Software Partitioning

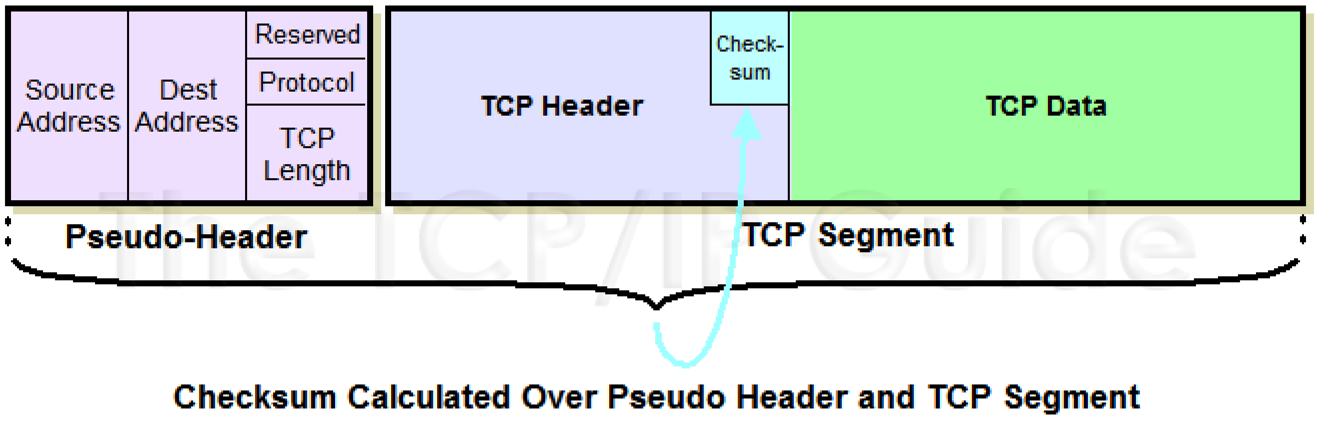
Week 3

Introduction

- We discuss partitioning in communications software, both in terms of functionality and system implementation
- We discuss the concepts of tasks and modules and how a typical communications system is implemented

Software Partitioning Protocol Layers

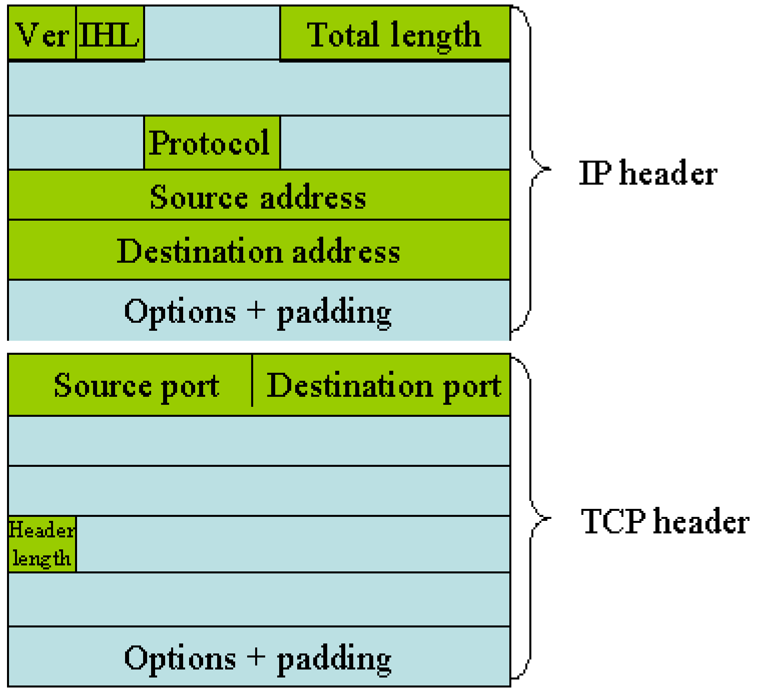
- Protocol Dependencies
  - The required knowledge of src/dst IP address in TCP pseudo-header for TCP checksum is violation of strict layering (RFC 793)



