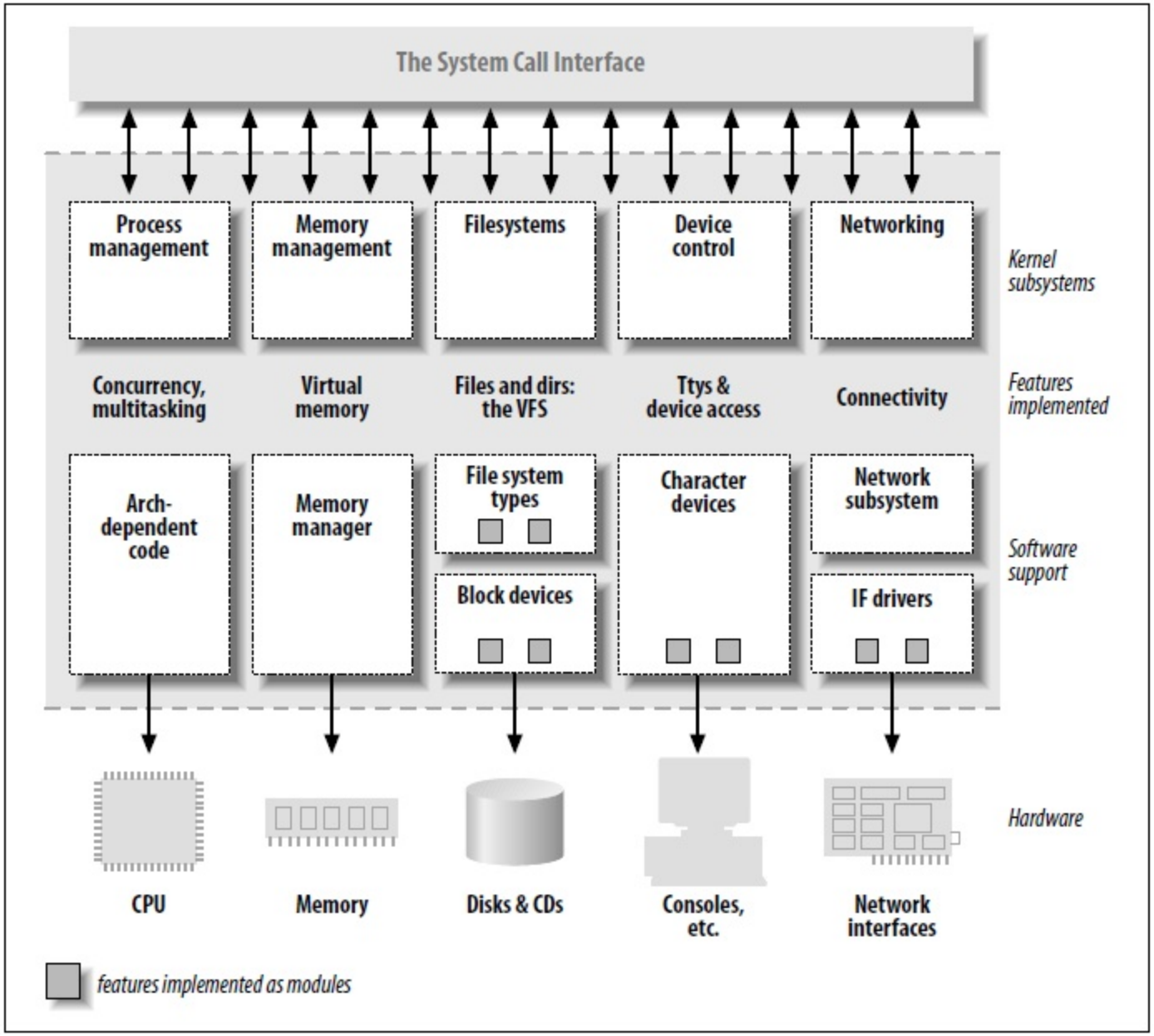
Agenda

Device Drivers

1. Introduction to Device Drivers (quick repeat for completness)
2. More on Device Drivers
3. Taking stock before EndSem Examination
4. Collection of Answering Machine Assignment

Device Drivers

Kernel view of Things



A split view of the kernel

Device Drivers

Kernel view of Things

1. The Kernel is the big chunk of executable code in charge of handling all requests → computing power, memory, network connectivity, any other resource allocation.

