DEVICE DRIVER

REJIMOAN R
SCTCE

Device Driver

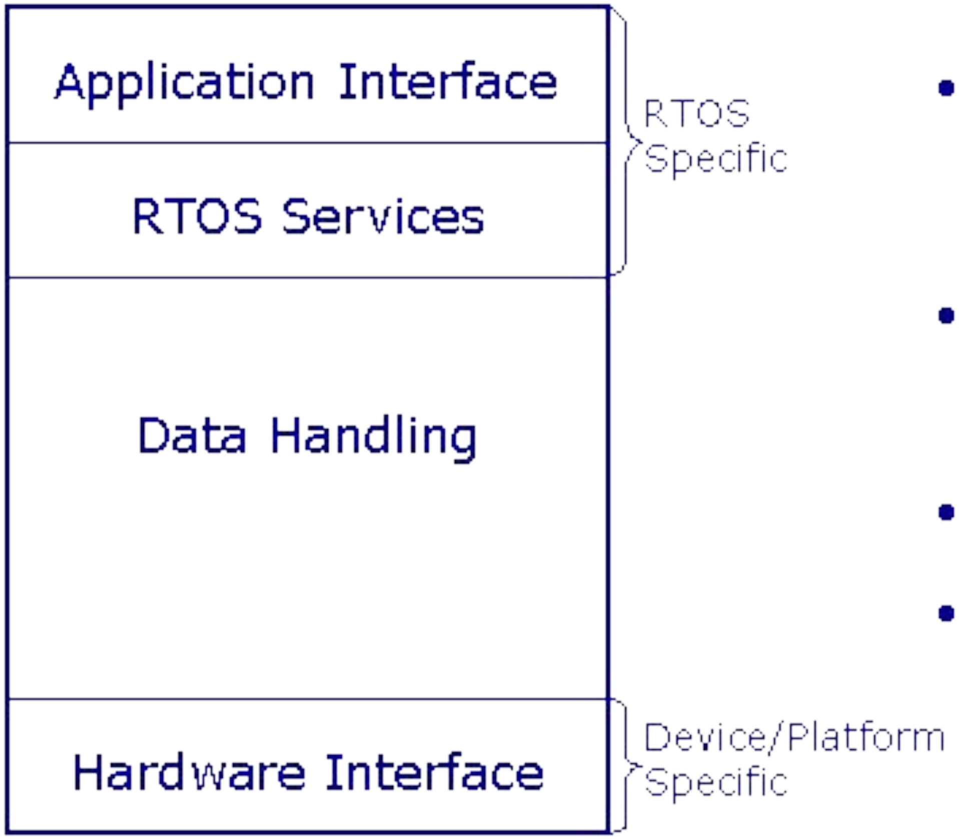
- A device driver is the glue between an operating system and its I/0 devices.
  - Device drivers stand between an operating system and the peripherals it controls.
- Device drivers act as translators, converting the generic requests received from the operating system into commands that specific peripheral controllers can understand.
- The applications software makes system calls to the operating system requesting services(for example, write this data to file x).
- The operating system analyzes these requests and issues requests to the appropriate device driver (for example, write this data to disk 2, block 23654).
- The device driver in turn analyzes the request from the operating system and issues commands to the hardware interface to perform the operations needed to service the request (for example: transfer 1024 bytes starting at address 235400 to disk 2, cylinder 173, head 7, sector 8).

Device Driver

- A device driver contains all the software routines that are needed to be able to use the device.
- Typically a device driver contains a number of main routines like a initialization routine, that is used to setup the device, a reading routine that is used to be able to read data from the device, and a write routine to be able to write data to the device.
- The device driver may be either interrupt driven or just used as a polling routine.

Device Driver

- Drivers accept standard requests for I/O services from the operating system and convert them into the hardware-specific commands and operations required to support the peripheral's interface.
  - This not only relieves the operating system of the burden of handling the details of all the different peripherals, it makes it easy to support another peripheral just by adding a driver and without having to modify the operating system itself.
- The UNIX operating system supports four different types of drivers (block, character, terminal, and STREAMS) which provide different interfaces between the kernel and the driver.
  - This makes it possible to select the most appropriate driver model and interface for the type of device to be supported.
- A driver is implemented as a collection of routines, usually written in the C language, that is linked into the kernel.



- Application Interface
  - Calling Conventions
  - Parameters, e g
    - Device identification
    - Data
- RTOS Services
  - Semaphores
  - Queues
  - Memory Allocation
- Data Handling
  - Formatting
- Hardware Interface
  - Device setup
  - Read/Write/Control

Figure 1.
Typical main parts of a device driver

Major Design Issues

- Issues to be considered are divided into three broad categories:
  1. Operating System/Driver Communications
  2. Driver/Hardware Communications
  3. Driver Operations
- The first category covers all of the issues related to the exchange of information (commands and data) between the device driver and the operating system. It also includes support functions that the kernel provides for the benefit of the device driver.
- The second covers those issues related to the exchange of information (commands and data) between the device driver and the device it controls (j.e., the hardware). This also includes issues such a s how software talks to hardware-and how the hardware talks back.
- The third covers issues related to the actual internal operation of the driver itself

Device Driver Operations

- Interpreting commands received from the operating system
- Scheduling multiple outstanding requests for service
- Managing the transfer of data across both interfaces (operating system and hardware);
- Accepting and processing hardware interrupts;
- Maintaining the integrity of the driver's and the kernel's data structures.