Software Partitioning Protocol Layers

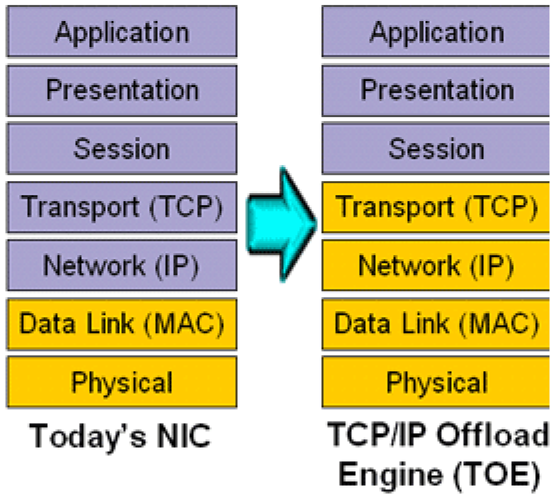
- Performance
  - The knowledge of exact size of downstream protocol headers can avoid copying the frames by starting the TCP packet at an offset from start of IP and Ethernet headers.

Software Partitioning Protocol Layers



Software Partitioning Protocol Layers

- Configuration of H/W and S/W
  - Layers may be implemented in HW -- TCP off-load engines



Task and Modules

- Task and Module based partitioning
  - Task is a thread of execution while a module implements a specific function
  - A process in workstations can have multiple threads of execution. A thread is a lightweight process, which can share access to global data with other threads.
  - In embedded systems with no memory protection, a task is the equivalent of a thread—so we can consider the software for such an embedded system as one large process with multiple threads.

Task and Modules

- Task Implementation
  - Independent functions and those that need to be scheduled at different times can be implemented as tasks
  - Subject to the performance issues related to context switching.
  - The need to keep the timers independent and flexible, can result in TCP and IP being separate tasks
    - TCP Timers: Connection-establishment timer, Retransmission timer, Delayed-acknowledgement timer, Keep-alive timer

Task and Modules

- Task Scheduling
  - Pre-emptive priority based scheduling
    - Lower priority task is pre-empted only by a higher priority task
    - Not for novice programmers.
  - Non pre-emptive scheduling
    - A task relinquishes control to the scheduler
    - Less common in Embedded systems
  - Pre-emptive priority-based scheduling with a time slice for tasks of equal priority

