A 32-BIT PORT

by Martin Bela

Ever wanted to control your domestic appliances from your keyboard? Or fully automate your model railway? You have? Well, you're going to need an input and output port of some sort and this could be just what you need...

THE FIRST BIT

It's possible to control things via the printer port, but with only seven bits it's rather limited.

The 32-bit I/O port described here is designed to access upto 32 external devices each of which can be either an input or an output device.

WHAT'S WHAT?

Connector K1 is an edge connector for plugging into the expansion port of the CPC. IC1 and IC2 provide most of the address line decoding, with the outputs of IC2 (pins 15 and 13) selecting the appropriate Parallel Input/Output Controller (PIO) chip.

When IC2 pin 15 goes low, IC3 is selected, allowing data to be passed to or from I/O ports 0 and 1.

Similarly, when IC2 pin 13 goes low, IC4 (ports 2 and 3) are selected.

A BIT TOO SELECTIVE

As you can see, the circuitry around IC4 is almost the same as that around IC3; the only difference being the select lines from IC2. If you really need more I/O lines then you can duplicate the IC3 circuitry again, but this time take the select line from pin 11 of IC2. This will add two more ports (4 and 5) giving you a grand total of 48 lines (or bits).

You can of course remove the IC4 circuitry if you only want 16 lines.

THE CURRENT PROBLEM

The output pins from IC3 and IC4 cannot provide much current, and so it is intended that these outputs will be used to control relays, which in turn will switch the external devices (such as lamps, motors, solenoids etc) on and off. The circuit draws it's power from the CPC's 5 volt supply, but I would recommend using a separate Power Supply Unit (PSU) to power the relays themselves so as not to overload the CPC's supply. An interface for the relays will be needed and this will be covered next month.

I haven't shown a separate Power Supply Unit in the diagram, as you will either be quite capable of building your own or if not, then as it would need to be mains-powered, it would be safer to buy a ready-made unit.

Devices other than relays can of course be connected to the ports, such as an analogue to digital converter, for signal measurement or sound sampling for instance or perhaps infra-red LED's for control via fibre-optic cables.

For the moment though, I'll assume that you wish to use 12 Volt relays, as these are probably the easiest to understand and use.

A TERMINAL CASE OF PORT

The items labelled T00, T01, T02 etc are terminal blocks to connect the relay interface (or whatever) to.

Each PIO chip has two 8-bit ports which I will refer to as PA and PB. The actual ports that you connect to are indicated by the letter T followed by the port number, followed by the bit number; so T24 refers to bit 4 of port 2; T00 is bit 0 of port 0; and T31 is bit 1 of port number 3 etc.

WHAT'S YOUR ADDRESS?

I don't think the addresses used by this design clash with any other add-ons, but just in case, it's best to remove other expansion add-ons before plugging this board in. The addresses used are the ones recommended in the CPC manual, and are as follows:

&FAEX (selects ports 0 and 1)
&FAFX (selects ports 2 and 3)
&FBEX (selects ports 4 and 5 if used)

where X is one of the following:

&8 (i.c.'s port PA Data in/out)
&A (i.c.'s port PB Data in/out)
&C (i.c.'s port PA Control Register)
&E (i.c.'s port PB Control Register)

So, to send data in or out of port 2 (which is terminals T20 to T27) the address required is &FAF8.

Another example; address &FAEE would give you access to the Control Register of port 1 (I.C.3's port PB).

PORT PARTS LIST

Below is a parts list for the 32-bit version. For the 48-bit version add an extra R4, R5, C5, IC4, and Terminal blocks (or connectors). Delete these parts if you only want a 16-bit port.

ITEM	DESCRIPTION	VALUE
R1-6	0.25W 5% resistor	1k0
C1,2	Ceramic capacitor	100nF
C3	Electrolytic capacitor	100uF
C4,5	Ceramic capacitor	22nF
IC1	13 input NAND gate	74LS133
IC2	3 to 8 line Decoder	74LS138
IC3,4	4.5MHz Z80A PIO	Z8420APS
K1	50 way edge connector	
T00-T37	Terminal blocks or connectors of your choice	

The capacitors should be rated at 10 Volts or greater and the tolerance isn't too important; 20 percent or better is quite adequate. C1,2,4, and 5 should be mounted as close to the i.c. legs as possible.

The connector K1 can be an IDC type which is attached to a length of 50 way cable (like that used on romboxes, multifaces etc) in which case the I/O board end will need another plug and socket or, alternatively, you could use an edge connector which is soldered directly to the I/O board.