2. The distinction between the different kernel tasks isn't always clearly marked, the kernel's role can be split into :-

3. *Process Management* :: Creating and destroying processes and handling their connection to the outside world (IO). Communication among different processes (through signals, pipes, IPC). The scheduler, which controls how processes share the CPU, is part of process management.

4. *Memory Mangement* :: The computer's memory is a major resource, and the policy to deal with it is a critical one for overall system performance. The kernel builds up a virtual address space for all processes. The different parts of the kernel interact with the memory-management subsystem through simple `malloc / free` and other more complex memory functions.

5. *FileSystems* :: Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resultant file abstraction is heavily used throughout the whole system.

6. *Device Control* :: Almost every system operation eventually maps to a physical device. With the exception of the processor, memory and a very few other entities; any and

all device control operations are performed by code that is specific to the device being addressed. That code is called *Device Driver*. The kernel must have embedded in it a device driver for every peripheral present on a system (hard drive, printer, kbd, tape..)

7. *Networking* :: Networking must be managed by the operating system, because most network operations are not specific to a process. Incoming packets are asynchronous events. These packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control execution of programs according to their network activity.

Device Driver

What is a Device Driver

1. A set of software procedures or apiś which enable higher level of programs (applications) to interact with hardware device

2. A Device Driver ”Controls”, ”Manages” and ”Configures” devices connected to the system

3. The driver ha few functions and provides the logic to initialize and communicate with the hardware

Device Driver

Device Driver Types

1. Service Device Driver :: If the device driver will be communicating directly with the hardware device, this interface will be the memory ranges, registers and/or ports through which I/O to the device takes place.

2. System Device Driver :: If the device driver will be communicating with its device via an intermediate device, this interface will be whatever APIs the driver for the intermediate device exposes. Eg. a GPS device driver will use the serial port driver API – the stream interface functions – to communicate with a GPS receiver.

Device Driver

Device Driver Need & Challenges

1. By providing a clear abstraction between the driver and the application, one can essentially free the application of the specifics of a certain peripheral and port it more easily to new hardware

2. Device Drivers have a tight connection to the target device and the development environment, these are usually not portable. Eg., device drivers across microcontroller families.

3. Device Drivers consist of a lot of "bit bashing" and register programming. One needs to get all the details right $\rightarrow$ the bits, sequences of initialization and exit, timing (say flash, ddr), or else the hardware would malfunction or not provide the desired output.

4. Device Drivers also enable Debug of system malfunction (due to modularization). E.g. 85% of failures in WinXP were from buggy device drivers.

5. GROUP DISCUSSION :: So, How would you go about developing a Device Driver?

Device Driver

Device Driver Development → HardWare Side

1. Reading the hardware manual and learning the chip internals

2. Learn about the hardware / platform. Identify the interface between the driver and the device

   (a) How the device is reset

   (b) How the device is "address mapped"

   (c) Return codes and software protocols recognized by the device

   (d) What are the types of DMA transfers possible with the device

   (e) How the device reports hardwar failures

   (f) How the device sends / responds to interrupts

3. Test the hardware to make sure it is functioning. This becomes important for a newly developed device.

Device Driver

Device Driver Development → SoftWare Side

1. What kind of device driver library will enable the application ?
   (a) Identify the APIs that the driver must expose
   (b) Initialization and De-initialization routines (eg. setting up of baud rate or timer periods)
   (c) Run time control (Read, Write, Interrupts, Signal Responses..)
   (d) Notification and any additional interfaces that the Device Driver needs to use or other OS APIs or services
   (e) Version Control for Software
   (f) Bug Tracking and closure utility
   (g) Suite of testcases to test scenarios where the Device Driver is used
   (h) Integration into the System and the OS Kernel

Device Driver

Device Driver Development → SoftWare Side

1. Monolithic device driver library
   (a) Based on a single piece of code that exposes hardware devices functionality directly to the OS
   (b) Accesses device(s) directly
2. Layered device driver library
   (a) A model device driver based on an "upper" layer which is logical and "lower" layer which is physical
   (b) RTOS abstraction layer
   (c) Compiler abstraction layer ?
   (d) Hardware abstraction layer ?

Device Driver

Device Driver Development → SoftWare Side PROs and CONs of Monolithic Vs Layered

1. Monolithic :: Performance is better as it avoids calling many functions.

2. Layered ::

   (a) Because of modularity, the structure of software is easy to understand.

