Flags, Interrupts and Other Devices

It is apparent from several of our previous discussions that a major problem for any computer system is synchronising with external events, be they real events, completion of operations or arrival of information. All these events share one thing in common - they are asynchronous to the operation of the computer. In order to allow the computer to synchronise with these things some form of "flag" must be used to inform the system software that the event has occurred (eg an input device has raised the DR line) and along with that it is required that the system must have the programming built in to respond to that flag.

Responses to flags are generally handled by one of three mechanisms:

- wait for flag

- interrupts

- DMA (Direct Memory Access)

Wait for Flag

In the simplest form the wait for flag consists of a program fragment which simply reads the flag and waits for the event. In PASCAL this could be (depending upon implementation):

```
while port[IN] == 0 do;
```

which simply hangs around until the location IN becomes non-zero. Most languages have this form of construction and in C it could be:

```
while ( *mem(IN) == 0 );
```

and in ASSEMBLER

```
LOOP
        LDA    $IN
        BEQ    LOOP
```

However this is not the time to consider all the various forms. All forms have one thing in common and that is that the computer can do absolutely nothing else except wait for that flag and if it doesn't come until tomorrow then we have just lost 24 hours available computer time.

We can recover some of that time under some circumstances. Every event has a "latency time" which is the time after the flag going up by which it MUST be serviced. Consider the case of data coming off a disk. The latency time for a word here is very short because the next one is right behind it. However if we consider the "complete" flag for an a/d converter then the latency time may well be infinite as the converter will hang around for ever until we get round to it. In this case we can allow the program to occasionally check the flag. In general the computer must check the flag more often than the maximum allowable latency time[12].

Thus we get:

```
repeat
    if port[IN] <> 0 then service_flag();
    go_do_something_else();
until FALSE;
```

In fact we can do a whole series of flag checks at one time so long as the worst case latency time is not exceeded. In C this looks like

```
FOREVER {
        if ( *mem(flag1) != 0 ) process1();
        if ( *mem(flag2) != 0 ) process2();
        if ( *mem(flag3) != 0 ) process3();
        ..........
        ..........
        ..........
        ..........
        }
```

which allows us to keep up with a large number of things simultaneously. However in a large system latency becomes a real problem and it becomes necessary to think of other ways of doing things. However before we go the obvious route, let us consider some semi-hardware solutions to some problems of this nature. The reason for doing this is that interrupt and

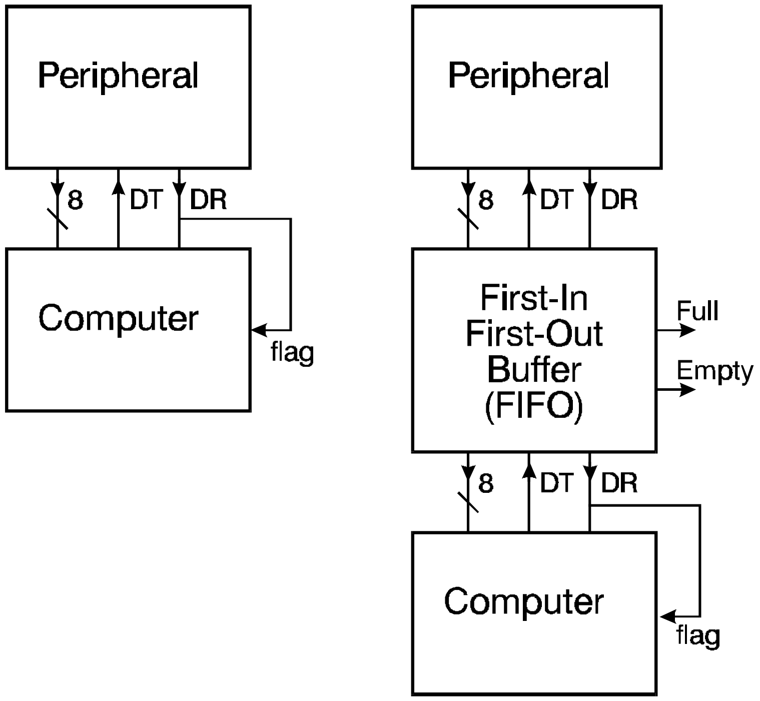
[12] This, of course, can be a remarkably difficult thing to calculate - which is why it is often determined by experiment!

