Many embedded systems developers will tell you that writing a device driver consists of a lot of "bit-bashing and register-twiddling" to convince some ornery unit of hardware to submit to the control of driver software. You've got to get every one of a myriad of details right -- the bits, the sequences, the timing -- or else that chunk of hardware will just refuse to do its thing. Traditionally, the focus in writing device drivers has been at this nuts-and-bolts level. But I would like to take a somewhat different, higher-level view of device input/output driver software.

Nowadays embedded systems software developers take advantage of the services of a real-time operating system (RTOS) to structure their device driver software. A device driver can be organized as a collection of "chunks" of concurrent software. The RTOS is involved in both scheduling the "chunks" and allowing them to communicate cleanly with one another. In different operating systems, the concurrent "chunks" of software might be given different names like 'threads' or 'tasks'. For example, in the pSOSystem RTOS they are called 'tasks' and 'Interrupt Service Routines' (ISRs).
MUTUAL EXCLUSION OF DEVICE ACCESS

One of the most basic requirements of a device driver might be the need to ensure that only one application task at a time can request an input or output operation on a specific device. For example, if you've got an application task which needs a temperature measurement in units of degrees Celsius and another task which uses degrees Kelvin, you had better make sure that only one task at a time is asking for a temperature measurement.

An easy way to ensure this is with a semaphore, operating in binary fashion. Make sure that each application task obtains the semaphore's token before initiating a temperature measurement. And make sure that it returns the semaphore's token at the end of the temperature measurement.

Usually, this can be done right inside the device driver. The driver requests the semaphore's token as soon as it is called by an application task. And after the completion of the I/O operation, the driver returns the semaphore's token.

For devices which may be thought to be "session-oriented", exclusive access must be granted for an entire "session" which may consist of many individual I/O operations. For example, a single task might want to print an entire page of text, whereas the printer driver's output operation prints only a single line of text. In such a case, a driver's 'open session' operation would request the semaphore token. And a 'close session' operation would return the semaphore token. [The "session" semaphore would initially contain one semaphore token, to indicate "session available".] Any other task attempting to 'open session' while the semaphore's token is unavailable, would be denied access to the driver software.
SYNCHRONOUS VS. ASYNCHRONOUS I/O MODELS

A much larger question in structuring a device driver, is the question of synchronous versus asynchronous driver operation. To put it another way: Do you want the application task which called the device driver to wait for the result of the I/O operation that it asked for? Or do you want the application task to continue to run while the device driver is doing its I/O operation?

The device driver's architecture will be of vastly differing structure for each of these alternatives.

SYNCHRONOUS I/O DRIVERS

In a synchronous driver, the application task which called the device driver will wait for the result of the I/O operation that it asked for.

This does not mean that your entire application will stop and wait while the driver is working with the I/O hardware to perform the I/O operation. Other tasks will be allowed to continue working, at the same time that the I/O hardware is working. Only the task which actually called the driver will be forced to wait for the completion of the I/O operation.

Synchronous drivers are often simpler in their design than other drivers. They are built around a mechanism for preventing the requesting task from executing while the driver and I/O hardware are working; and then releasing the requesting task for continued execution when the driver and I/O hardware have completed their work.

This can be done with a (binary) semaphore. [Please note that this is a different semaphore than the binary semaphore described earlier for purposes of mutual exclusion of tasks. So a synchronous driver might actually contain 2 (or more) binary semaphores.] At driver initialization time, this new semaphore would be created but not given any semaphore tokens. An attempt to get a semaphore token when none is available would force the requesting software to stop executing.
Using pSOS+ for example, this is done by having the entry-point area of a driver behave essentially like a subroutine of the requesting task. So, for example, if a task calls a driver via a ‘read from device’ call, the entry area of the driver implementing the call would act much as a subroutine of the task. And if the entry area of the driver attempts to get a semaphore token which is not present, the entry area of the driver would be blocked from continuing execution; and together with it, the requesting task would be put into the blocked state.

We can see this in the diagram below. The requesting task is shown on the upper left as a rectangle with rounded corners. Beneath the semaphore, is an ellipse representing an ISR. The "lightning" symbol represents the hardware interrupt which triggers execution of the ISR.