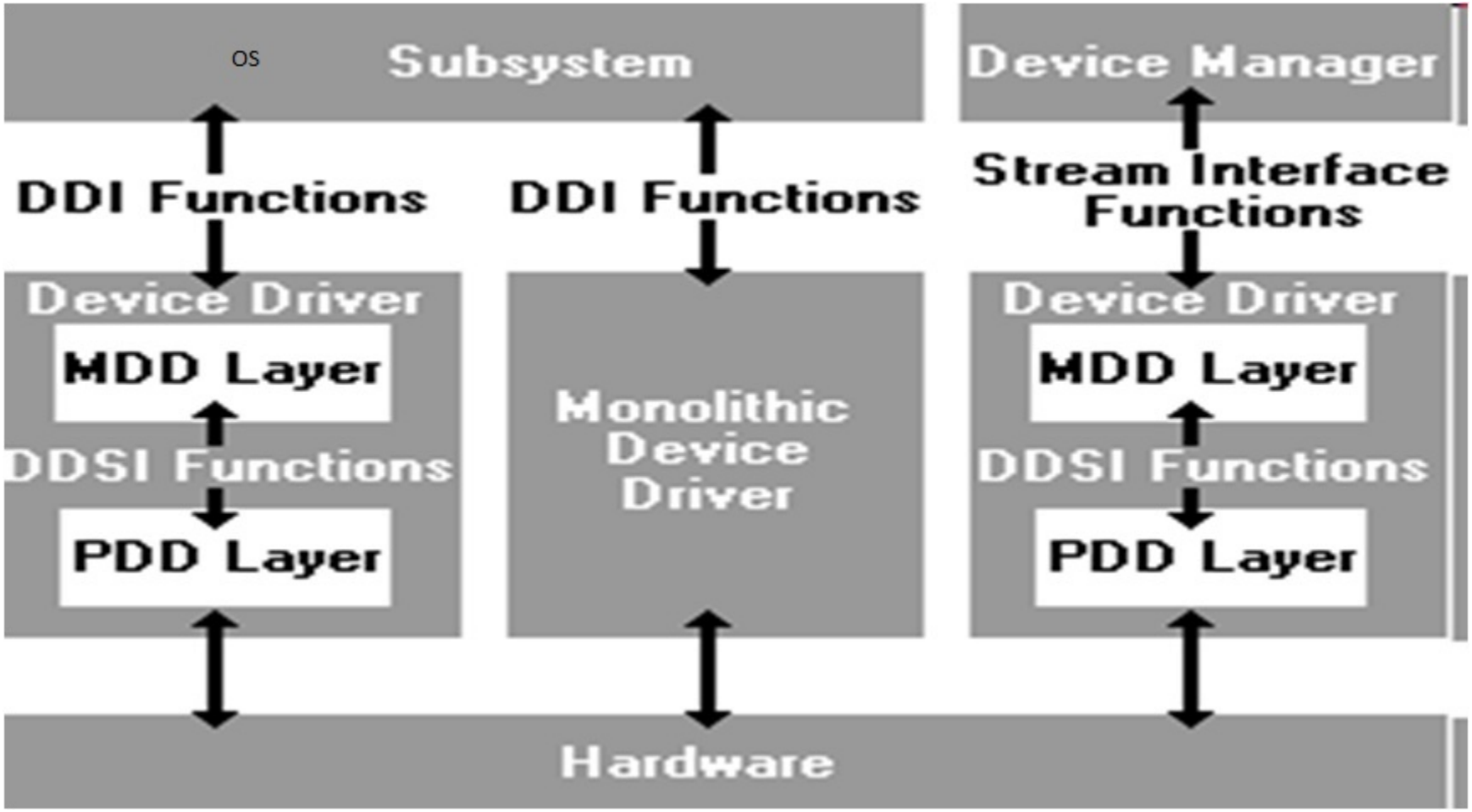
   (b) It is easy to add or modify features of the overall application as it evolves and gets deployed.

   (c) Because there is one module that ever interacts directly with the peripherals registers, the state of the hardware device can be more accurately tracked.

   (d) Software changes that result from hardware changes are localized to the device driver, making software more portable.

   (e) Enhance the reusability, but bit of extra effort up front, at design time, in order to realize the savings.

