D. Kalinsky Associates

Advanced Technical Paper:

"Architecture of Device I/O Drivers"

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 driver software.

These days more and more embedded systems software developers take advantage of the services of a real-time operating system ("RTOS") to structure their application software. An embedded application software system 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'. In most RTOSs they are called 'tasks' and 'Interrupt Service Routines' ("ISRs"), so these are the terms we will use in this paper. An "ISR" is a concurrent "chunk" of software that executes in direct response to an interrupt, while a 'task' is a concurrent "chunk" of software that executes in direct response to a software event such as the arrival of a message. The device driver designs shown in this paper assume that you are using an RTOS.

A device driver itself is a collection of functions that are programmed to make a hardware device perform some input/output-related ("I/O") activities. A driver might contain an "initialization" function, a "read" function, a "write" function, etc. Device drivers that work with hardware devices that deliver interrupts also include the ISRs for those interrupts as an additional component of the device driver. The functions of a device driver can be called by application software tasks that would like to get some hardware I/O-related activities to happen.

MUTUAL EXCLUSION OF DEVICE ACCESS

Often, 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. For example, if you've got an application task that needs a temperature measurement in units of degrees Celsius and another task that 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 releases the semaphore's token at the end of the temperature measurement.

Usually, this can be done right inside the device driver. The driver function 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 function releases the semaphore's token.

For devices that may be thought of as "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 function can print 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 that 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 structure will be quite different for each of these alternatives.

SYNCHRONOUS I/O DRIVERS

In a synchronous driver, the application task that 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 that actually called the driver function 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 and become blocked.

This is achieved straightforwardly, since each driver function is essentially a subroutine of the requesting task. So, for example, 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 function attempts to get a semaphore token that is not present, the driver function would be blocked from continuing execution; and together with it, the requesting task would be put into a blocked (or "waiting") state.

We can see this in Figure_1 below. The requesting task is shown on the left as a light blue rectangle with rounded corners. It calls the driver's "read" function, shown as a yellow rectangle (named "**`DevRead()`**"). This call can be either through the RTOS, or bypassing the RTOS (shown as a black-dotted rectangle). The driver's "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. Since the semaphore initially has no tokens, the driver "read" function, and hence the task to its left, will become blocked To the right of the semaphore, is a pink ellipse representing an ISR. The "lightning" symbol represents the hardware interrupt that triggers execution of the ISR. When device hardware completes the requested "read" operation, it will deliver an interrupt that triggers this ISR, that will put a token into the semaphore. This is the semaphore token for which the entire left side of the diagram is waiting, so the left side of the diagram will then become un-blocked and will proceed to fetch the newly-read data from the hardware.

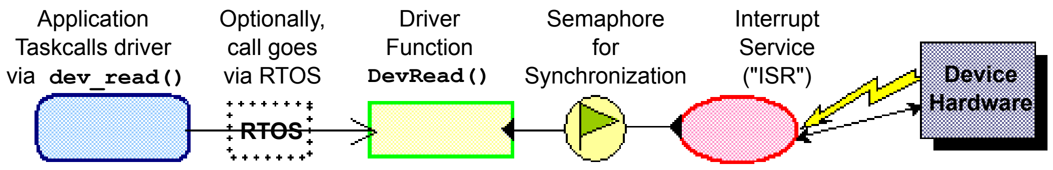
Application Taskcalls driver via dev_read()
Optionally, call goes via RTOS
RTOS
Driver Function DevRead()
Semaphore for Synchronization
Interrupt Service ("ISR")
Device Hardware

Figure 1: Basic Synchronous Driver for an Interrupting Input Device

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 responsibilities are to handle the immediate needs of the device hardware, and then 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 all the software on the left from waiting in the blocked state; and when it resumes executing it can take the final results of the hardware I/O operation and begin processing them.