The ISR is considered part of the driver. Its job is to execute when an interrupt arrives announcing that the hardware has completed its work. In this example, the interrupt announces that the hardware has completed reading a new input. When the ISR executes, its main function is to create a new semaphore token (out of "thin air", if you will) and to put it into the semaphore upon which all the rest of the software here is waiting. The arrival of the semaphore token releases the software on the upper left from the blocked state; and when it resumes executing it can take the final results of the hardware I/O operation and begin processing.
The entry area of the driver (shown as a rectangle in the upper center of the diagram), does the logic described in the following pseudocode when called by a task:

```plaintext
DevRead_entry:
    BEGIN
        Start IO Device Read Operation;
        Get Synchronizer Semaphore Token (Waiting OK);
        /* Wait for Semaphore Token */
        Get Device Status and Data;
        Give Device Info. to Requesting Task;
    END
```

The ISR has very simple logic:

```plaintext
DevRead_ISR:
    BEGIN
        Calm down the hardware device;
        Put a Token into Synchronizer Semaphore;
    END
```

Now let's complicate the situation....

**ASYNCHRONOUS I/O DRIVERS**

In an asynchronous driver, the application task which called the device driver may continue executing, without waiting for the result of the I/O operation it requested.

This is true parallelism, even in a single-CPU hardware environment. Your task can continue to execute at the same time that hardware is executing the I/O operation which that task requested.

Asynchronous drivers are more complex in their design than other drivers. In some cases, an asynchronous driver might be "overkill". You need to ask yourself the question, "If my task requests an I/O operation through a driver, then what work can it usefully do before that I/O operation is done?" Occasionlly the answer may be "No, I need the I/O completed before my task can usefully do anything else.". Such an answer says than asynchronous driver is overkill.

For example, say you're designing a driver for an input device. What can a task do with that input before it's ready?? Not much of anything!
But perhaps we can set things up so that every time a task asks the device driver for a new input, two things happen:
(a) The driver asks the I/O hardware to start getting a new input; and
(b) The driver gives the requesting task the last previous input to work on in the meanwhile.
Sometimes this may be a useful way to work. So here's what the design of such a driver would look like:

The new ladder-like symbol appearing here represents a message queue. It's a place to store information about a previous input. The entry area of the driver can get a previous input from the queue, to give to the requesting task. And the ISR will put new input into this queue whenever it gets one.
The entry area of the driver does the logic described in the following pseudocode when called by a task:

```plaintext
DevReadAsync_entry:
BEGIN
    Get Message from the Queue (Waiting OK);
    /* Wait if Queue is Empty */
    Start new IO Device Read Operation;
    Give old Device Info. to Requesting Task;
END
```

The ISR has this logic:

```plaintext
DevReadAsync_ISR:
BEGIN
    Calm down the hardware device;
    Get Data/Status Info. from Hardware;
    Package this Info. into a Message
    Put Message into the Queue;
END
```

In order for this to work, a driver initialization operation needs to create the Message Queue which is at the heart of this driver. If the hardware input device never delivers an input unless it is requested to do so by software, the maximum length (depth) of this queue will be one message.
LATEST INPUT ONLY ASYNCHRONOUS DRIVER

If a hardware input device is free to deliver inputs even when not explicitly requested by software, the asynchronous design we have just seen may not be very good. The problem is that old inputs would queue up in the Message Queue. Requesting tasks would be fed very old inputs, while newer inputs would languish in the message queue.

So for a free-running input device, a better driver architecture might be that shown below.

This design shows a protected shared data area at its center. The shared data store, shown as a pair of parallel horizontal lines, always contains the latest input value. It is protected from data corruption and access collisions by its associated (binary) semaphore.

Whenever the ISR is triggered by a new interrupt, it gets new input data from the hardware, and overwrites the previous content of the shared data area. In order to do this cleanly, the ISR must obtain access permission from the associated semaphore.
Whenever a task requests input data from the driver, the driver entry software reads it from the shared data area, after obtaining access permission from the associated semaphore. Since old input data is overwritten by the ISR, the data being read and fed to the requesting task is always the "freshest" data available.