Device Driver



1. PDD :: Platform Dependent Driver (physical layer)
2. MDD :: Model Device Driver (logical layer)
3. DDI :: Device Driver Interface
4. DDSI:: Device Driver Service Provider Interface

Device Driver

Concepts → Mutual Exclusion of Device Access

1. One of the most basic requirements of a Device Driver is the need to ensure that only one application task at a time can request an input or output operation on a specific device

2. Example :: Same hardware I2C is shared between multiple devices like FM tuner, LCD, Temperature sensor on Hardware board

3. Semaphores can be used in the open and close function to ensure mutual exclusion

4. In case the target allows multiple simultaneous accesses, like in some complex file IO, it can be through system of arbitration logic

Device Driver

Concepts → Synchronous Vs Asynchronous driver operation

1. Do you want the application task that called the device driver to wait for the result of the IO operation that it asked for? → Blocking (Synchronous)

2. OR do you want the application task to continue to run while the device driver is doing its IO operation? → Not Blocking (Asynchronous)

3. Question to Ask (Software Side) :: If my task requests an IO operation through a driver, then what work can it usefully do before that IO operation is done?

4. Question to Ask (Hardware Side) :: Does the completion of the IO operation through driver provide a logical status only or it also provides an interrupt?

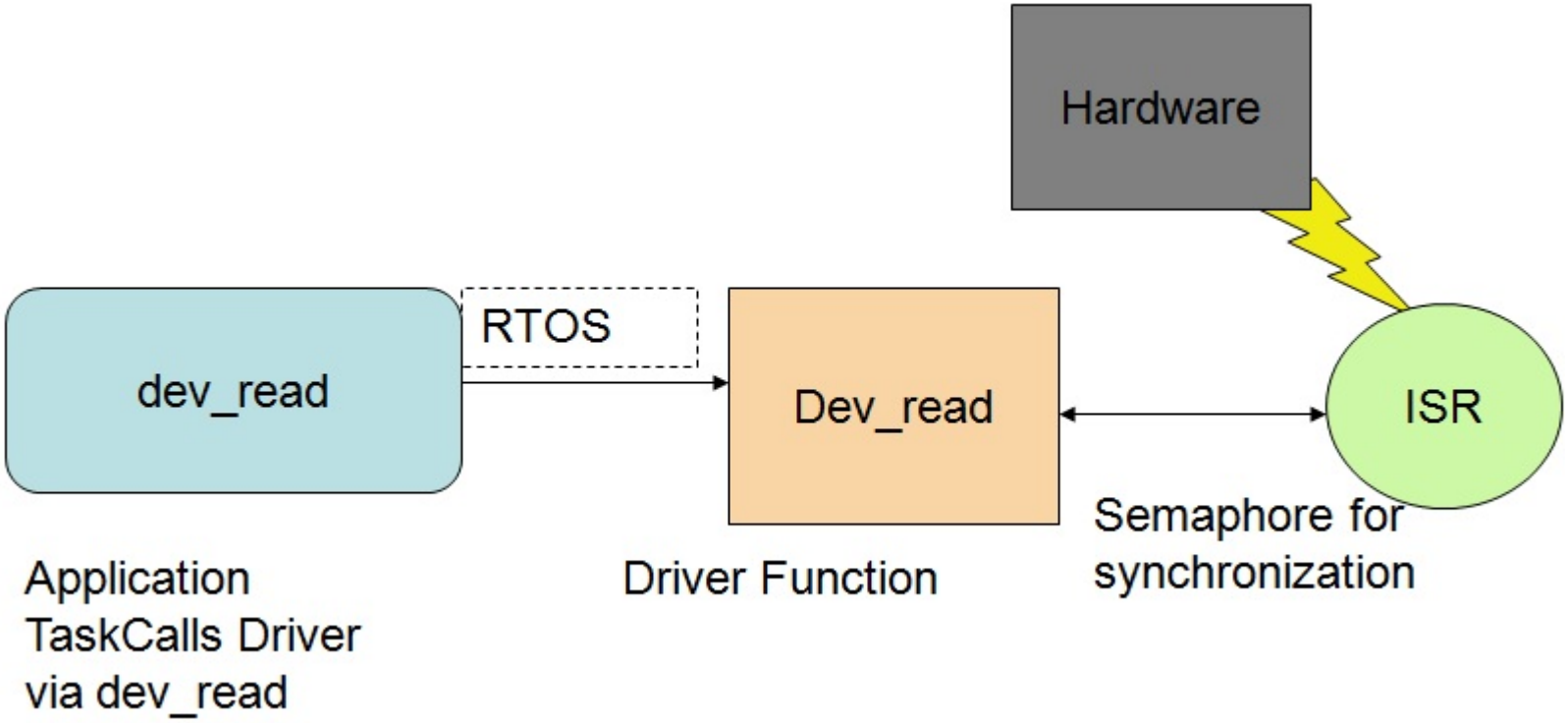
5. Based on such questions, one can decide Synchronous Vs Asynchronous drivers