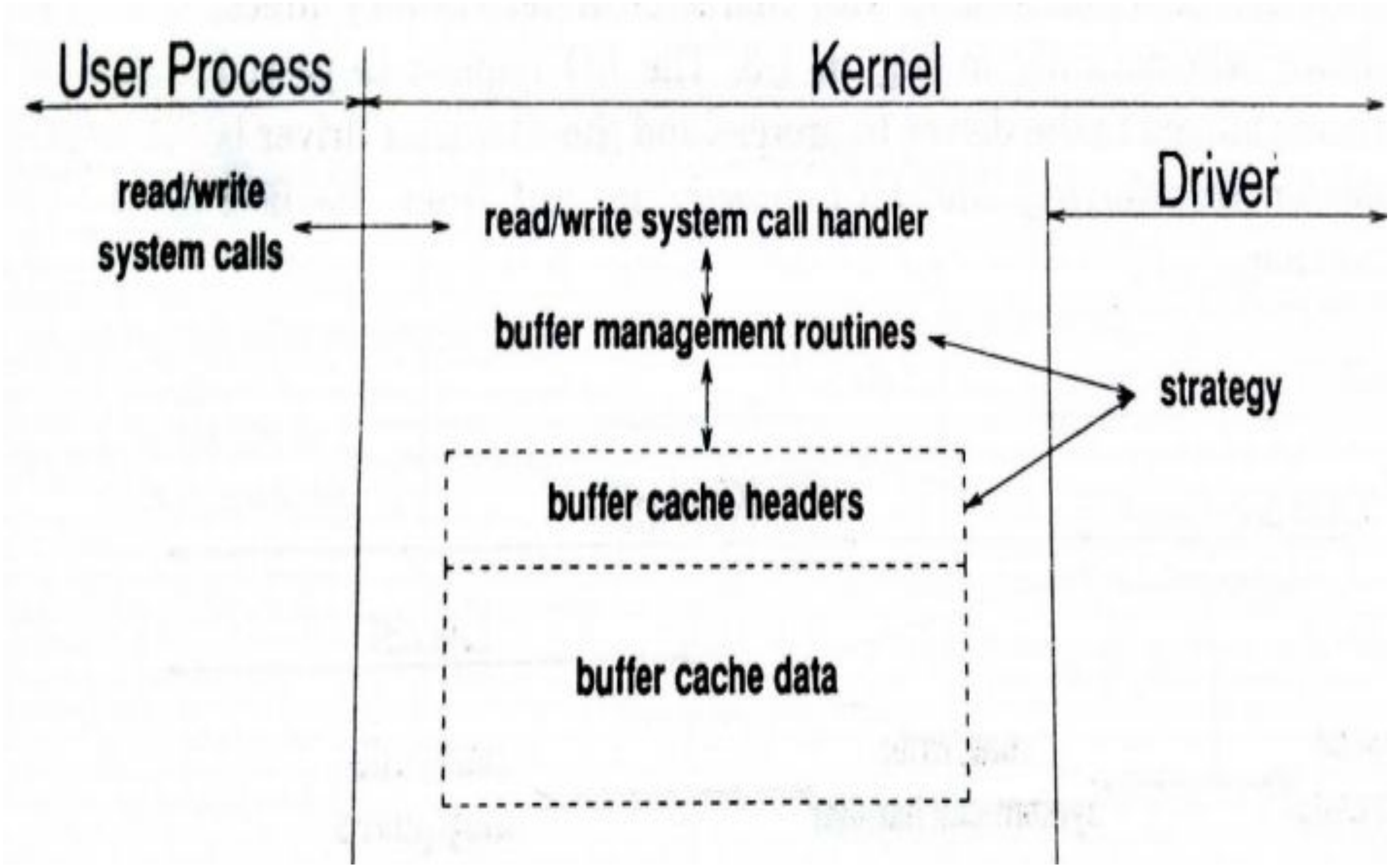
Types of Device Drivers

- UNIX device drivers can be divided into four different types based entirely on differences in the way they communicate with the UNIX operating system.
- The types are: block, character, terminal and Streams.
- The kernel data structures that are accessed and the entry points that the driver can provide vary between the various types of drivers.

Block drivers

- Block drivers communicate with the operating system through a collection of fixed-sized buffers
- The operating system manages a cache of these buffers and attempts to satisfy user requests for data by accessing buffers in the cache.
- The driver is invoked only when the requested data is not in the cache, or when buffers in the cache have been changed and must be written out.
- Block drivers are used primarily to support devices that can contain file systems (such as hard disks).