The entry area of the driver does the following when called by a task:

DevReadLatest_entry:
BEGIN
  Get Shared Data Access Semaphore (Waiting OK);
  /* Wait for Semaphore Token */
  Read latest input from Shared Data area;
  Return Shared Data Access Semaphore;
  Pass input data on to requesting task;
END

The ISR does this:

DevReadLatest_ISR:
BEGIN
  Calm down the hardware device;
  Get Shared Data Access Semaphore (No Waiting);
  /* ISRs should never wait */
  IF Semaphore OKs access
  THEN
    Get Data/Status Info. from Hardware;
    Write latest input to Shared Data area;
    Return Shared Data Access Semaphore;
  ELSE ...
  ENDIF
END

In rare instances, this ISR will be unable to obtain the semaphore it needs to access the Shared Data area. These will be instances of the two 'sides' of the design trying to access the Shared Data area simultaneously. In these instances, it should not attempt to write into the Shared Data area, as that would very likely cause corrupted data to be delivered to the requesting task. The device driver architect will need to decide how the ISR will handle unavailability of access to the Shared Data area.
SERIAL INPUT DATA SPOOLER

Very often a hardware input device can deliver inputs freely to a computer without its being explicitly requested by software. We would like to capture and process all of the incoming information, without losing any -- even if it is arriving in irregular bursts.

An example of this is the arrival of character strings from a serial line. Every character is a byte of incoming data, announced to the CPU by an interrupt.

Message queues are good for buffering irregular bursts. However a message queue might impose too much performance penalty on a driver if it were to hold each arriving character in a separate message. So perhaps it would be better to use a message queue to hold pointers to larger buffers which would contain complete character strings. If the serial line never delivers strings of length greater than 'S' characters, then buffers of length 'S' bytes can be used for all strings.

Many real-time operating systems have a Memory Partitions service which can manage buffers of standard sizes. The ISR part of the driver could "borrow" a buffer from a Partition of appropriate buffer size, and fill that buffer with a character string. And then put a pointer to that buffer into the Message Queue, for transfer to the non-ISR part of the driver. We see this pictured in the following diagram:
The entry area of the driver does the following when called by a task:

```
DevInSpool_entry:
BEGIN
    Get Message from the Queue (Waiting OK);
    /* Wait if Queue is Empty */
    Extract string information from message;
    Give string to Requesting Task;
    Return buffer to its Partition;
    /* When buffer no longer needed */
END
```

The ISR has this logic:

```
DevInSpool_ISR:
BEGIN
    Calm down the hardware device;
    Get new character from Hardware;
    IF in the middle of a string
        THEN Put next character into buffer
    ELSE /* Need to start on a new string */
        Put buffer pointer into a message;
        Put character count into this message;
        Put this message into Queue;
        Request new buffer from Partition;
        /* ISRs should never wait */
    ENDIF
END
```

In some instances, this ISR will be unable to obtain the memory buffer it needs from the Partition, to hold a new character string. The device driver architect will need to decide how to handle unavailability of buffer memory. In other instances, the ISR will be unable to send its message since the Queue may be full. The device driver architect needs to design a solution to this as well.
For many output devices, asynchronous drivers have clear advantages over synchronous drivers. While the asynchronous driver is working with its I/O device hardware to complete one output operation, the requesting task can already be preparing for the next output operation.

For example, a task may be preparing strings of text for printing while at the same time the printer driver is printing out previously prepared strings. This sort of driver is often used to allow numerous tasks to prepare and queue up their outputs. Queuing of outputs is typically in FIFO order.

The driver design shown below for an asynchronous printer driver, is quite similar to the device input spooler shown earlier. Two differences are the directions of access to the Message Queue and Memory Partition.
The entry area of the driver does the following when called by a task:

```c
DevOutSpool_entry:
BEGIN
    Request new buffer from Partition;
    Fill buffer with string for printing;
    Put buffer pointer into a message;
    Put character count into this message;
    Put this message into Queue;
END
```

The ISR has this logic:

```c
DevOutSpool_ISR:
BEGIN
    Calm down the hardware device;
    IF in the middle of the string
    THEN Send the next character to printer
    ELSE /* Need to start on a new string */
    Return previous buffer to its Partition;
    /* When old string no longer needed */
    Get Message from the Queue;
    /* ISRs should never wait */
    Get new string pointer from message;
    Get character count from message;
    Send first character to printer;
    ENDIF
END
```

This driver design pretty much follows the pattern set by the previous driver designs. It seems like it ought to work pretty well, just like the previous designs will work pretty well if you use them in appropriate situations. But this one will fail miserably.