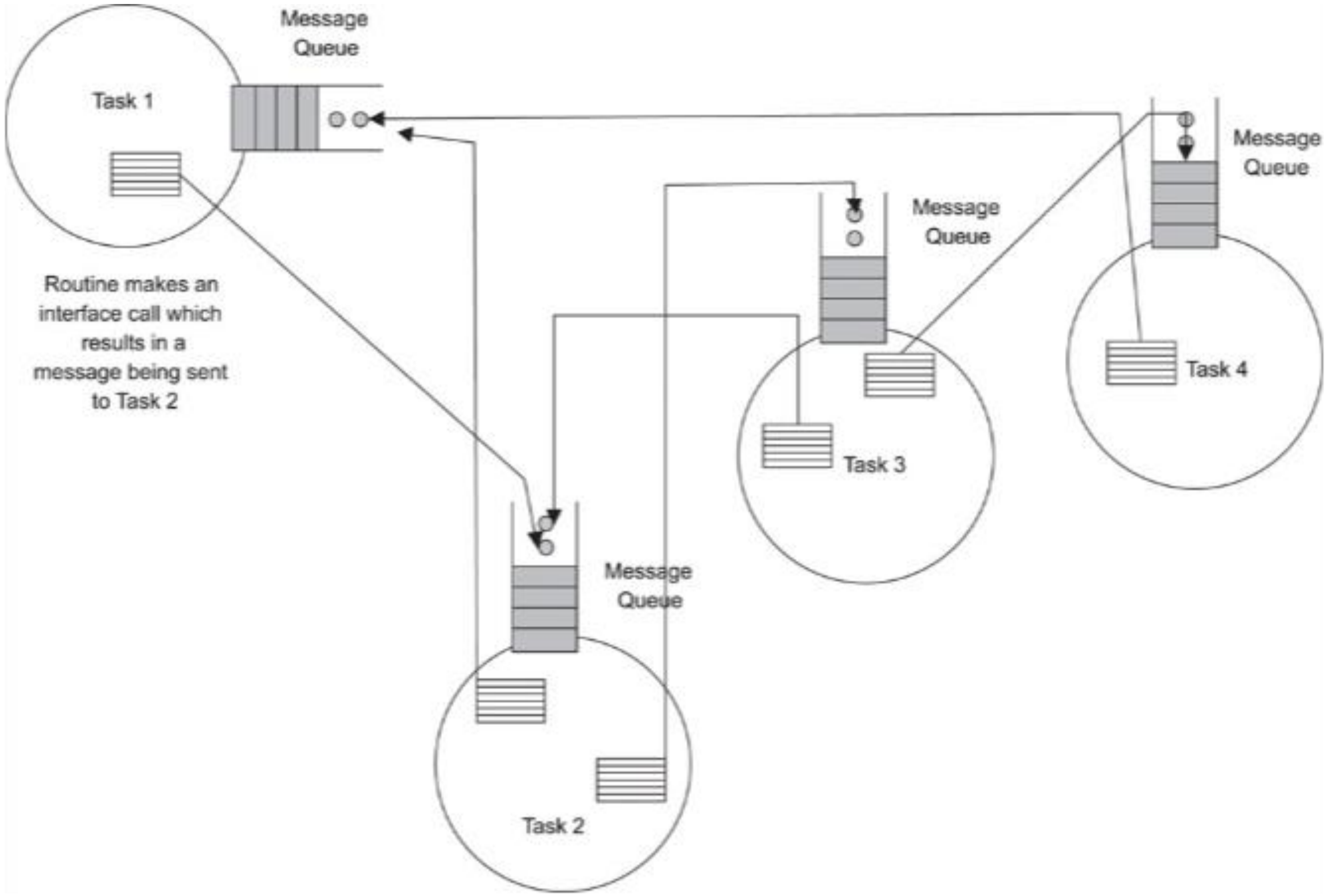
Task and Modules

- Guidelines for organizing the modules and tasks in a communications system
  - Each driver can be implemented as a task if there are one or more Interrupt Service Routines (ISRs) for each driver.
  - The drivers can be implemented as a module when it interfaces with a higher layer task for reception and transmission of data. This must be the single point of entry into the system for received frames (only one task can access the driver).

Task and Modules

- Each protocol can be implemented as a task if it requires independent scheduling and handling of events (e.g. timers).
- Control and data plane functions, along with fast/slow path considerations will be used for grouping tasks and modules.
- SNMP agents and Command Line Interface (CLI) will need to have their own task(s).
- The interfaces between the tasks will be via messages.

Messaging Mechanism Interface


Message
Queue
Task 1
Routine makes an interface call which results in a message being sent to Task 2
Message
Queue
Message
Queue
Task 4
Task 3
Message
Queue
Task 2

Drivers

- Frame Reception
  - Receive buffers are located in system memory so the controller (HW) can DMA (Direct Memory Access) the received frames to system memory.
  - Controller manages frames by utilizing a link list of fixed size buffers to handle data that exceeds the size of a single buffer
    - Multiple buffers and partial space buffer may be used to contain frames that exceeds the size of a single buffer

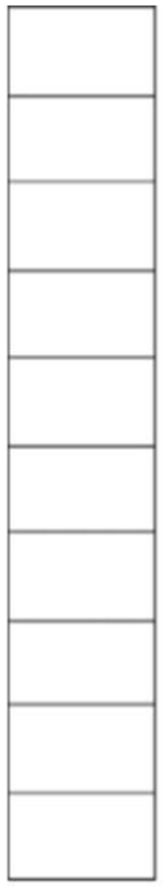
Driver – Received Frames

- Handling Received Frames
  - Received frame is typically enqueued to the higher layer module by the driver without copying the frame into the module's buffers.