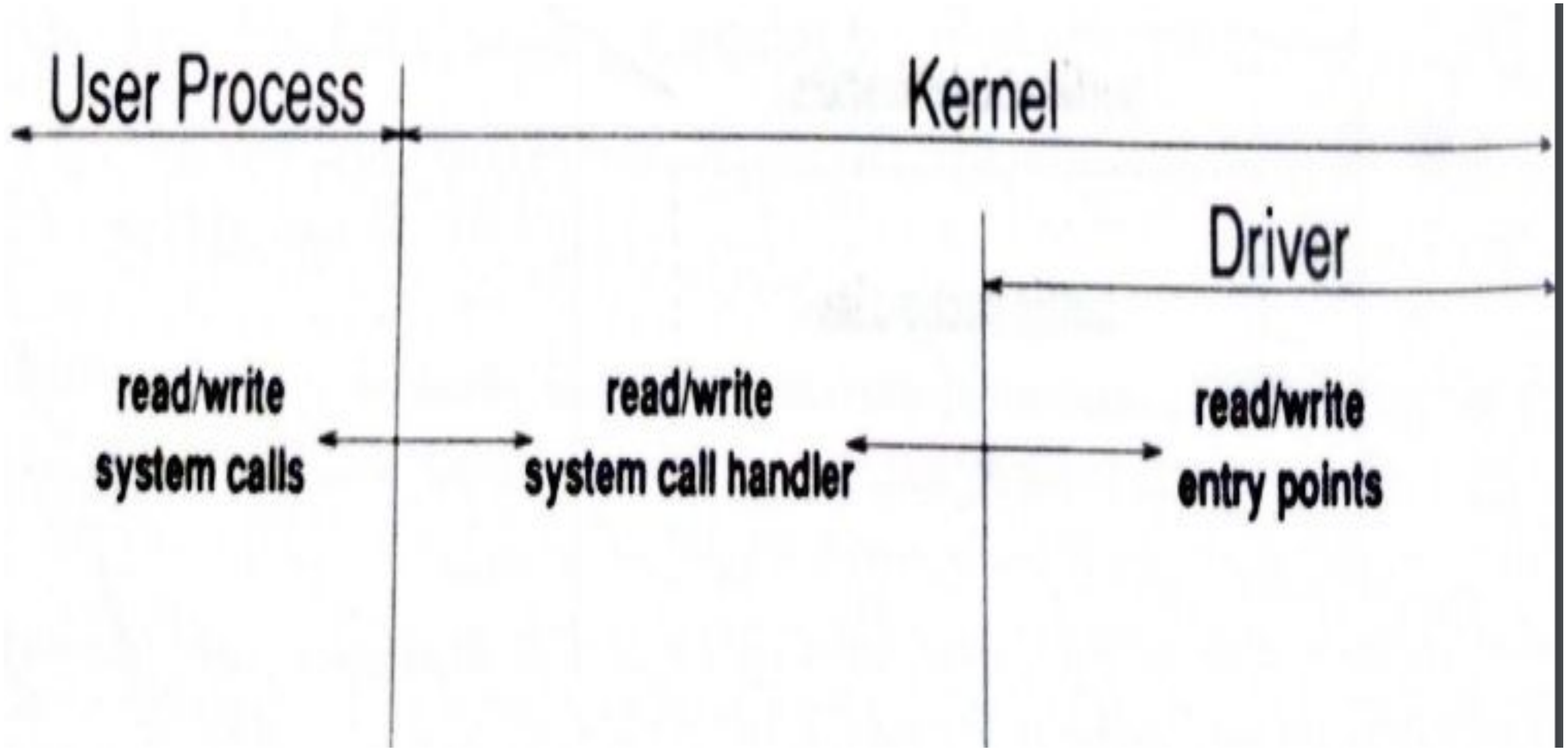
Block drivers



Character Drivers

- Character drivers can handle I/O requests of arbitrary size and can be used to support almost any type of device.
- Usually, character drivers are used for devices that either deal with data a byte at a time (such as line printers) or work best with data in chunks smaller or larger than the standard fixed sized buffers used by block drivers.
- The I/O request is passed to the driver to process and the character driver is responsible for transferring the data directly to and from the user process's memory.
- Differences between block and character drivers
  - User processes interact with block drivers only indirectly through the buffer cache, their relationship with character drivers is very direct.

