Some Driver and Device Support Issues

T.S.

ISR vs. Task Driven

- ISR driven:
  - does work in ISR
  - many ported drivers are written this way
  - inadequate for real-time OS
- Task driven
  - ISR schedules a driver task
  - task does all real work
  - adequate for real-time OS

Rationale

- real-time OS must be deterministic, minimal latencies
- any work in ISR adds to latencies
- driver shouldn't set policy (priority) but leave this to application
- any ISR driven device can preempt a more important task but no task can preempt ISR work -> *bad*
- less resources available to ISR than task

Rationale (cont.)

- reassigning task priorities is simple
- reviewing and rewriting drivers is hard

ISR driven

- Typical driver ISR:

```
general_driver_isr()
{
   /* clear_edge_interrupt() here! */
   do_io_and_other_work();
   clear_level_interrupt();
}
```

- Note: Must clear edge-sensitive IRQ *before* servicing to prevent lost interrupts (2nd IRQ between service + clear)
- clear level-sensitive IRQ after servicing to avoid leaving pending
- details (pay attention; hard to debug!) depend on particular device

Task Driven

- ```
  tsk_driven_ISR()
  {
     mask_interrupt_at_device();
     semaphore_release();
  }
  ```
- ```
  driver_task()
  {
    while (1) {
      semaphore_obtain();
      generic_driver_isr();
      unmask_interrupt_at_device();
  }
  ```

Exceptions

- Only exception: work is so trivial that it doesn't justify a task-context switch (< 1us).

EPICS Device Support

- Be aware that devsup read/write 'methods' are executed from *shared task contexts*:
  - scanners (scan Once, periodic scanners)
  - 3 callback tasks (event, I/O intr, all callback work)
- -> Anything you do, might delay other work.

What to Avoid

- blocking operations:
  - epicsMutex
  - epicsEventWait
  - printf, read, write
  - ...
- slow and non-deterministic operations
  - malloc
  - polled wait
  - ...

What to Do

- asynchronous record processing
- create your own tasks (appropriate priority) to do work

Other Considerations

- *Never* do implicit I/O (but use in_le32() etc.)
  [more readable, guarantees in-order *execution* of I/O]
- *Always* declare `volatile`:
  - memory-mapped device registers
  - global variables that can be changed from ISRs or other tasks.

  [guarantees in-order *compilation*]

Examples

- ```
  extern volatile int stop;

  do_something()
  { /* 'stop' is set by other task */
    while ( !stop )
      out_le32(addr, val);
  }
  ```
  ```

Examples

- ```
  extern volatile int stop;

  do_something()
  { /* 'stop' is set by other task */
    while ( !stop )
      out_le32(addr, val);
  }
  ```
- Without '`volatile`' the compiler may generate instead:

  ```
  do_something()
  {
    if ( !stop ) {
      while (1) out_le32(addr,val);
    }
  }
  ```
  ```

Example 2

- ```
  extern volatile int cnt;

  do_count()
  { unsigned x;
    rtems_interrupt_disable(x);
    cnt++;
    rtems_interrupt_enable(x);
  }
  ```
  ```

Example 2

- ```
  extern volatile int cnt;

  do_count()
  { unsigned x;
    rtems_interrupt_disable(x);
    cnt++;
    rtems_interrupt_enable(x);
  }
  ```
- Without `'volatile'` the compiler may generate:

  ```
  do_count()
  { unsigned x;
    rtems_interrupt_disable(x);
    rtems_interrupt_enable(x);
    cnt++;
  }
  ```
  ```

Summary

- Favor task driven approach to ISR when writing drivers.
- Keep EPICS callback threads responsive. Let devsup read/write methods use asynchronous processing and your own threads.
- Pay attention to variables with side-effects. Problem becomes more apparent with more modern compilers and CPUs (optimizations abandon sequential execution, faster execution).