The driver function (shown as the yellow rectangle near the center of Figure_1), does the logic described in the following pseudocode when called by a task:

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

The ISR has very simple logic:

```
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 that 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. A task can continue to execute at the same time that hardware is executing the I/O operation that the task requested.

Asynchronous drivers are more complex in their design than other drivers. In some cases, an asynchronous driver might be unnecessary "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?"*** Occasionally the answer may be ***"No, I need the I/O completed before my task can usefully do anything else."***. Such an answer says that an asynchronous driver is overkill: In this case, a synchronous driver would do just fine (see Figure_1).

For example, say you're designing a driver for an input device. What can a task do with that input before it's ready?? Most often, 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 Figure_2 shows what the design of such a driver would look like:

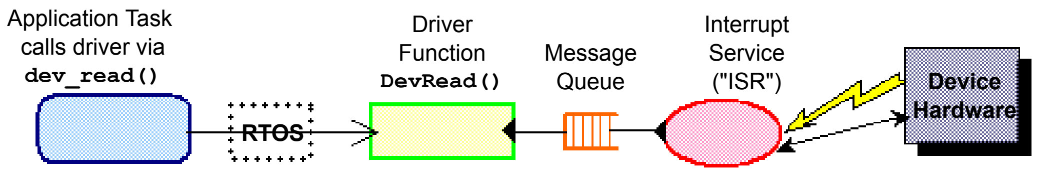
Application Task calls driver via dev_read()
RTOS
Driver Function DevRead()
Message Queue
Interrupt Service ("ISR")
Device Hardware

Figure 2: Basic Aynchronous Driver for an Interrupting Input Device

The new ladder-like symbol appearing here represents a message queue. It's a place to store information about one or more previous inputs. The driver's "read" function 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.
task. And the ISR will put new input into this queue whenever it gets one.

If device hardware will create new inputs only when requested to do so by the driver's "read" function, then a queue of maximum length 1 message is sufficient. If, on the other hand, device hardware is "***free-running***" so that it can create new inputs even when not requested to do so by explicit software request, then the queue should be assigned a length sufficient to hold rapid bursts of inputs.

The driver's "read" function does the logic described in the following pseudocode when called by a task:

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

The ISR has this logic:

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

In order for this to work, a driver initialization function needs to create the Message Queue that is at the heart of this driver.

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 might in some cases not be what is desired. Sometimes what is desired is the latest input only. The problem with the design in Figure_2 is that old inputs would queue up in the Message Queue. Requesting tasks could be fed very old inputs, while newer inputs would languish in the message queue.

So for a free-running input device, a latest-input-only driver architecture could be that shown in Figure_3 below.

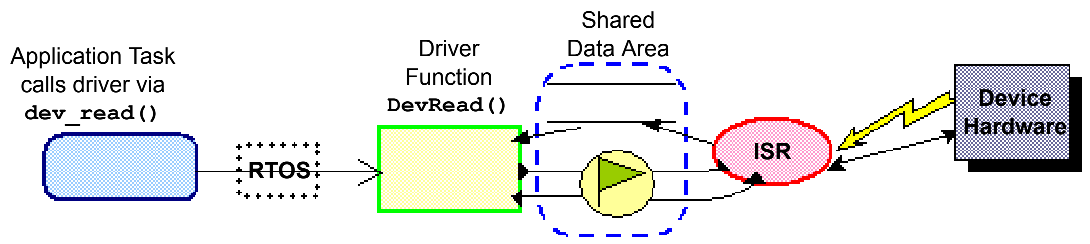

Figure 3: Aynchronous Driver for Latest Input Only

This design shows a protected shared data area at its center. The shared data area, 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 "read" function software reads it from the shared data area, after obtaining access permission from the associated semaphore. Since old input data are overwritten by the ISR, the data being read and fed to the requesting task are always the "freshest" data available.

The driver "read" function does the following when called by a task:

```
DevReadLatest_function:
   BEGIN
           Get Shared Data Access Semaphore (Waiting OK);
                   /* Wait for Semaphore Token */
           Read latest input from Shared Data area;
           Release 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 Information from Hardware;
                   Write latest input data to Shared Data area;
                   Release 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, the ISR 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

Often a hardware input device can deliver large amounts of input data freely to a computer without its being explicitly requested to do so by software. In the next design model, would like to capture and process all of the incoming information, without losing any -- even if it is arriving in irregular large bursts. An example of this is the arrival of data packets from a communications network.