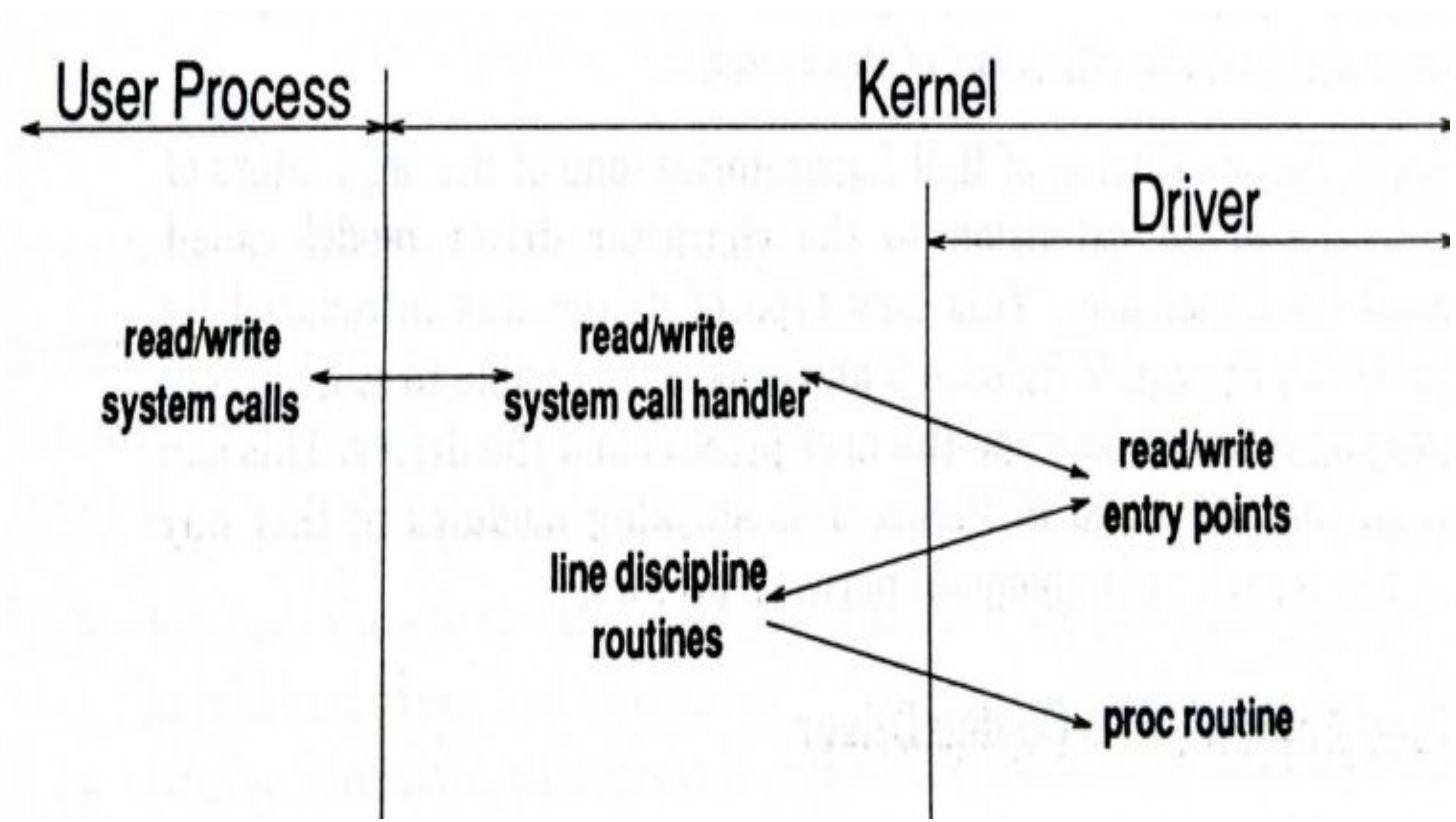
Character Drivers



Terminal Drivers

- Terminal drivers are really just character drivers specialized to deal with communication terminals that connect users to the central UNIX computer system.
- Terminal drivers are responsible not only for shipping data to and from users' terminals, but also for handling line editing, tab expansion, and the many other terminal functions that are part of the standard UNIX terminal interface

Terminal Drivers



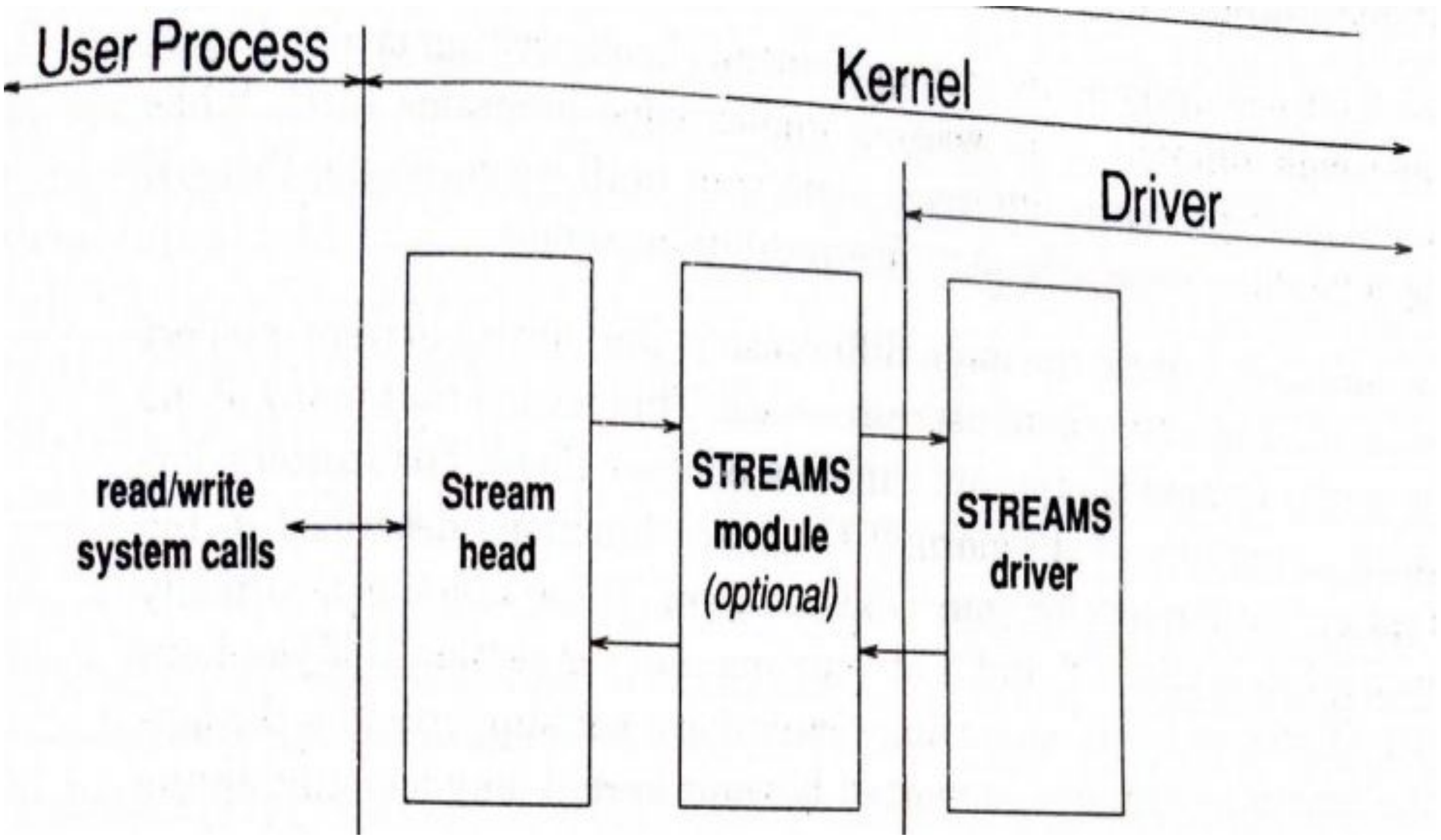
STREAMS Drivers

- STREAMS drivers are used to handle high speed communications devices such as networking adapters that deal with unusual-sized chunks of data and that need to handle protocols.
- STREAMS drivers are also used to interface terminals.
- Networking devices usually support a number of layered protocols. The character model essentially required that each layer of the protocol be implemented within the single driver. This lack of modularity and reusability reduced the efficiency of the system.
- It is much easier to implement network proto-cols.