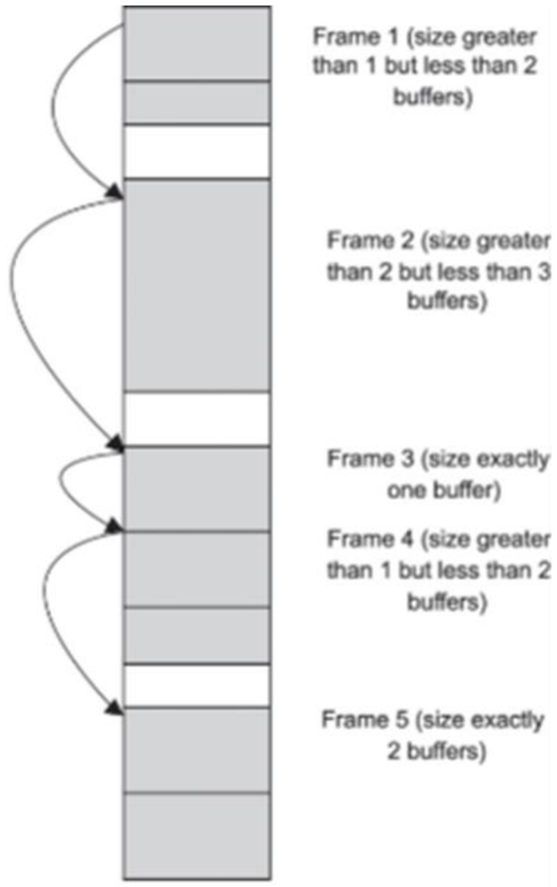
  - An event notifies the higher layer task/module of the presence of the received frame, so that it can start processing the frame

  - If the driver and the higher layer are in two separate memory areas, the driver copies the frame into a common area or into the buffers of a system facility like an Inter-Process Communication (IPC) queue and signals the higher layer.
    - This approach uses an extra copy cycle—which can degrade performance.