Another example is the arrival of character strings from a serial line. Every character arriving via serial line is a byte of incoming data, announced to the CPU by a separate 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 that 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 Pools service that can manage large numbers of RAM memory buffers of standard sizes. The ISR part of the driver could "borrow" a buffer from a Pool 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 (left half of the diagram). We see this pictured in Figure_4 below.

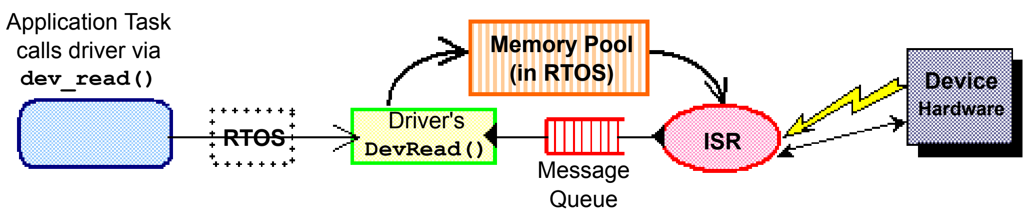

Figure 4: Serial Input Data Spooler

The driver's "read" function does the following when called by a task:

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

The ISR has this logic:

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

In some instances, this ISR will be unable to obtain the memory buffer it needs from the Pool, 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.

OUTPUT DATA SPOOLER

For many output devices, asynchronous drivers have clear advantages over synchronous drivers. While the asynchronous driver "write" function 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 in Figure_5 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 Pool.

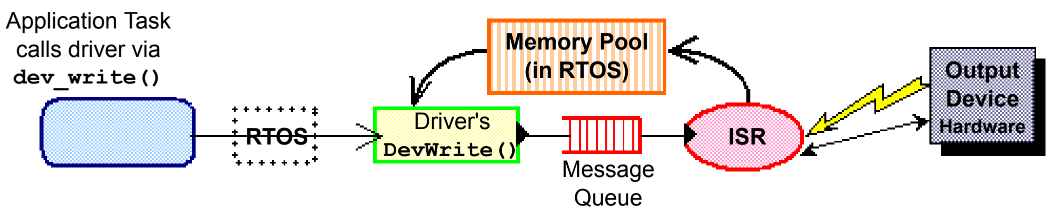

Figure 5: Output Data Spooler

(Note: See CAUTION !! below before using this design.)