It's got a built-in assumption which is probably not going to always be true on your embedded system. The assumption is that the Message Queue which brings new buffers to the ISR, always has a message in it. In other words, it's assuming that there's always something new which needs to be printed.
What will go wrong if there's nothing new that needs to be printed?? Well, after printing the last character that needs to be printed, the ISR will try to get the next message from the queue. But at that time the queue will be empty, since there's no "next message" waiting. So the ISR will exit without sending a new character to hardware. And so the hardware won't deliver another interrupt. [On output devices, an interrupt usually means "I'm done doing the previous output and I'm ready for a new one."] And so the interrupt service routine will never get to run again.

Even if new messages get queued up for printing, the ISR won't run again. And so the new messages will not get handled by the ISR. And so printing will never get started again, if we use a driver structured in this way.

But don't cross out this part of the paper quite yet. We'll put a small change into this driver design, and get a pretty similar driver design which does work.
We can revive the Output Data Spooler driver design just described, and make it work properly, by breaking the ISR apart into two pieces.

The first piece of the ISR is the part previously described as "Calm down the hardware device". This is all sorts of device hardware-specific activity which needs to be done upon each interrupt occurrence to make sure that the device is working properly and will work properly on the next output operation.

The second piece of the ISR is all of the remaining ISR logic. It relates to sending out a character to hardware, and also to making sure that the proper characters are being readied for subsequent output. This second piece might be called the "Character Handler".

Normally while characters are being printed, the first piece of the ISR can simply call the second piece every time it runs. This was the driver design we studied in the previous section. The "Calm Down" part of the ISR calls the "Character Handler" each time it runs. But that design ran into trouble. And the trouble was that there was no way to run just the "Character Handler" if an interrupt didn't arrive to run the "Calm Down" part of the ISR first.

Breaking the ISR's logic into two pieces can help, because it will allow us to call the "Character Handler" from "Calm Down" when interrupts are coming in. And it will allow us to call the "Character Handler" in some other way when interrupts are not coming in.

Designers refer to calling the "Character Handler" in these other ways as "Priming the Pump", since as soon as "Character Handler" is called when there's a message queued up to be printed, the "Character Handler" will send the first character to the printer and (as if by magic) the hardware will come alive and respond with an interrupt (to announce that it finished printing that first character). Once that first new interrupt arrives, the ISR begins to run in normal fashion again, with "Calm Down"s calling the "Character Handler" exactly as originally designed.
But a question remains: How does another chunk of software know whether or not it needs to call the "Character Handler"? And the answer is... that the "Character Handler" can detect when no more interrupts will be arriving and the driver is about to "die". It detects this indirectly, by detecting that the Message Queue which feeds it character strings for printing is empty at a time when a new character is needed to continue printing.

If "Character Handler" is usually run as part of an ISR, it can not wait for messages on this Queue. So it's got to do something else. One thing it can do is to put a semaphore token into a Semaphore set up especially for this purpose. This is a signal to any other interested chunk of software, that no interrupts are expected. And so this chunk of software needs to call the "Character Handler" directly in order to "Prime the Pump".

An example of such a driver architecture is shown in the following diagram. It's an output data spooler, where the entry area of the driver may sometimes need to call the "Character Handler" in order to "Prime the Pump".
The entry area of the driver does the following when called by a task:

DevOutSpoolBetter_entry:
BEGIN
    Request new buffer from Partition;
    Fill buffer with string for printing;
    Put buffer pointer into a message;
    Put character count into this message;
    Put this message into Queue;
    IF semaphore is set
    THEN call 'Character Handler' directly
    ENDIF
END

The ISR has this very simple logic:

DevOutSpoolBetter_ISR:
BEGIN
    Calm down the hardware device;
    Call 'Character Handler'
END

And the 'Character Handler' itself looks like:

Character_Handler:
BEGIN
    IF in the middle of the string
    THEN Send the next character to printer
    ELSE /* Need to start on a new string */
        Return previous buffer to its
    ENDIF
    /* When old string no longer needed
    */
    Get Message from the Queue;
    /* ISRs should never wait */
    IF there is no message queued
    THEN set the semaphore
        /* Interrupts may soon stall
    */
    ELSE
        Get new string pointer from message;
        Get character count from message;
        Send first character to printer;
    ENDIF
END
This driver will recover from situations where there's nothing to print for a while. The semaphore which has been added here gives the signal to "Prime the Pump".

CONCLUSION

This has been just a short introduction to the world of device driver architecture. Depending on the nature of your hardware and your I/O requirements, things can get more complex in the architecture of both synchronous and asynchronous device drivers.

===
end
===