Anatomy of a Device Driver

- A driver is a set of entry points (routines) that can be called by the operating system.
- A driver can also contain: data structures private to the driver- references to kernel data structures external to the driver; and routines private to the driver (i.e., not entry points).

Anatomy of a Device Driver



Anatomy of a Device Driver

- Most device drivers are written as a single source file.
- The initial part of the driver is sometimes called the prologue.
- The prologue is everything before the first routine.
  - #include directives referencing header files which define various kernel data types and structures;
  - #define directives that provide mnemonic_ names for various constants used in the driver (in particular constants related to the location and definition of the hardware registers);
  - declarations of variables and data structures.
- The remaining parts of the driver are the entry points (C functions referenced by the operating system) and routines (C functions private to the driver.)

Anatomy of a Device Driver

- A device driver has three sides:
  - one side talks to the rest of the kernel
  - one talks to the hardware
  - one talks to the user.
- In order to talk to the kernel, the driver registers with subsystems to respond to events. Such an event might be the opening of a file, a page fault, the plugging in of a new USB device, etc.
- User Interface of a Device driver
  Since Linux follows the UNIX model, and in UNIX everything is a file, users talk with device drivers through device files.
  - Device files are a mechanism, supplied by the kernel, precisely for this direct User-Driver interface.

Basic Device Driver Interface

- Many RTOSs have a standard interface to the device driver like create ( ), open ( ), read ( ), write ( ), ioctl ( ).
- Other RTOSs may have their own propriety interface with the same type of functions but with different names and a faster access time to the device.
- As many RTOSs offer an interface to be able to call the device drivers, the device drivers and the application do not have to be linked together.
- This makes it easier to be able to create hardware independent applications.

Device Driver Design Patterns

- Subsystem maintainers will ask driver developers to conform to the design patterns such as
  - State Container
  - container_of()

State Container

- While the kernel contains a few device drivers that assume that they will only be probed() once on a certain system, it is custom to assume that the device the driver binds to will appear in several instances.
- This means that the probe() function and all callbacks need to be re-entrant.

container_of()

- container_of() is a macro defined in <linux/kernel.h>
- What container_of() does is to obtain a pointer to the containing struct from a pointer to a member which allows object oriented behaviours.
- The contained member must not be a pointer, but an actual member

- When you design your system it is very good if you can split up the software into two parts, one that is hardware independent and one that is hardware dependent, to make it easier to replace one piece of the hardware without having to change the whole application. In the hardware dependent part you should include:
  - Initialization routines for the hardware
  - Device drivers
  - Interrupt Service Routines

- The device drivers can then be called from the application using RTOS standard calls. The RTOS creates during its own initialization tables that contain function pointers to all the device driver's routines. But as device drivers are initialized after the RTOS has been initialized you can in your device driver use the functionality of the RTOS.
-
- When you design your system, you also have to specify which type of device driver design you need. Should the device driver be interrupt driven, which is most common today, or should the application be polling the device? It of course depends on the device itself, but also on your deadlines in your system. But you also need to specify if your device driver should called synchronously or asynchronously.

Synchronous Device Driver

- When a task calls a synchronous device driver it means that the task will wait until the device has some data that it can give to the task
- The task is blocked on a semaphore until the driver has been able to read any data from the device.