6. GROUP DISCUSSION :: if a task calls a driver via a "read" call, the driver function implementing the call would act as a subroutine of the caller task. And if this driver attempts to get a semaphore token that is not present, the driver function would be blocked from continuing exectuion; and together with it, the requesting task would be in waiting state → Synchronous or Asynchronous

Device Driver

Concepts → Synchronous Vs Asynchronous driver operation

1. GROUP DISCUSSION :: if a task calls a driver via a "read" call, the driver function implementing the call would act as a subroutine of the caller task. And if this driver attempts to get a semaphore token that is not present, the driver function would be blocked from continuing exectuion; and together with it, the requesting task would be in waiting state → Synchronous* or Asynchronous



2. the read call is dev_read() from the OS or RTOS side.

3. this in turn calls the device driver routine named dev_read()

4. the drivers read function will request the device hardware to perform a read operation, and then it will attempt to get a token from the semaphore to its right.

5. Since the semaphore initially has no tokens, the driver "read" function, and hence all the related tasks will be "BLOCKED".

6. The hardware interrupt triggers the execution of the ISR.

7. When device hardware completes the previous read operation, it would trigger this ISR, that will put a token into the semaphore.

8. this is the semaphore token for which the Task is waiting, and will become unblocked and proceed to fetch the newly-read data from hardware

```
dev_read() {
  Start IO Device Read Operations;
  Get Synchronizer Semaphore Token /*Wait for Semaphore token */;
  Get Device Status and Data;
  Give Device Information to Requesting Task;
}

dev_isr {
  Calm down the hardware device;
  Put a token into synchronizer semaphore;
}
```

1. GROUP DISCUSSION :: Write a similar code for Asynchronous Drivers (NON Blocking)

Device Driver

Concepts → Synchronous Vs Asynchronous driver operation

1. In an "Asynchronous" driver, the application task that called the device driver may continue executing, without waiting for the result of the IO operation it requested

2. it is more complex than synchronous drivers

3. GROUP DISCUSSION :: Write a similar code for Asynchronous Drivers (NON Blocking)

```
dev_read_async() {
  Get Message from the Queue /* Wait only if Queue is Empty */;
  Start new IO device read operation;
  Give old Device Information to the Requesting Task;
}

dev_read_async_isr() {
  Calm down the hardware device;
  Get Data/Status information from Hardware;
  Package this information into a Message;
  Put Message into the Queue;
}
```

1. Complexity is :: How much buffer is required, need to create an memory management structure around it → stack, fifo

Device Driver

Concepts → Miscellaneous

1. Compiler Selected :: for code optimization, selection of variable types (global, volatile, loops..)
2. Memory Mangement :: usage of stacks, fifos
3. Interrupt Vs Polling
4. Sampling Frequency :: How often to read from ADC?
5. DMA Vs Byte/Word/Double Mode
6. How much to buffer
7. Data structure

Device Driver

Concepts → Performance Analysis

1. Interrupt Latency :: Response time, very key to RTOS

2. Performance and Throughput :: Memory read /write speed from a SD Card

3. Memory Foot Print :: For a Network Interface Controller (NIC) what is the memory requirement on per packet basis, or the max number of packets if burst modes are supported

4. CPU Load :: "GROUP DISCUSSION :: give examples"

5. Hardware Resource Used :: Some device drivers may use multiple hardware resources. Say a higher level CodecDriver would use DMA, SPI, Memory interfaces including DSP Core bandwidth.