The function "**`DevWrite()`**" part of the driver does the following when called by a task:

```
DevWriteSpool_function:
       BEGIN
               Request new buffer from Memory Pool;
               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:

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

CAUTION !!

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**.

The only positive thing that can be said of this driver design, is that it will ***continue*** working once it's been successfully started, as long as its message queue continues to contain output (printing) requests.

But it's got a built-in assumption that is probably not going to always remain true in your embedded system. The assumption is that the Message Queue that brings new buffers to the ISR, will always have at least one message in it. In other words, it will continue to work assuming that there's always something new that 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. Remember, an ISR is not permitted to wait for a message (or for anything else, for that matter). And so the hardware won't deliver another interrupt, since it hasn't been asked to do anything new. [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 later 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 that's structured in the way shown in Figure_5.

Please note that this driver design will also run into a similar problem when the driver tries to begin running for the first time. In other words, it will be ***"dead in the water"*** the first time it tries to do some output, and it will never succeed in getting started with its first output to the hardware device.

But don't cross out this part of the paper quite yet. We'll put a small change into this driver design model, and get a pretty similar driver design that does work properly.

"PRIMING THE PUMP" IN AN OUTPUT DATA SPOOLER

We can revive the failed 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 that needs to be done when each interrupt arrives, to make sure that the device is working properly and will work properly on the next I/O operation. Let's now call it "**Handle Output Done**".

The second piece of the ISR is all of the remaining ISR logic. Its job is to send out the next character to hardware, and also to make sure that the proper characters are being readied for subsequent output. This second piece will be called "**Set Up Next Output**".

See Figure_6 below, showing "Handle Output Done" and "Set Up Next Output" after they've been separated..

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 "Handle Output Done" part of the ISR calls "Set Up Next Output" each time it runs. But that design ran into trouble. And the trouble was that there was no way to run just "Set Up Next Output" if an interrupt didn't arrive to run the "Handle Output Done" part of the ISR first.

Breaking the ISR's logic into two pieces can help, because it will allow us to call "Set Up Next Output" from "Handle Output Done" when interrupts are coming in. And it will allow us to call "Set Up Next Output" in some other way when interrupts are not coming in.

Designers refer to calling "Set Up Next Output" in these other ways as "**Priming the Pump**". When interrupts are "dead" and "Set Up Next Output" is called by non-ISR software, it checks if there's a message queued up to be printed. Then "Set Up Next Output" 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 "Handle Output Done"s calling "Set Up Next Output" exactly as originally designed.

But a question remains: How does a chunk of non-ISR software, like the driver's "`write`" function, know whether or not it needs to call "Set Up Next Output"? The answer is that "Set Up Next Output" 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 that feeds it character strings for printing is empty at a time when a new character is needed to continue printing. Since "Set Up Next Output" is usually run as part of an ISR, it cannot wait for messages on this Queue. So it's got to do something else. One thing it can do is to put a 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 "Set Up Next Output" directly in order to "**Prime the Pump**".

An example of such a driver architecture is shown in Figure_6. It's a repaired output data spooler, where the "**write**" function of the driver may sometimes need to call "Set Up Next Output" directly in order to "**Prime the Pump**".

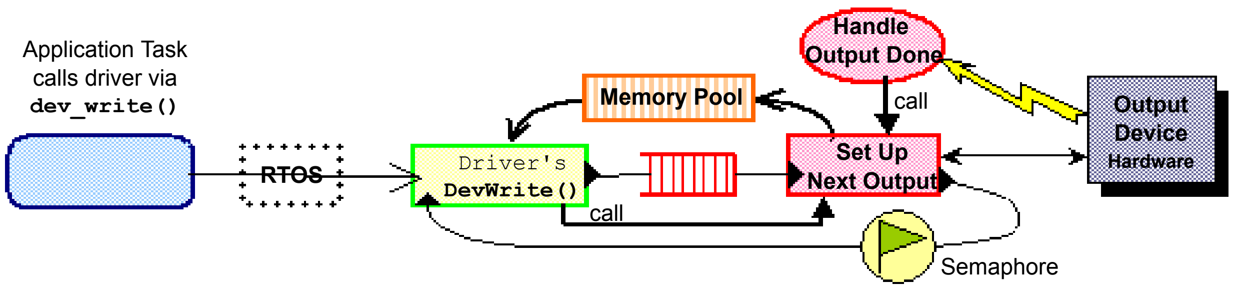

Figure 6: Correctly Operating Output Data Spooler

The "**write**" function of the driver does the following when called by a task:

```
DevWriteSpoolBetter_function:
BEGIN
        Request new buffer from Memory Pool;
        Fill buffer with string for printing;
        Put buffer pointer into a message;
        Put character count into this message;
        Put this message into Queue;
        IF Get 'Device Stalled' Semaphore (Without Waiting);
          THEN     /* Interrupts stalled */
               Call 'Set Up Next Output' directly
        ENDIF
END
```

The ISR "**Handle Output Done**" has this very simple logic:

```
DevHandleOutputDone_ISR:
BEGIN
        Calm down the hardware device;
        Call 'Set Up Next Output'
END
```

And "**Set Up Next Output**" itself looks like:

```
Set_Up_Next_Output:
BEGIN
        IF in the middle of a string
          THEN Send the next character to printer
          ELSE /* Need to start on a new string */
                Return previous buffer to its Memory Pool;
                  /* When old string no longer needed */
                Get new Message from the Queue (Without Waiting);
                IF there is no message queued
                   THEN Set 'Device Stalled' Semaphore to 1
                           /* Interrupts stalled */
                   ELSE
                        Get new string pointer from message;
                        Get character count from message;
                        Send first character to printer;
                ENDIF
        ENDIF
END
```

This driver will recover from situations where there's nothing to print for a while. The semaphore named 'Device Stalled' that 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.