Driver – Buffer handling

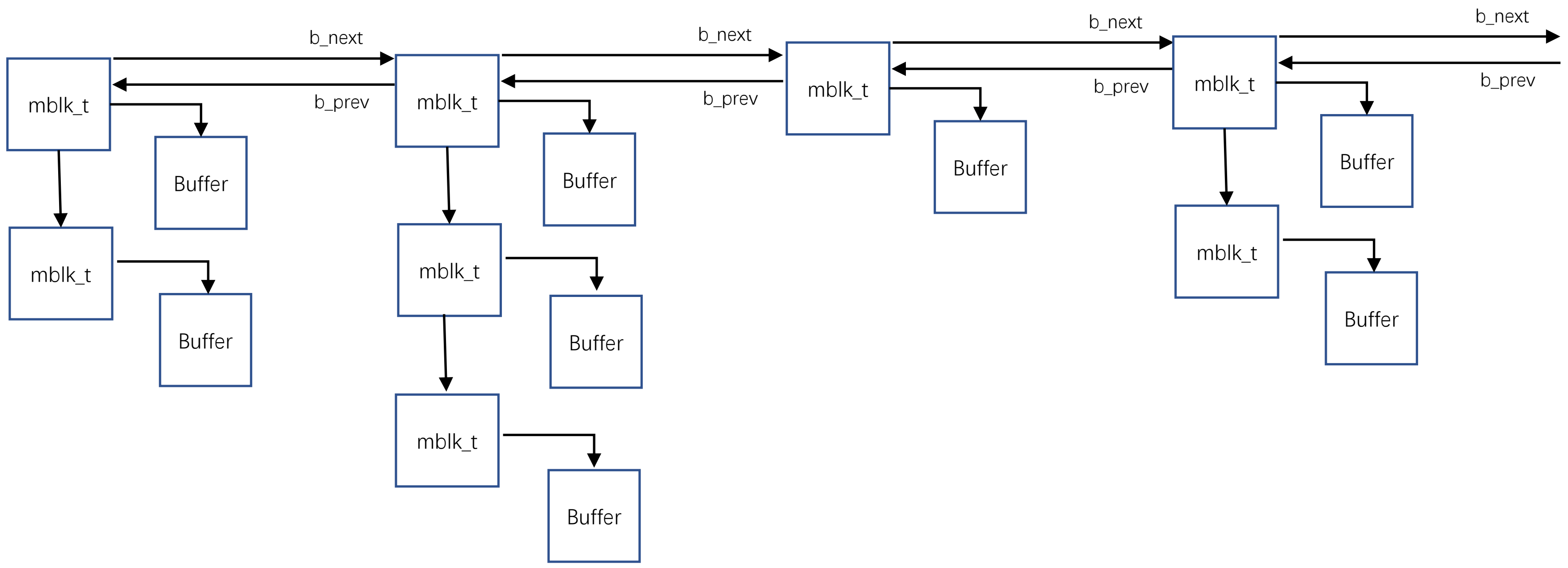


Buffers (1 to 10) before receiving frames


Frame 1 (size greater than 1 but less than 2 buffers)
Frame 2 (size greater than 2 but less than 3 buffers)
Frame 3 (size exactly one buffer)
Frame 4 (size greater than 1 but less than 2 buffers)
Frame 5 (size exactly 2 buffers)

After frame reception - a frame can span more than one buffer. Subsequent frames are linked to the previous frame via a pointer/logical frame count