If you decide on the cable, then make it as short as possible, no more than say about 300mm or it could give you some problems. The pin numbers shown are those of the CPC's expansion port, as shown in the CPC manual.

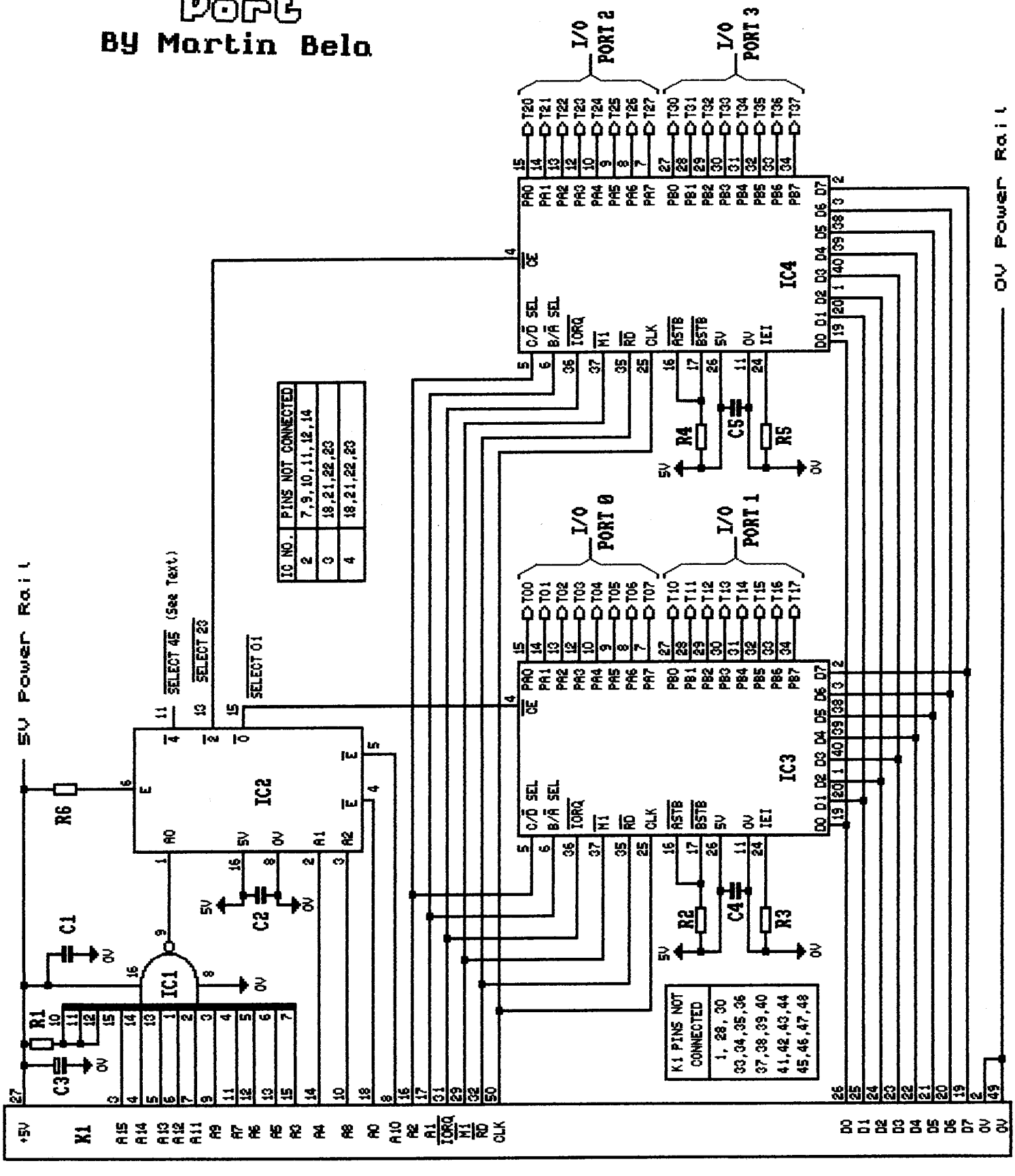
THE NEXT BIT

Next month we'll have a go at programming the ports, and find-out how to connect relays, switches, and some LED's to it.

Happy Soldering....TTFN...

32-Bit CPC InPut/OutPut Port

By Martin Bela



32 BIT IO-PORT PART TWO

MARTIN BELA

If you've built the I/O Port from last month you'll no-doubt be eager to try it out, so without further ado, dig out that soldering iron again (and be careful which end you hold)...

