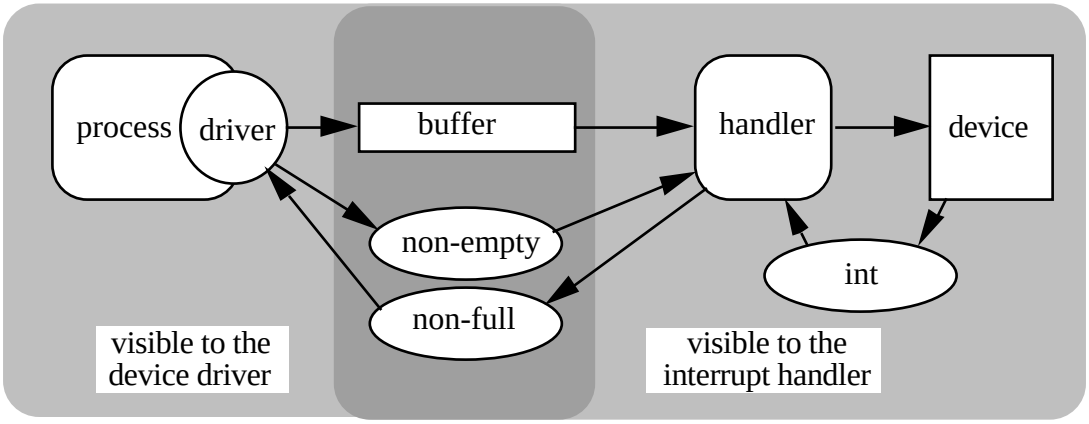
DEVICE CONTROL SOFTWARE

The programme (written in Modula 2) and diagram below illustrate the relationship between different parts of the operation of moving data from a process to a device. In terms of the chapter *MAKING DEVICES WORK*, this is the interrupt handler and part of the device driver, though for the sake of simplicity it's a rather less elaborate design as might be appropriate for a rather simpler system. We've copied it from somewhere, and we would gladly acknowledge the source if we could remember where it was. Any relevant information would be welcome.

The module accepts single characters one by one, buffers them as required, and sends them to a single-character device. It does not handle an explicit input-output request queue, but the buffer queue is equivalent. Here's a picture :



HEADER.

The software is presented as a **device module.** Only the name of the character output procedure, which you call to move a character to the output device, is visible from outside the module; the remainder of the software is private.

We've added the comments throughout the programme. (They're the bits after `--`.)

```
device module buffered_char_output[ 4 ];
define procedure driver( in ch : char );

private
```

GLOBAL DECLARATIONS.

All the variables declared here are associated with communication between the procedure **driver** and the **handler** process. The type **csr** is a rather underhand way of managing bit manipulations : **intenable** and **ready** are device status indicators found at bit positions 6 and 7 respectively of a device status byte. The **ready** indicator is not used in this part of the software. Used as a function, **csr** returns a byte with the prescribed bits turned on.

```
    const buf_size = 32;
    type
         csr = set of ( intenable at 6, ready at 7 );
         count = integer range 0 .. buf_size;
         index = integer range 1 .. buf_size;
    var
         inx, outx : index;
         n : count;
```

```
        non_full, non_empty : signal;  -- Semaphores.
        buf : array[ index ] of char;-- The character buffer.
```

THE DEVICE DRIVER.

- or part of it, anyway. In a full implementation of a system as described in *MAKING DEVICES WORK*, this procedure would be called *within* the device driver in order to print each character.

```
    procedure driver( in ch : char ); -- The output procedure visible from
outside.
    begin
        if n = buf_size
        then wait( non_full )          -- Await a signal from handler if
buffer's full.
        end if;
        buf[ inx ] := ch;              -- Put the character in the buffer.
        inx := ( inx mod buf_size ) + 1;
        n := n + 1;
        send( non_empty );             -- Something in the buffer; tell handler.
    end;
```

THE INTERRUPT HANDLER.

handler is the interrupt handler. It communicates with the rest through the global variables declared earlier, but runs independently as a separate process.

The three variables declared within **handler** are all associated with specific memory addresses by the "**at**" clauses in their declarations. (The addresses are 16-bit addresses, expressed in octal notation.) This fixed identification of device components with memory addresses is appropriate in systems – typically microprocessors – which rely on "memory mapping" to communicate with the external world. For a larger system, one would be more likely to provide the memory addresses, or some other specification of communication path to be used such as port numbers, as parameters to the installing programme.

```
    process handler;
    var
        int at #64 : signal;           -- Interrupt address for "character
received".
        status at #177564 : csr;       -- Memory-mapped device status register.
        bufreg at #177564 : char;      -- Memory-mapped device character
buffer.
    begin
        do
            if n = 0
            then wait( non_empty )     -- Wait until there's something in the
queue.
            end if;
            bufreg := buf[ outx ];     -- Put the next character in the device
buffer.
            outx := ( outx mod buf_size ) + 1;
            status := csr( intenable );    -- Permit interrupts.
            wait( int );               -- Wait until device acknowledges receipt
            status := csr( );          -- Disable interrupts.
            n := n + 1;
            send( non_full );          -- There's space in the queue.
        end do
```

```
end;
```

SETTING UP THE SYSTEM.

This code is the "body" of the module. It is executed when the module is started; it sets sensible initial values for variables which need them, and establishes **handler** as a separate process.

```
begin                                   -- Set up the initial state.
    n := 0;
    inx := 1;
    outx := 1;
    init handler                        -- Set up handler as a separate process.
end module buffered_char_output;
```