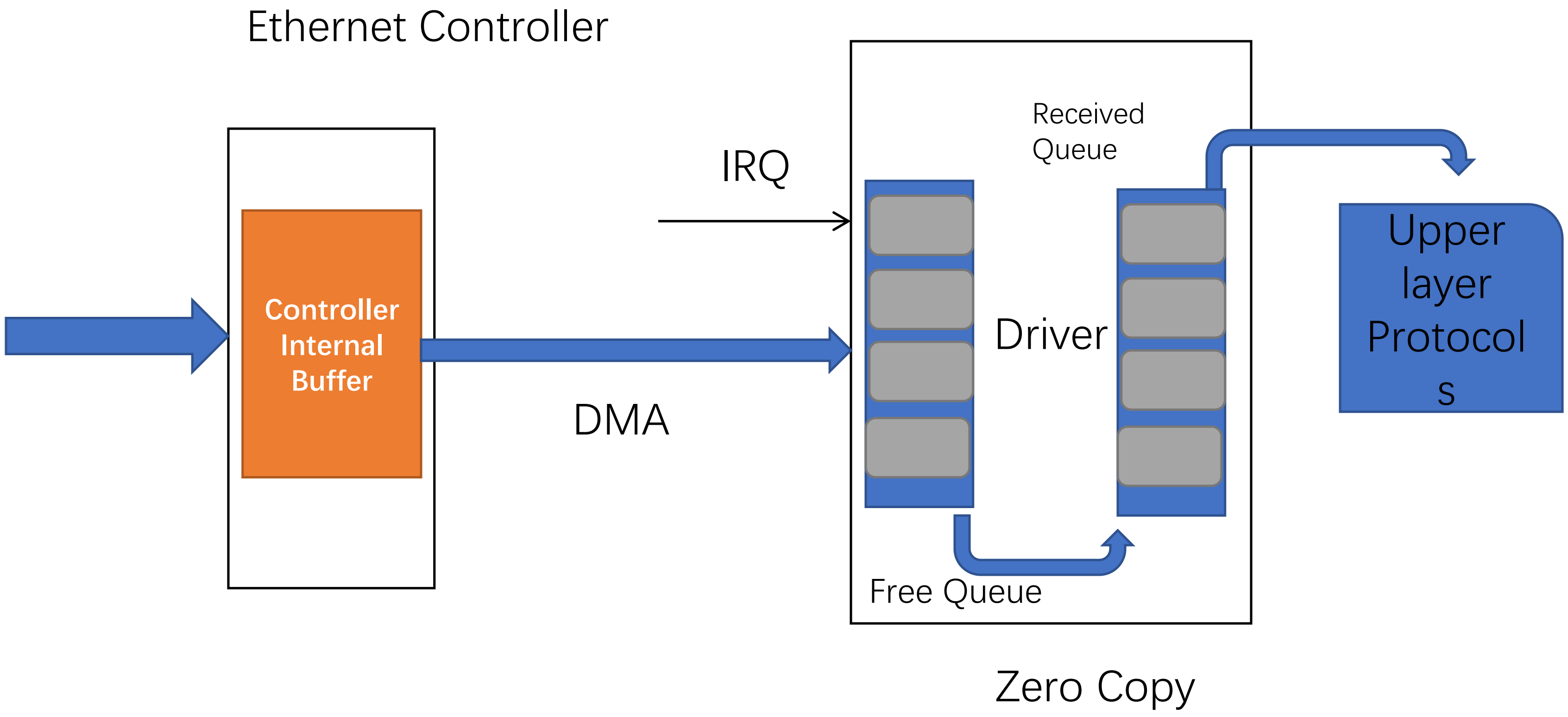
Driver – Buffer handling



Driver – Buffer handling

- Driver may copy the frames from the driver area to the specified area in memory or provide the buffers to the controller for zero-copy performance
    - When a frame is received (informed via interrupt), the driver moves the filled buffer into “received” queue and hands controller a new buffer from “free” queue.
    - It uses the received queue to pass the frames to the higher layer.
- No additional copying of frames- received framed queued in link list of buffers can be processed while receiving the new frames.

Driver – Buffer handling



Driver – Transmitting Frames

- Handling Transmitting Frames
  - Frame transmission also depends on memory and controller functions.
  - The driver can either poll the device or be interrupted on the completion of frame transmission to release the buffers with the data that was transmitted.
  - Alternatively, interrupts may be used only for transmission errors while a regular polling process is used for processing successful transmission completions.

Switching Task

- Control Plane protocols (all) run as single "Switching Task"
    - Avoids context switching and reduced memory requirements
    - It complicates the controlling logic
    - Builds a forwarding table with frames information
    - Switching task needs to run at a rapid rate and is hence a high priority task

Switching Task – Demultiplexing

- Demultiplexing
  - Implemented at a layer above the driver
  - Involves pre-processing of frame data from multiple ports and sending them to appropriate tasks
  - In Layer 2 Switch, Switching task performs the demux operation
  - Classified by the protocol type field in Ethernet frame (IP: 0x0800)
  - If no protocol to deliver, frame can be dropped.

Module and Task Interface

- Consider the “switching task” calls a driver function SendFrame() to transmit a packet
  - The driver provides this routine as part of a driver interface library
  - The higher layer can use it to request the driver to transmit a packet on the Ethernet interface.
  - SendFrame() will execute the steps for queuing the packet to the controller, setting up the transmit registers of the controller