OUT AND ABOUT

To start with, we'll have a look at a few output circuits. Figure 1 shows a VMOS Power-MOSFET being used to drive a relay and an LED. The MOSFET (T1) will be turned on when the I/O port output goes high (about 5 volts), this in turn will energize the relay coil and cause LED1 to illuminate. You can use any of the I/O port terminals to control T1, just connect it directly to the terminal block that you wish to use as an output (see diagram from last month). The 0V to T1 connects to the 0V power rail, and the +Ve (DC) supply is whatever you decide to use to drive the relay. Diode D1 (1N4001 etc) is needed to absorb the reverse voltage generated by the relay coil as it de-energizes.

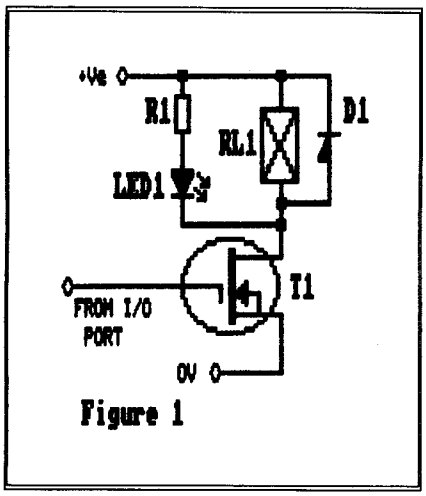


Figure 1

Choose T1 to suit the current requirements of the relay coil, for instance if the relay requires about 200mA T1 could be an inexpensive device such as a VN10KM or VN10LM.

To give about 10mA for the LED, R1 would need to be 1K0 for a 12V supply, or about 200R for a 5V supply. Just for trying-out the example programs in this article, you could leave-out the relay (and D1) and connect the +Ve terminal to the port's 5V power rail, as the CPC won't object to supplying a few milliamps for the LED. R1 will of course need to be 200 Ohms or so.

PIO LIMITED

Normal (bipolar) transistors aren't really suitable in place of T1, as the PIO outputs have a very limited current capability; the exception being the ports PB (0 to 7). These outputs can drive Darlington (super-alpha pair) transistors; as shown in Figure 2. The value of R2 will depend upon the transistor used, but would typically be around 2K0 for an MPSA14 or a TIP122.

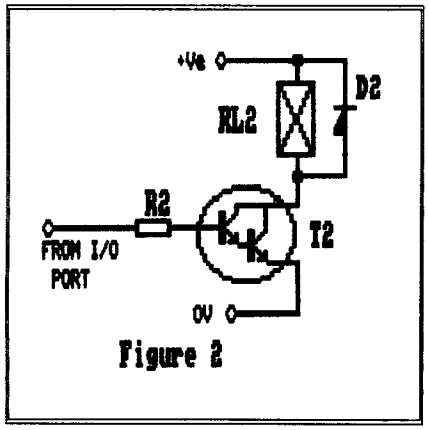


Figure 2

PORT AND AFTER EIGHTS

To use all eight pins of a port as outputs would of course need eight circuits like fig.1 or fig.2. Fortunately this has already been done in the form of a chip, the ULN2803 (see Figure 3). IC1 contains eight Darlington transistors, with a 2K7 resistor in series with each input. Each output can drive a 500mA load although you should derated this as more outputs are used (down to about 100mA per output if all outputs are used). Unused IC1 inputs or outputs can be left unconnected. The protection diodes (D1, D2) are already built into the IC, with the cathodes brought out to pin 10. Connect this pin to the supply side of the relay coils, or leave unconnected if you are driving non-inductive loads (lamps, LED's etc.)

A more readily obtainable device, the ULN2003 is almost the same; the only difference being that it has seven Darlington drivers instead of eight.

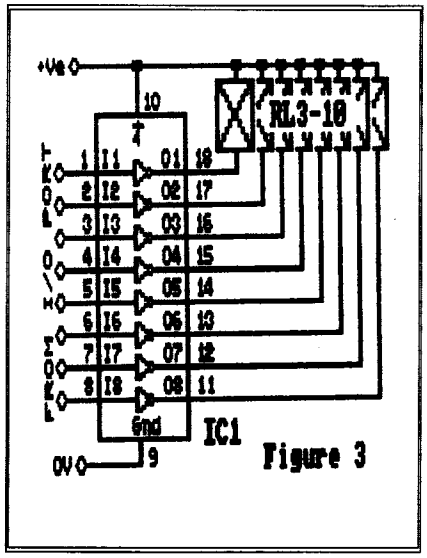


Figure 3

Any port inputs or outputs can instead go directly to other logic IC's without the need for any of these interface circuits; provided that they are also operating from a 5V supply.