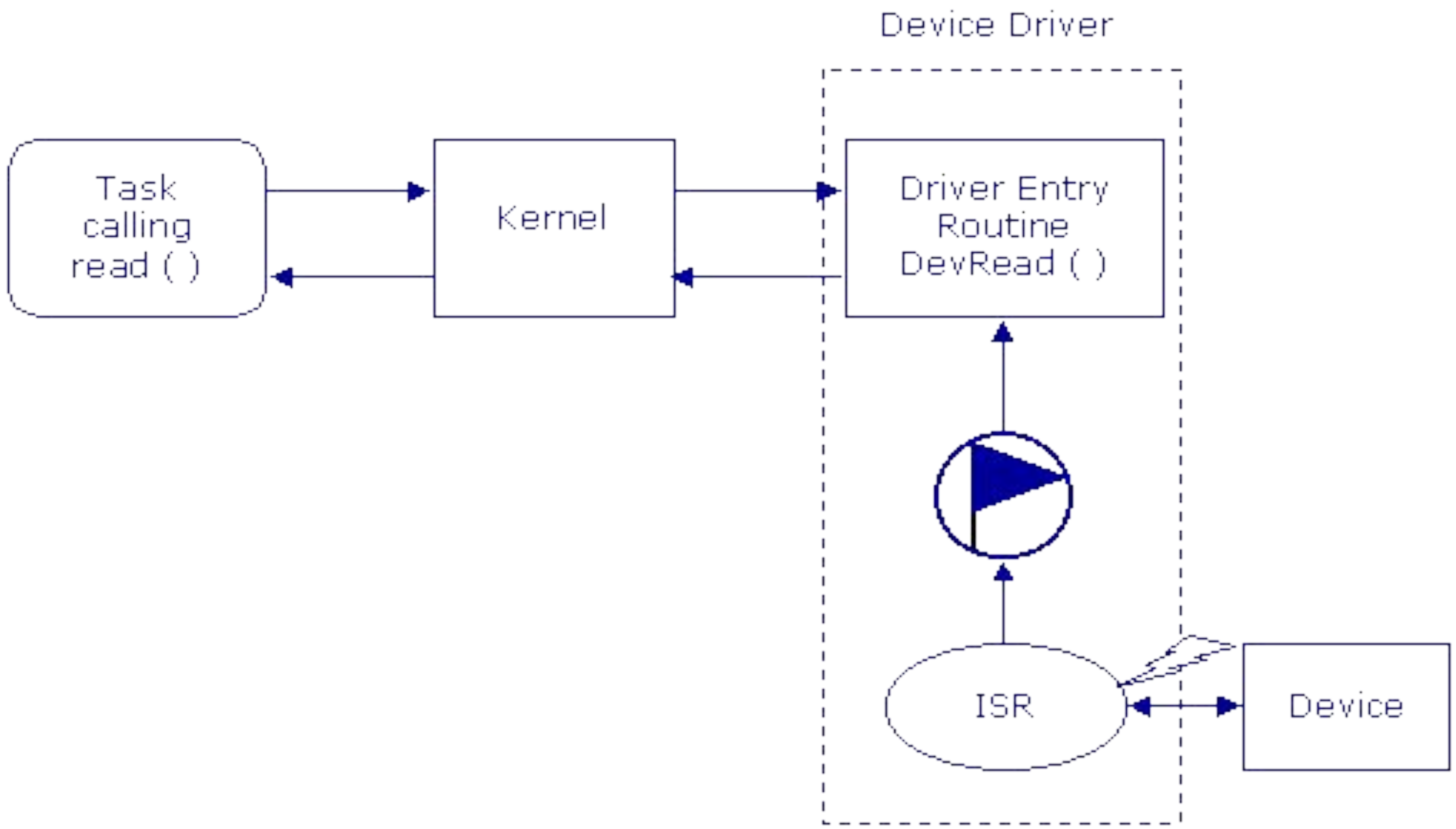


Figure 2. Synchronous device driver
The Task calls the device driver via a kemel device call. The Device Entry Routine gets blocked on a semaphore and blocks the task in that way. When an input comes from the device, it generates an interrupt and the ISR releases the semaphore and the Device Entry Routine returns the data to the task and the task continues its execution.

Asynchronous Device Driver

- When a task calls an asynchronous device driver it means that the task will only check if the device has some data that it can give to the task
- The task is just checking if there is a message in the queue. The device driver can independently of the task send data into queue.