DMA programming are inherently harder to manufacture and debug. "Skip-on-no-flag" is much easier.

Increasing Latency Time

Central to the problem is a short latency time and the solution can often be as simple as increasing the latency time of a single peripheral. This can be accomplished by a number of techniques but the simplest is the "management" technique of servicing infrequently but giving it a lot to do when you do.

The device that allows this is the First-In, First-Out (FIFO) buffer which is a data silo with an IN and OUT end. Data goes in the IN and comes out the OUT end in the same order but the storage inside is elastic to some depth, e.g. 64 words. Thus 64 words can be put in before any have to be taken out.



Adding a "First-In First-Out (FIFO) Buffer

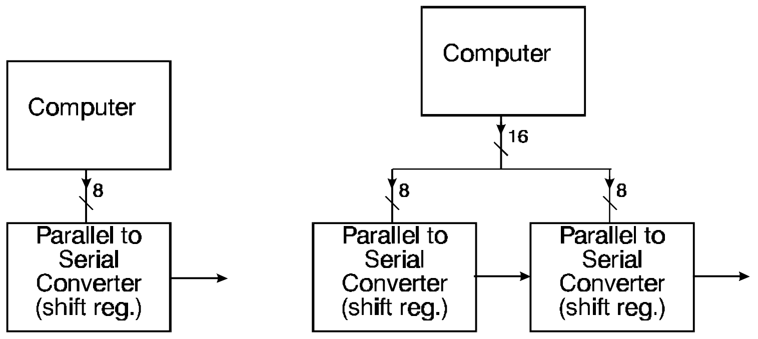
This implies that you can ignore the FIFO until it is nearly full so long as you empty it sufficiently when you do access it. If we make the simple assumption that the FIFO is completely emptied by the service routine then the latency time is increased by a factor equal to the depth of the FIFO. Since FIFOs can have "depths" ranging from 4 to effectively infinity, this simple transformation can increase the latency by a large factor.

Most FIFOs have control lines for handshaking, full, empty and some even generate 1/4, 1/2 and 3/4-full flags for intermittent data where you really only need to service the thing very occasionally.

The programming for such a system then appears as:

```
repeat
    while port[IN] <> 0 do service_flag();
    go_do_something_else();
until FALSE;
```

Here is an even simpler solution which occurs in parallel/serial conversion. Here a parallel data word is clocked out one bit at a time onto a single line. Thus for an n-bit word the computer has to supply a byte every *n* output clock cycles. Normally this is handled a byte at a time but in order to increase the latency time a longer shift register can be used which doubles the latency time immediately - it can even make the programming simpler in some instances.


Computer
8
Parallel to Serial Converter (shift reg.)
Computer
16
8
8
Parallel to Serial Converter (shift reg.)
Parallel to Serial Converter (shift reg.)

Serial-to Parallel Conversion to Increase Latency Time

In summary then the real reason for skip-on-flag is it's simplicity. It is inefficient in the use of computer resource because time is wasted in short loops which do nothing. However many programs run entirely by skip-on-flag.

Interrupts

Interrupts are a hardware device (i.e. a piece of wire going to the CPU) for getting the computer's attention. Before an interrupt can be made the following things must be present:

- The device must be capable of generating interrupts
- The hardware must be present to wire the interrupt to the processor
- The device interrupt generation system must be enabled
- The processor interrupt recognition must be enabled
- The processor programming must contain code to deal with interrupts
- The device must be known to that code

- The device must actually generate the interrupt.[13]

Not all devices are capable of generating interrupts but most of the "peripheral" chips are, including the Peripheral Interface Adapter (PIA) discussed in the Appendix. Some of the bits in the PIA control register are dedicated to "enabling" and "disabling" the interrupts. This must be possible because you may not want a device to be always interrupting, like a fretful child, if you have nothing to give it to do.