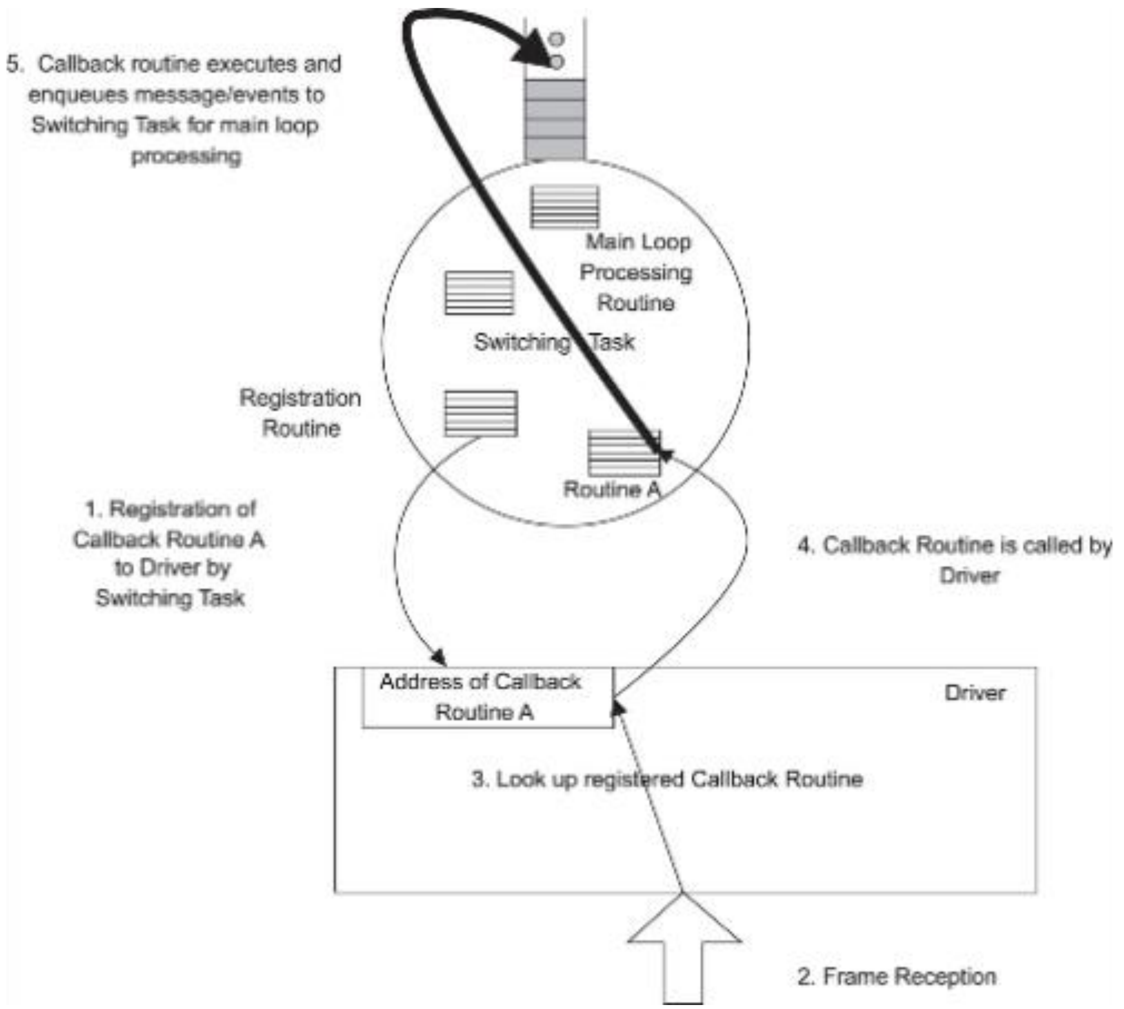
Synchronous and Asynchronous Interface

- Synchronous operation is used when memory space or context is shared by all modules
  - Synchronous operation (call): Function blocks until all its operations have completed.
    - If the function returns immediately after queuing the frame to the controller, the calling function may not know the status of the transmission. Queuing the frames does not mean the transmission was completed.
    - Blocking is not desirable since it can hold up processing of the switching task
    - A solution could be for function to return immediately and for switching task to poll for status

Synchronous and Asynchronous Interface

- Asynchronous operation (callback function):
  - Instead of polling, we could use interrupt mechanism
  - Call back function provides the driver with a function to call when full packet is received.
  - The callback routine, while implemented in the switching task, is called from the Interrupt Service Routine (ISR) by the driver.
  - Though this routine is 'owned' by the switching task, it is always called by the driver-a subtle distinction. The switching task's data is visible only inside ISR.

Functional Interface


5. Callback routine executes and enqueues message/events to Switching Task for main loop processing
Main Loop Processing Routine
Switching Task
Registration Routine
Routine A
1. Registration of Callback Routine A to Driver by Switching Task
4. Callback Routine is called by Driver
Address of Callback Routine A
Driver
3. Look up registered Callback Routine
2. Frame Reception

Functional Interface

- Key requirements of interrupt handlers:
  - ISR must complete their work and return quickly.
  - Since callback routines can be called from an interrupt handler, it is best to keep the callback routine very short.
  - To fulfill above requirement, the callback routine sends an event or notification to the switching task and exits immediately.

Functional Interface

- The main loop of any task implemented in the communications system waits on events like message reception, timer expiration, inter-task communication, and so on.
- Each event is identified by its type and additional parameters.
- On event reception, the task will classify and process the events.