Device Driver

Example → LoopBack (codec) driver

1. Writing a codec driver for a DSP involves programming three different peripherals :: the codec itself, the SPI, and the DMA controller

```
void main() {
  void* buf0, buf1, buf2, buf3;
        /*Allocate buffers for the SIO buffer exchange*/
buf0 = (void*) MEM_calloc(0, BUFSIZE, BUFALIGN);
buf1 = (void*) MEM_calloc(0, BUFSIZE, BUFALIGN);
buf2 = (void*) MEM_calloc(0, BUFSIZE, BUFALIGN);
buf3 = (void*) MEM_calloc(0, BUFSIZE, BUFALIGN);

      /*Create the task and open the I/O streams */
      TSK_create(echo);
      inStream = SIO_create("/codec", SIO_INPUT, BUFSIZE);
      outStream= SIO_create("/codec", SIO_OUTPUT, BUFSIZE);

      /*Start the scheduler when main() exits*/
}
```

Device Driver

Example → LoopBack (codec) driver

1. The application uses the SIO_create() call to create the channels.

2. The SIO_create() function arguments indicate the design decisions when implementing the driver.

3. The SIO stream can be opened either for reading or writing, but not both. Example :: LCD is only write and KBD is only read.

4. If BiDi communication is required, the application opens two channels (as above example).

5. Most Codecs operate of fixed-sized frames of data, so BUFSIZE can be a constant.

Device Driver

Example → LoopBack (codec) driver

```
void echo(){
  int sizeRead; // Number of buffer units read
  unsigned short *inbuf, *outbuf;

  /* Issue the first & second empty buffers to input stream.*/
  SIO_issue(inStream, buf0, SIO_bufsize(inStream), NULL);
  SIO_issue(inStream, buf1, SIO_bufsize(inStream), NULL);

  /* Issue the first & second empty buffers to output stream. */
  SIO_issue(outStream, buf2, SIO_bufsize(outStream), NULL);
  SIO_issue(outStream, buf3, SIO_bufsize(outStream), NULL);

  for (;;) {
  /* Reclaim full buffer from input stream
     and empty from output stream. */
     sizeRead = SIO_reclaim(inStream, (void**)&inbuf, NULL);
     SIO_reclaim(outStream, (void**)&outbuf, NULL);

  /* Copy data from input buffer to output buffer.*/
  for (int i = 0; i < sizeRead; i++) {
    outbuf[i] = inbuf[i];
  }

  /* Issue full buffer to output stream
     and empty to input stream. */
  SIO_issue(outStream, outbuf, nmadus, NULL);
  SIO_issue(inStream, inbuf, SIO_bufsize(inStream), NULL);
  }
}
```

Device Driver

Example → LoopBack (codec) driver

1. mdCreateChan() :: Create a channel. Add data structure which can include channel state information, information about the current IO packet being processed, a linked list of packets queued for processing, and the call back function that is to be used to notify the driver that a packet processing is complete.

2. mdBindDev() :: Initializes the devices

3. mdUnBindDev() :: Freez any resources allocated by the driver

4. mdSubmitChan() :: Processes IO request (read / write). The function will receive an IO packet from the driver and either put the packet in queue if the function is already working on a previous job, or start working on the packet right away. All of the state driver information required to accomplish this is contained in the channel-object strucutre. Interrupts is disabled in this function (by design) to maintain the coherency of the channel state. Assumption is that the period is short for proper driver functioning.

5. mdControlChan() :: Enables the application to perform device-specific control, such as device reset, volume change.

Device Driver

Example → LoopBack (codec) driver

```
//data structure used & created in mdCreateChan()
typedef struct {
     bool inuse; // TRUE => channel has been opened
     int imode; // IOM_INPUT or IOM_OUTPUT
     IOM_Packet *dataPacket; // current active I/O pkt
     QUE_Obj pendList; // list of packets forI/O
     unsigned int *bufptr; // pointer *within* current buf
     unsigned int bufcnt; // remaining samples
     IOM_TiomCallback cbFxn; // used to notify client
     void* cbArg; // arg passed with callback function
   } ChanObj, *ChanHandle;

static int mdSubmitChan(void* chanp, IOM_Packet *packet) {
   ChanHandle chan = (ChanHandle) chanp;
   unsigned int imask;

   imask = HWI_disable(); // disable interrupts
   if (chan->dataPacket == NULL) {
     /* Start I/O job. */
     chan->bufptr = (unsigned int *)packet->addr;
     chan->bufcnt = packet->size;

     // dataPacket must be set last, to synchronize with ISR.
     chan->dataPacket = packet;
  } else {
```

```
    /* There is an I/O job already pending; queue packet. */
    QUE_put(&chan->pendList, packet);
  }
  HWI_restore(imask); // restore interrupts
  return (IOM_PENDING);
}
```

Device Driver

Example → LoopBack (codec) driver

1. GROUP DISCUSSION :: Write a state diagram of the use of the driver by the application.

Device Driver

General look of a Driver The structure of a driver is similar for a peripheral device (say TouchScreen)

1. There is an init routine for initializing the hardware, setting memory from the kernel and hooking the driver-routines into the kernel.