THE IN CROWD

Figure 4 shows a simple way to connect a switch to a port input.

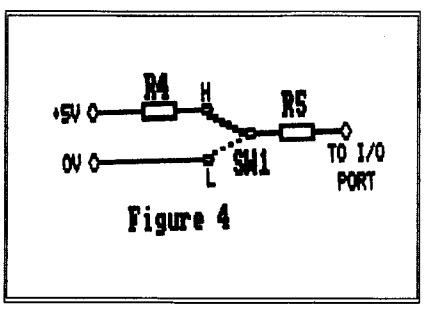


Figure 4

Resistor R5 (1K0) is there to protect the PIO just in case you program that pin as an output.
The switch should preferably be a break-before-make type, but if you do happen to use a make-before-break type then resistor R4 (1K0) will prevent you from shorting-out the power rails.
To experiment with then, I would suggest you use the input circuit of fig.4 and the output circuit of fig.1 (without the relay) as these can both be convieniently powered from the I/O port power rails.

MODUS OPERANDI

Each port has a Control Register, the contents of which determine how the port operates. A port can operate in one of four modes; Input, Output, Bidirectional, or Bit Control. We will be using the Bit Control mode (Mode 3) being the only mode which allows us to set-up each I/O line individually as an input or output. So, the first thing to do is to select a port and tell it to operate in Mode 3.

PB OR NOT PB

Each PIO chip (IC3 & IC4) has two ports, called PA and PB. PORT 0 is therefore port PA of IC3. The addresses required were listed last month, so we can see from the list that we need to select address &FAEC for the control register of port PA of IC3, which is the control register for our PORT 0.

So, we now know which address we want, but what data do we send to it? Figure 5 shows the data that needs to be sent to select the different Modes.

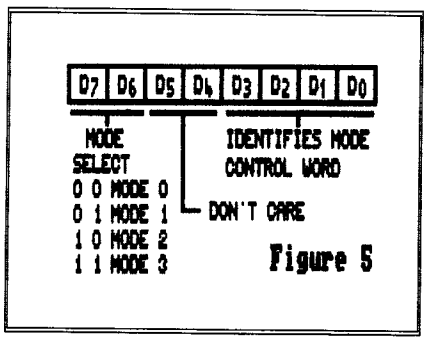


Figure 5

D0 to D3 need to be set to 1, this tells the IC that you are setting the Mode. D4 and D5 can be set to 0 or 1, it doesn't matter which. D6 and D7 select the Mode to be used. In our case we want Mode 3, so these two bits need to be set to 1. If we set D4 and D5 to 1, then this gives us a nice easy number to remember:

11111111

This is of course in binary, so on the CPC we should type it as &X11111111 (&X denotes a binary number). This can be written in hexadecimal as &FF or in denary (decimal) as 255.
If you want to use the CPC to convert from binary to decimal then you can use PRINT &X11111111
This will display the decimal number 255. If you want to convert the other way around, from decimal to binary, then use PRINT BIN$(255,8)
This will convert 255 into the binary equivalent, displayed as an eight bit number; ie. 11111111
From now on I shall show all binary numbers with the &X prefix.
To access an address, we need to use the OUT command. To send 255 to I/O address &FAEC we can use this command:-

```
OUT &FAEC,255
```

If you prefer not to convert to decimal, then you can use:-

```
OUT &FAEC,&X11111111
```

IN, OUT, AND SHAKE IT ALL ABOUT

Once you've sent the Mode Control Word (which we've just done) the PIO will expect us to send something called the I/O Register Control Word. This is the word which tells the port which lines to set to inputs and outputs, and must be sent next.
Figure 6 shows how the word representsthe individual I/O lines of a port. In our case, we need to set port PA of IC3, so, figure 6 represents PA7 to PA0 (in other words, our PORT 0).