Functional Interface

```
Task Main Loop:
Do Forever
{
    /* Hard Wait - call will block if there are no events */
    Wait on Specific Events;
    /* If we are here, one or more events have occurred */
    Process Events;
}
```

Messaging Mechanism Interface

- Used when memory space or context of modules are separate and not shared
- Messaging systems are synonymous with implementing queuing mechanisms in modules
  - Every module maintains a queue to which other tasks can write to… and use it to transfer messages
  - Messaging systems fit well with multi-board and distributed architectures
  - An event is a special kind of message that includes additional information needed or used by a task
  - An event can also be used to communicate an effect

Messaging Mechanism Interface

Event processing in the main loop.

```
While (1)
{

        Wait for any of the events ( …)

        /* break out of the hard wait loop */
        if (messageQueuing event)
                Process MessageQueue

        if (timer event)
                Process TimerEvents

        if (callBack event)
                Process CallBackEvents; /* based on type + params */

        Perform Housekeeping functions; /* Releasing buffers.. */

}
```

Standards-based Interfaces

- IPC mechanisms can be implemented using Standard Operating System calls
- Standards like POSIX (Portable Operating Systems Interface for Unix) help software to run on different operating systems as is without any changes
- Standards help bring embedded applications to market faster with code that is proven… which in turn leads to prevalence of standards based code
- developers from the desktop programming world can take up embedded programming faster if it’s standards based
- In embedded communication systems, portability of protocols is critical since the hardware can transition from one platform to another

Proprietary Interfaces

- Proprietary Interfaces are system specific
  - Biggest reason for their use is “performance”… since the interface implementation can be tweaked to meet the specifications of the underlying H/W
- When designing proprietary software
  - Constantly review the tradeoff between performance and maintainability critical to any embedded communication system
  - Make every attempt to increase the shelf life of software
  - Make every attempt so migration to newer hardware platforms is as seamless as possible
  - Make every attempt to make it easy for new developers to understand the design and implementation of the application

Homework

- Remember we said earlier:
  - Each driver can be implemented as a task if there are one or more Interrupt Service Routines (ISRs) for each driver. The drivers can also be implemented as a module when it interfaces with a higher layer task for reception and transmission of data.
- Question:
  - How would you implement a receive frame driver using modules? What are the potential problems if this driver is not a task

Homework

- If multiple tasks can use the same controller driver to transmit a packet, how and when would you check if it is safe to write the next packet to the controller’s transmit buffer in your driver?
- What can you do to minimize a wait on ‘transmit done’ event? How would you implement this?

Layer 2 Switch - Drivers

- Switches Ethernet frames between the ports
  - Components that constitute S/W architecture for the switch:
    - Device Driver
    - Protocol Functionality
    - System operation and Management

Layer 2 Switch - Drivers

- Device Driver
  - Closest to H/W and responsible for transmission and reception
  - Reception of frames is either through Polling or Interrupt driven
  - Choice of Polling or Interrupt process is determined by the frequency of packet traffic
  - In Layer 2 Switch, a combination of polling and interrupt is used

Layer 2 Switch - Protocols

- Control Plane protocols in Layer 2 implement control tasks
  - Spanning tree algorithm and protocol (STP)
    - Detects loops in the switching topology and avoids them by de-activating certain ports
    - Transmission of STP frames is initiated by a timer that is maintained by the STP task.
  - Generic VLAN Registration Protocol (GVRP)
    - VLAN (Virtual LAN) provides for a logical partitioning where nodes communicate without use of a router
    - GVRP provides the Layer 2 switch with knowledge of ports and VLAN membership so the nodes can communicate with each other

Management and Control

- Layer 2 switches have full TCP/IP stack functionality to handle configuration, control and monitoring of the switch:
  - TCP over IP -- HTTP over TCP
  - Simple Network Management Protocol (SNMP) over UDP over IP to control, monitor and configure
  - Internet Control Message Protocol (ICMP) functionality like ping and traceroute
- System and Management task
  - SNMP agent permits SNMP manager to control and configure the system
  - Health Monitor task ensures correct operation of H/W and S/W
    - Watchdog timer, port status

Layer 3 Switch

- Layer 3 also termed as IP Switch performs Layer 3 switching
  - Forwarding frames based on layer 3 information
  - Control plane protocols implemented as separate tasks
- Routing Information Protocol (RIP) task
  - Provides information required to build forwarding table
    - Runs on top of UDP
  - Open Shortest Path First (OSPF) Protocol Task
    - Consumes substantial amount of CPU time for SPF calculation
  - Border Gateway Protocol (BGP) task
    - BGP interfaces with TCP and runs on top of it to communicate with peer routers
- Switch Operations more complex compared to Layer 2
- Routers often built as multi-board systems with control card and line cards