2. There is a data structure that is initialized with routines that are provided with the device. This strucutre is key and is used by the applications.

3. Mostly there are open and release routines.

4. Mostly there are routines for reading and writing data to or from the driver, a ioctl routine that can perform special commands to driver like config requests or options.

5. Mostly there are routines for interrupt handling if hardware supports.

6. The Driver itself, can be either "Compiled as part of Kernel" or "Dynamically Loaded" at runtime. The only differences between a loadable "Module" and a kernel linked driver are a special `init()` routine that is called when the module is loaded into the kernel and a cleanup routine that is called when the module is removed.

Device Driver

ReCall → Touch Screen System Device Driver

1. Configure the controller hardware.
2. Determine if the screen is touched.
3. Acquire Stable, debounced position measurements.
4. Calibrate the touch screen.
5. Send changes in touch status and position to the higher level graphics software.

Device Driver

ReCall → Touch Screen System Device Driver

Configure the controller hardware.

1. Create a function named `TouchConfigureHardware()`
2. Decide – Should the driver be interrupt driven or polling driven.
3. In case of interrupts, the driver would actually use two :-
   (a) An interrupt to wake up when the screen is initially touched, known as the `PEN_DOWN` interrupt.
   (b) A second interrupt to signal when the ADC is available with the set of data conversions (X, Y).

Determine if the screen is touched.

1. Create a function named `WaitForTouchState ()`
2. When the controller is in the detection mode and a touch is detected, an internal interrupt can be generated called `PEN_DOWN IRQ`.
3. This detection is based on Y-axis touch plane tied high, X-axis touch plane tied low, and on the basis of touch the planes are shorted together and Y-axis plane is pulled low.
4. The driver task would not consume any CPU time until the `PEN_DOWN IRQ` event occurs. It would wake up and go into conversion mode only once the user touches screen.

Device Driver

ReCall → Touch Screen System Device Driver

Acquire Stable, debounced position measurements – Reading touch data

1. Create a function named `TouchScan()`. The outline of the procedure would be :-
   (a) Check to see if the screen is touched.
   (b) Take several raw readings on each axis for later filtering.
   (c) Check to see if the screen is still touched.
   (d) (Depending on H/W) store the readings to a memory block, or use FIFO entries.
   (e) (Depending on H/W) if the ADC output is 12bit, either pack data into 16bit (aligned) locations or chop/dither them to 8bit.
   (f) Taking Stable readings (filtering by using oversampling) is necessary for higher level drivers to act appropriately. NOTE :: This filtering could be a h/w function if CPU bandwidth is critical factor.

Calibrate the touch screen.

1. In an ideal scenario, the calibration would be run once during initial product power up and reference values saved in ”non-volatile” memory.

2. Create a function name `CalibrateTouchScreen ()` in case the user wants to calibrate using a graphical target on screen.