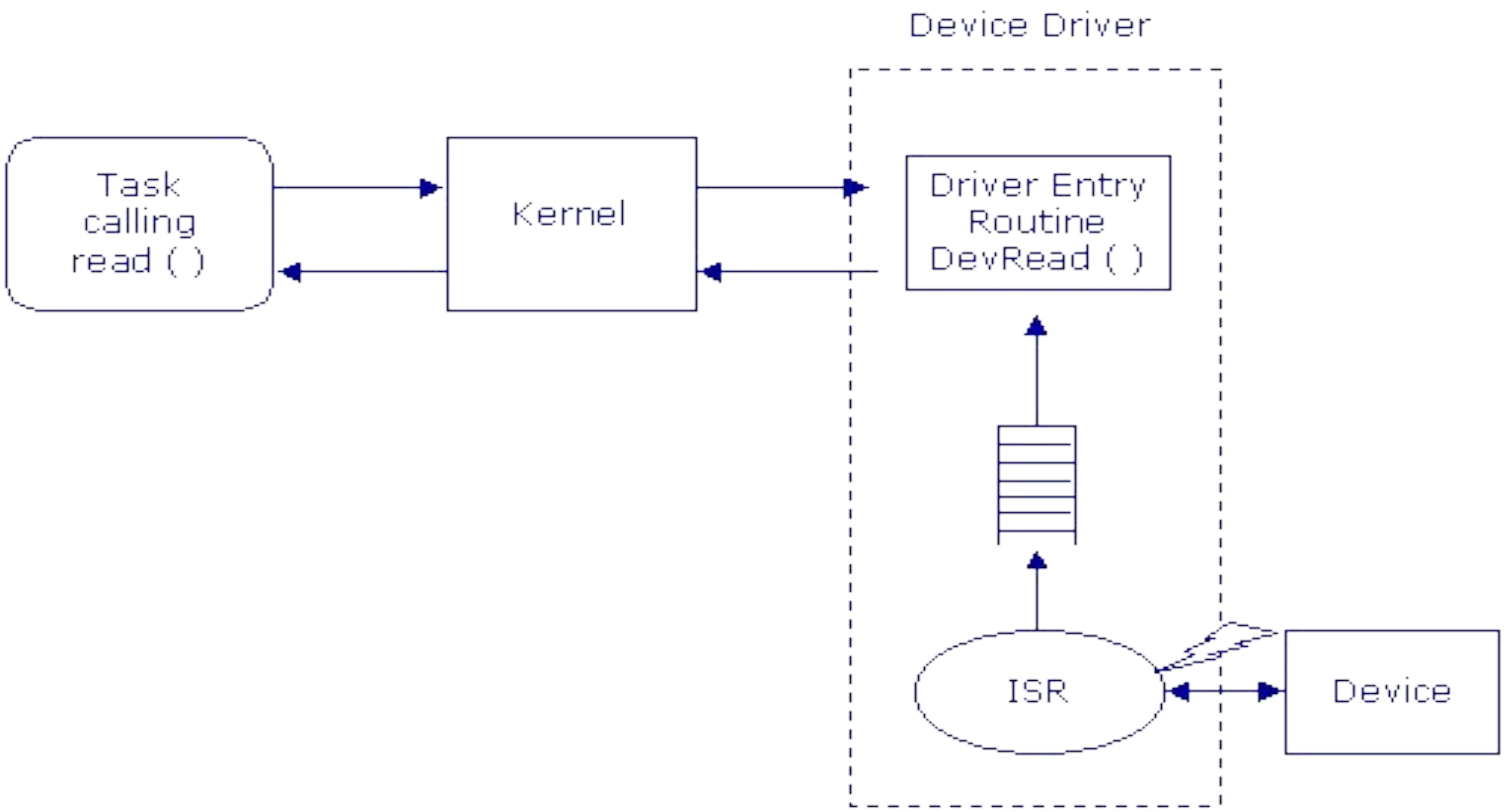


Figure 3. Asynchronous device driver
The Task calls the device driver via a kernel device call. The Device Entry Routine receives a message from the queue and returns the data to the task. When new data arrives from the device the ISR puts the data in a message and sends the message to the queue.

Latest Input only

- Sometimes a task only wants to read the actual data from the device, e.g. the speed of a motor or the actual temperature of the oil.
- Instead of using a queue for the data, we use a shared memory area protected by a semaphore

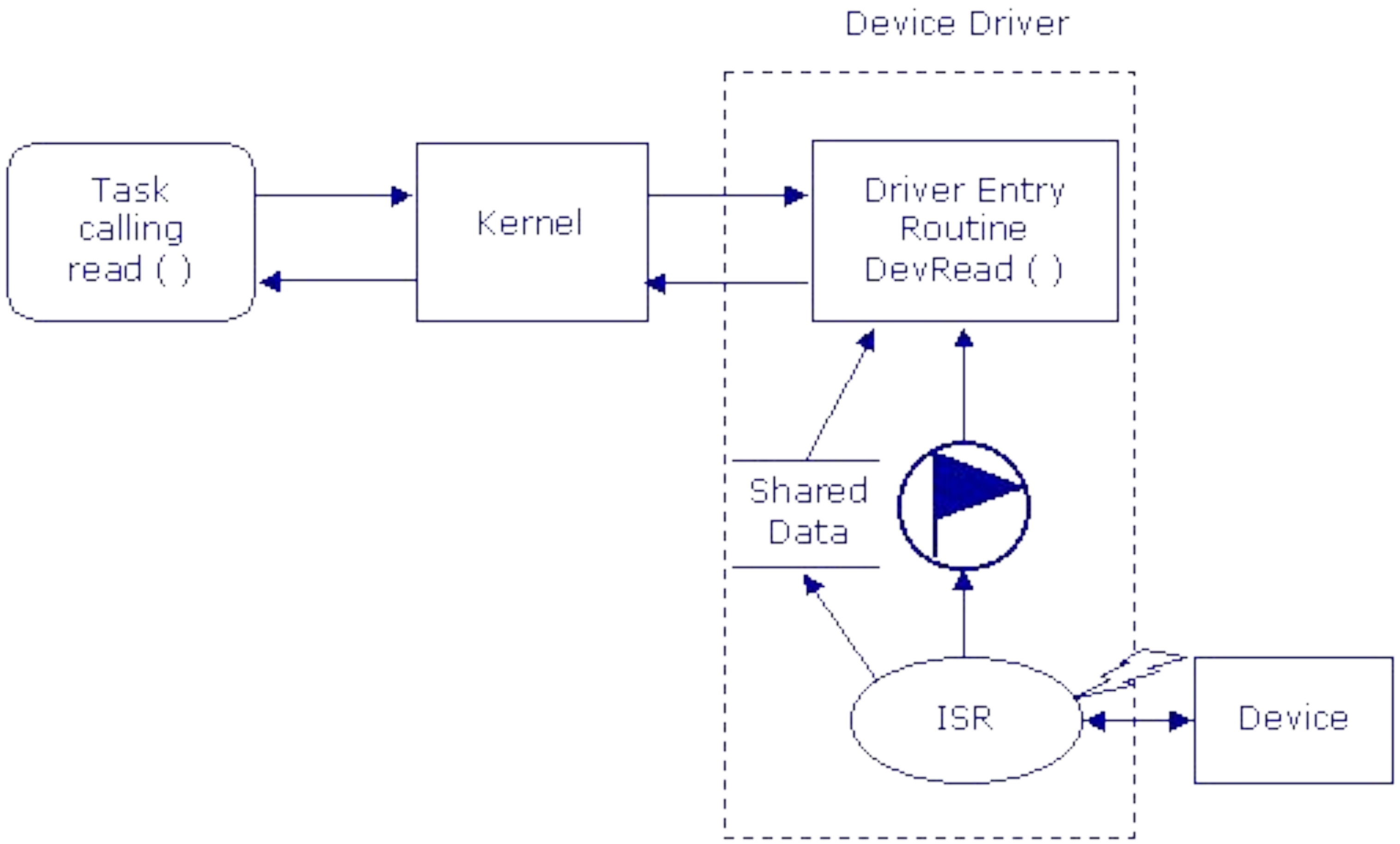


Figure 4. Latest Input Only
The Task calls the device driver via a kernel device call. The Device Entry Routine tries to acquire the semaphore to be able to read the shared data. When data arrives the ISR tries to acquire the semaphore to be able to write to the shared data. The designer has to decide how the ISR should react if the semaphore is busy.