The action of a simple interrupt on the processor is to cause a specific sequence of events something like:

- The current instruction is completed.
- Enough of the internal processor state is saved to enable processing to be resumed later
- A specific action is taken which results in processing jumping to a routine in the memory which will "service" the interrupt.
- Interrupt processing continues until a corresponding "return from interrupt" instruction is received
- The "save" is reversed and processing continues

Note The interrupt processing must remove the cause of the interrupt or the above sequence will loop indefinitely.

It is usual in simple interrupt processing to disable the interrupts at the computer end during interrupt "servicing" to prevent recursion. Because of the machine specific nature of interrupts, high-level support is a bit difficult. However in several systems code fragments have been written which cause a C function such as "irq" to be executed when interrupts are enabled and received. Thus code of this form is possible:

```
interrupt irq {
if ( *mem(flag1) != 0 ) process1();
if ( *mem(flag2) != 0 ) process2();
..............
.............
.............
}
```

[13] This may seem like a pretty obvious and trivial list. However many problems brought ot my attention have boiled down to a failure of an item of this list - frequently including the wires!

Note that the routine must use a "return from interrupt" instruction to finish up the routine instead of the more usual "return from subroutine". This is handled in this pseudo-implementation by the "interrupt" directive at the top of the routine which tells the compiler to generate the "save" code and the "return" code. You will notice that the body of the routine actually looks very similar to the "skip-on-flag" routines above - and it is. In fact if all the machine has to do is to run around the flags then the interrupt facility is useless. However we can now use any free time the processor has to run another program such as a PASCAL system.

There is still a problem here because of latency. It is still necessary to get round to things before they become critical and if there is suddenly a lot to do, the important things may get neglected. For example a high-speed device may interrupt again before the maximum servicing time has elapsed. This can be circumvented somewhat by looking at the short latency flags first and only servicing one interrupt at a time.

```
interupt irq {
if ( *mem(flag1) != 0 ) process1();
else if ......................
else if .......................
.........................
.......................
}
```

However one can very rapidly get into a knot here because it does take time to service interrupts, i.e save machine state, find out where to go, go there, then return machine state after the interrupt and get on with things.

Some amelioration of this condition can be achieved by the use of "vectored interrupts". The philosophy is to allow the machine to quickly determine where the required interrupt service routine is, without checking a lot of flags. There are essentially two methods of doing this:

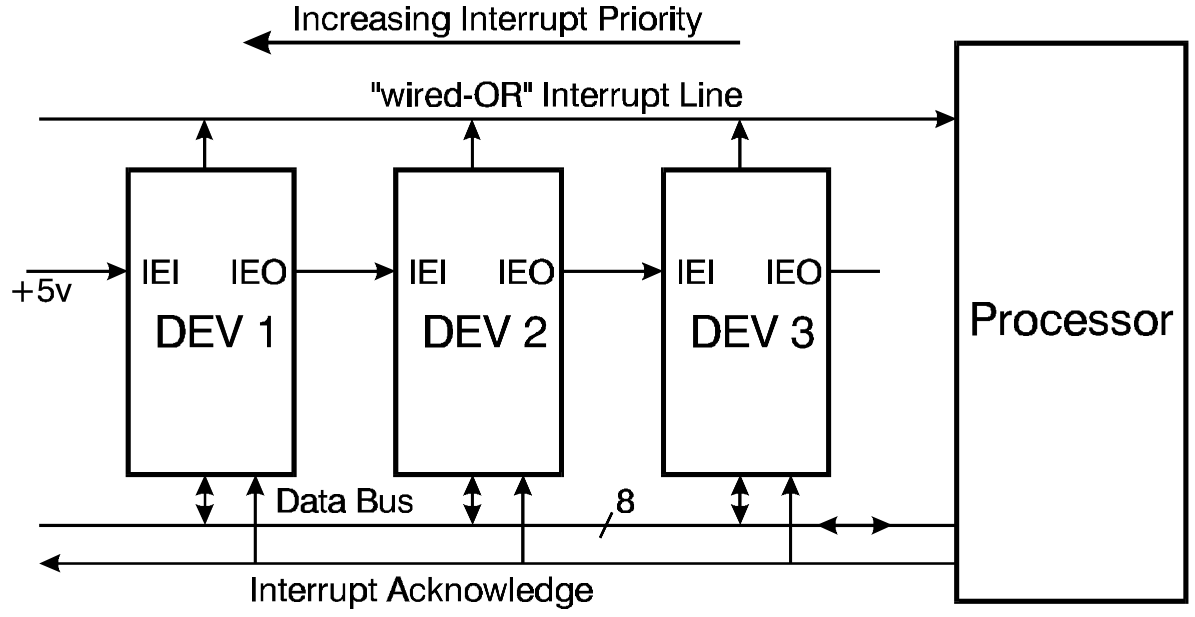
1) Use a number of interrupt lines each of which points at a particular table location in the memory