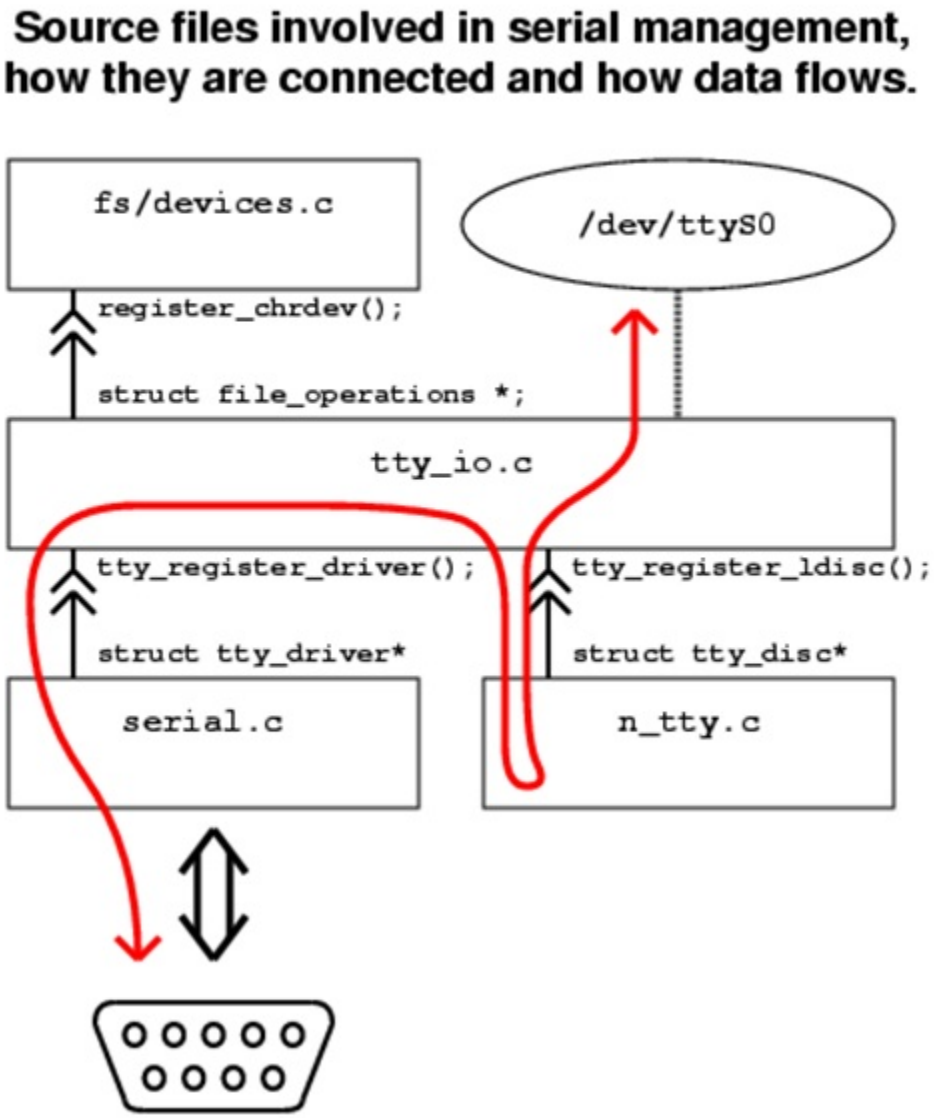
Send changes in touch status and position to the higher level graphics software.

1. Create a function named `GetScaledTouchPosition()`. This is a routine to read raw values and convert them to screen co-ordinates.

2. Create a function named `TouchTask ()`. This routine calls the actual task the user intended to be run while using the touch screen.

Device Driver

Serial Device Driver Example from Linux



Taking Stock

1. Introduction of the course – Harware developments from pure function based to cpu based design. HW - SW codesign flow.

2. Simple example of multiplier for 8085

3. Basic themes of HSCD – Modeling, Analysis and Estimation, System Level Partitioning, synthesis and interfacing, Implementation Generation, Co-simulation and Emulation.

4. Basic Pitfalls – Transient Overloads, Analysis Paralysis, Simulation & complexity, Applications of SoC.

5. Example of Touch Screen Controller

6. Basic Modeling Requirements, leading to SystemC

7. State Chart Diagrams

8. Modeling using SystemC

9. Analysis using Process Path Algorithm(s) and parallel process execution on one CPU.

10. Task Scheduling on one CPU – Rate Monotonic Priority Assignment algorithm, Deadline driven analysis, Mixed Scheduling.

11. Partitioning paper by Kalavade and Lee – Binary Partitioning, Extended Partitioning, Global Criticality Local Phase (GCLP) algorithm.

12. Example of GCD program C $\rightarrow$ Hardware method.

13. FPGA and Emulation systems :: descriptions, differences and their usage

14. Compiler, Linker, Loader :: Basic Steps, Intermediate Format Generation, Challenges for ASISP and ReTargetable Compilers. With examples on point items.

15. Programmers View and SW Development Environment of ARM CortexM3

16. Static Image Processing – Introduction, DCT, Quantization, Huffman Encoding, Run Length encoding. With point examples.

17. Video Image Processing – Motion Vector, Motion Estimation, Frame Types.

18. Device Drivers – Introduction and basic Methodology. Example of Touch Screen, Serial Loopback.

19. Answering Machine Assignment – Alloted time for group discussions

20. Could not cover :: ReConfigurable Computing :: Heterogeneous and Homogeneous Multiprocessor architectures, On Chip Communication Architectures (Network On Chip NoC) concepts

Device Drivers

Acknowledgements

1. Class Notes and Slides from Puneetha Mukherjee

2. Linux Device Drivers :: 3rd Edition