Serial Input and Output Data Spooler

- If the device driver should be able to handle blocks of data by itself, the device driver needs to have internal buffers for storing data.

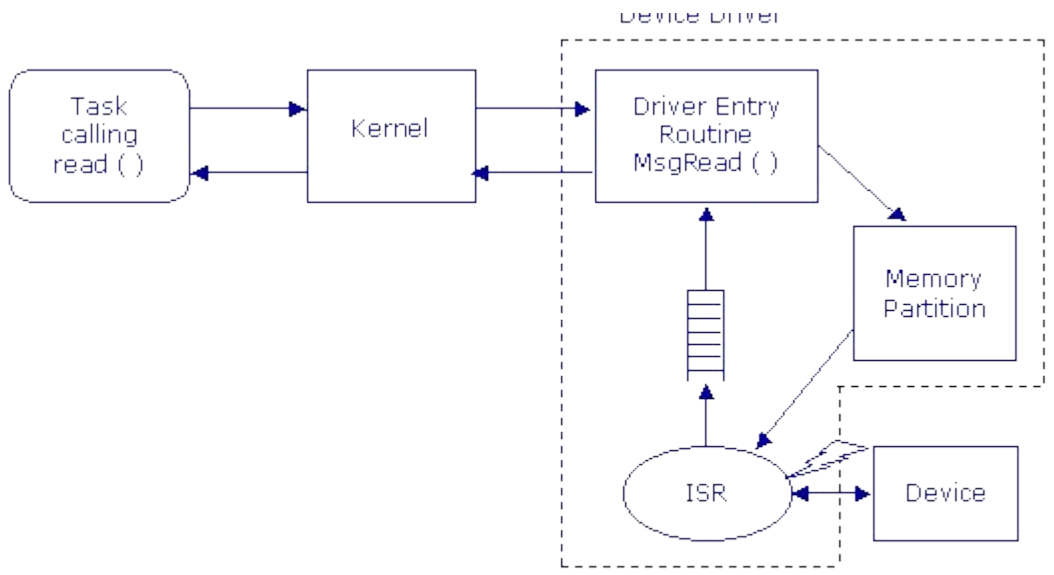


Figure 5. Serial Input Data Spooler
The Task calls the device driver via a Kernel device call. The Device Entry Routine receives a message from the queue and returns the data to the task and returns the buffer to memory partition. When new data arrives from the device the ISR allocates a buffer from a memory partition, puts the data in the buffer and puts a pointer to the buffer in a message and sends the message to the queue.

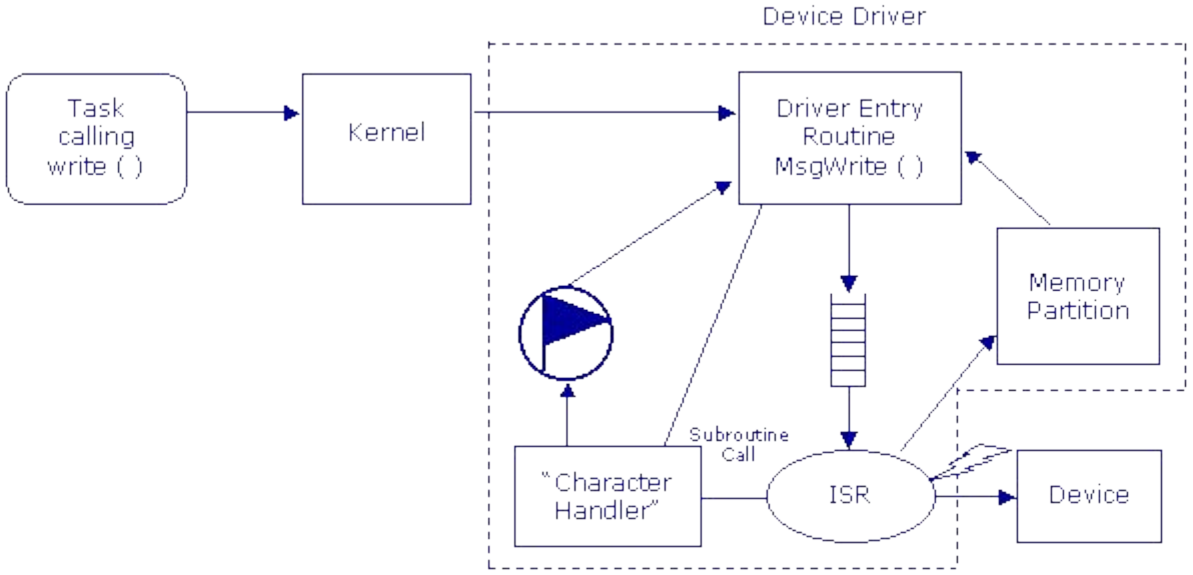


Figure 6. General Output Data Spooler
The Task calls the device driver via a Kernel device call. The Device Entry Routine allocates a buffer from a memory partition, puts the data in the buffer and puts a pointer to the buffer in a message and sends the message to the queue. When new data arrives to the queue the ISR receives a message from the queue, sends the data to the device and returns the buffer to memory partition. If the queue becomes empty = no more interrupts will arrive, then must the Device Entry Routine call the "Character Handler" to start sending of data again. The ISR now only handles the general interrupt handling and the "Character Handler" handles sending of data to the device.