2) Allow the peripheral device to supply the address in a special "interrupt acknowledge" cycle.

In the first case the system is limited to a few interrupts which may be "prioriterised" (if there is such a word) but the program can rapidly make changes in the handler address. The IBM PC uses this scheme with about 8 allowed interrupts. If you have more than 8 devices then you may multiplex several onto one interrupt line and then do a "skip-on-flag" choice between them at the start of the handler[14]. This sounds regressive but with 8 devices on each of 8 interrupts the correct handler can be found in an average of 4 checks compared with 32 checks for 64 devices multiplexed onto one interrupt line.

The second scheme requires more sophisticated peripherals but can cope with an (apparently) infinite number of devices without increasing the hardware complexity. Since this scheme lacks an inherent priority scheme one has to be enforced. An example of such a priority structure is a "daisy-chain" which is discussed below.

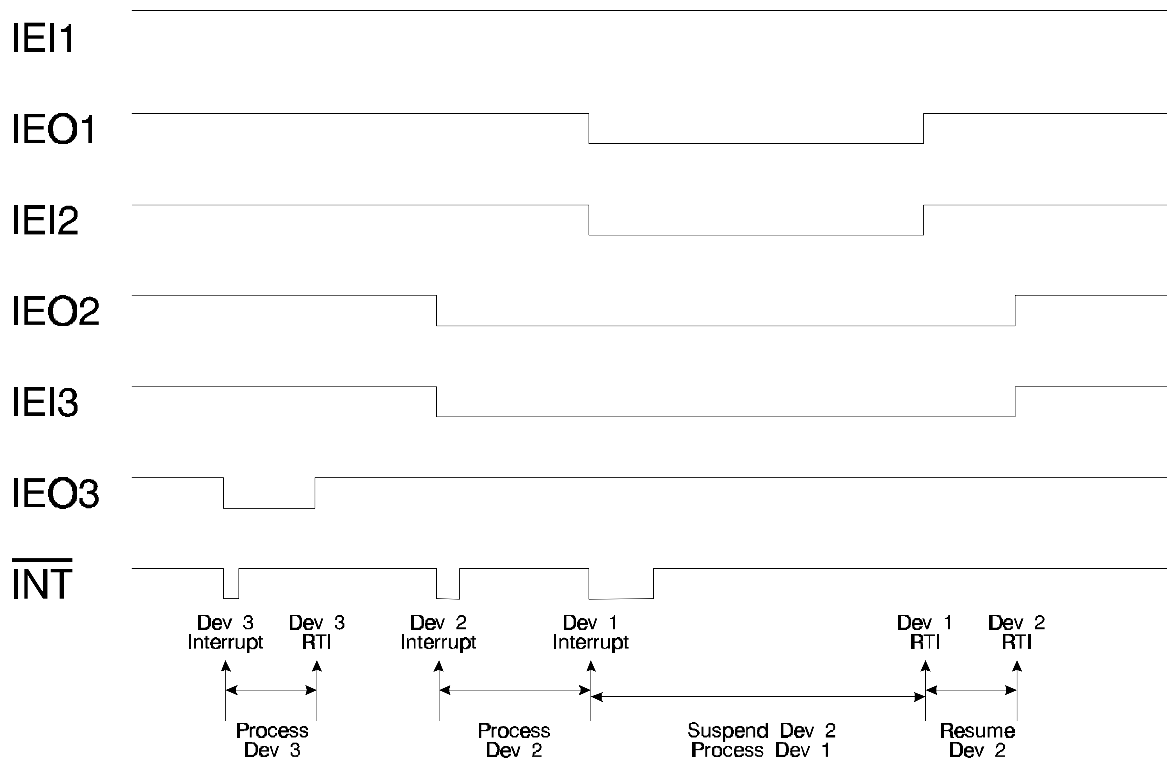
This scheme is implemented rather nicely in the Z80 family of peripheral chips (with Z80 processors only of course). Each chip is given the address of its interrupt service routine and supplies that to the processor. In order to make sure that high priority devices receive attention first, the devices are daisy-chained:



Z80 Vectored, Priority-Structured Interrupts

[14] At least you should be able to. However on an IBM PC bus it is remarkably difficult to do this in the general case owing to the use of tri-state buffers on the interrupt line rather than wired-OR buffers.

Initially all lines are de-asserted. The interrupt line is a "wired-OR"- active low. An idle device will transfer the state of IEI to IEO and an interupting device will hold IEO low.



Z80 Interrupts - Timing

To interrupt the processor a device asserts INT and lowers IEO unless IEI is low in which case it must wait. Thus devices higher in the chain get priority. If during interrupt servicing IEI goes low, then somebody higher in the chain is requesting service and we must wait until they have finished at which time they will raise their IEO which will raise our IEI and we can resume. Thus a possible sequence could be:-

- Immediate interrupt for device 3 (/INT↓[15],IEO3↓) commence service
- Device 3 RTI (/INT↑,IEO3↑)
- Device 2 interrupts (/INT↓,IEO2↓), commence interrupt service routine 2
- Device 1 interrupts (IEO1↓), suspend device 2, commence service routine 1
- Device 1 RTI (IEO1↑), resume device 2 interrupt processing
- Device 2 RTI (IEO2↑), allow interrupts from lower priorities

Notice that in order to get the vectored interrupt going it is necessary for the interrupt processor for each device to contain code to make the device deassert the INT state as soon as possible in the interrupt service routine. Failure to do this will "block" further interrupts of any priority.

Not shown in the timing diagrams is the fact that this also implements a vectored scheme. Every time the INT state is asserted, the processor runs an interrupt acknowledge cycle which interrogates the interrupting device for a vector address[16]. This address is held in a device register which can be loaded from the processor (at an earlier time) or changed at any time.

Direct Memory Access

Direct memory access is a system whereby two processors share the same bus and one is dedicated to the simple task of dealing with the device flags and only informs the other on completion of some specific act (such as the transfer of all the information requested). The master processor informs the slave where to put the information in memory (or where to find it) how much data to move etc. etc. and then the slave delivers it on demand. The slave uses the memory by interleaving with the master either invisibly by bus ingenuity, or visibly by demanding the bus and being granted it for a brief period. This acccess is usually by means of two lines connecting the master processor with the direct memory access unit. These lines are often called Bus REQuest (DMA→processor) and Bus GRaNT (processor→DMA). The method of operation of these lines is evident from the naming: The DMA uses BREQ to request the bus. At a convenient time, usually at the end of the current instruction, the

[15] I'm using /INT here to denote the complement (active low) form of an INT line. This is because my word processor won't put an overbar over the whole text without stress.

[16] Actually I've faked things a bit here to make it clearer. Actually it runs an M1 cycle with the IORQ line asserted (low) to indicate the acknowledge cycle - but the effect is the same.

processor releases the bus and riases BGRNT. The DMA then performs one or more memory operations on the bus and then lowers BREQ at which point the processor regains control of the bus and removes BGRNT[17].

Since you can tell the slave processor to "collect 1024 readings from the a/d and tell me when you're done" the master is very free to pursue more esoteric problems.

Programming for DMA is not, in general, very difficult as most of the DMA chips are hardwired computers rather than general-purpose ones. If you want to "interrupt on completion" then you will have the same problems as standard interrupt processing.

[17] Any similarity to a 2-wire handshake is <u>not</u> co-incidental!