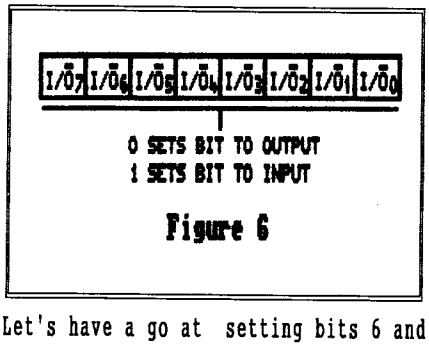


Figure 6

Let's have a go at setting bits 6 and 7 as outputs, and bits 0 to 5 as inputs (this will mean that the terminals T00 to T05 will be inputs, and T06 and T07 will be outputs).
Bits 6 and 7 need to be set to 1 (for inputs) and the rest to 0 (for outputs) to give:-

&X00111111

In decimal this is 63, so we now need to send 63 to address &FAEC like so:-

```
OUT &FAEC,63
```

Or, if you prefer, in binary:-

```
OUT &FAEC,&X00111111
```

Up to now then, we've managed to set PORT 0 to operate in Mode 3, with bits 6 and 7 (T06 & T07) as outputs, and bits 0 to 5 (T00 to T05) as inputs.
As another example, suppose we wanted to program PORT 3 as follows:
Inputs - T30, T31, T32, T35, T37
Outputs - T33, T34, T36
Again, from last month's WACCI we can see that PORT 3 is port PB of IC3, and it's Control Register is accessed from I/O address &FAFE.
First of all then, set the Control Register to operate in Mode 3:-

```
OUT &FAFE,&X11111111  (see fig.5)
```

Then send the I/O Register Control Word, remembering that the inputs will be bits 0,1,2,5,7 and the outputs will be bits 3,4, and 6:-

```
OUT &FAFE,&X10100111
```

Right then, that's the hard work over; the easy bit is to get data in and out of the ports.
As we've got the addresses for PORT 3 fresh in our minds (slapped wrists for anyone who's forgotten!) we'll attempt to set bits 3 and 4 low (logic zero) and bit 6 high (logic 1).
Looking at last month's address list again, we can see that the required address is &FAFA (because we want to send data out of port PB of IC4).
The data to be sent is:-

&Xx1x00xxx

Where x can be either 0 or 1, it doesn't matter. We have already told the PIO chip that bits 0,1,2,5, and 7 are inputs, so any OUTPUT data sent to these bits will be ignored. Replacing the x's with zeroes then, gives us:-

&X01000000

To set output bit 6 high then, we need to use:-

```
OUT &FAFA,&X01000000
```

To set it low again, just replace the 1 with 0, like so:-

```
OUT &FAFA,&X00000000
```

Let's set output bits 3 & 6 high with:

```
OUT &FAFA,&X01001000
```

Now, suppose we want to set bit 6 low but we want to leave bits 3 and 4 unchanged; the command would be:-

```
OUT &FAFA,&X00001000
```

This will set bit 6 low, bit 4 low, and bit 3 high, resulting in a change only in bit 6.

32 BIT IO-PORT PART THREE

MARTIN BELA

In this final part we'll manipulate individual bits of a port, and also look at a couple of circuits; but first let's continue from where we left-off last month...

SETTING A BIT HIGH

The easiest way to alter some bits without changing others is by using the logical operators OR and AND.
To start with, the current bit values need to be stored in a variable, which we'll call "current", so, using the example from last month:-

```
current=&X00001000
```

Next, decide which bit is to be set high, and create a new variable with that bit set to 1, and the bits that we don't want to change set to 0.
We'll call this variable "bitmask", as it 'masks-out' the bits that we do not want to change.
So, to set bit 6 high again:-

```
bitmask=&X01000000
```

The next thing to do is to OR these two values together:-

```
  current=&X00001000
OR bitmask=&X01000000
   result=&X01001000
```

The OR command works by comparing one bit of one variable with the same bit of the other variable. The result will be 1 only if that bit of either variable is 1; otherwise the result will be 0. Looking at bit 7 then, *current* and *bitmask* are both 0, so the result is also 0; bit 6 of *current* is 0 but bit 6 of *bitmask* is 1, so the result is 1, and so on for the rest of the bits. This means that any bit of *result* will be forced to 1 if *bitmask* is 1, but if *bitmask* is 0 then *result* will only be 1 if *current* is already 1 thus preserving the original value of that bit in *current*.
The only thing left to do then is to send the new value out to the port, then transfer *result* back into *current* to keep *current* updated with the new port settings.

In BASIC the above operation would look like this:-

```
180 ...
190 ...
200 bitmask=&X01000000
210 result=current OR bitmask
220 OUT &FAFA,result
230 current=result
240 ...
250 ...
```

You can of course alter more than one bit at a time; just put a 1 in each bit position of *bitmask* that you want to set high.

SETTING A BIT LOW

To set bits low again we can use the logical operator AND. Taking the above example again:-

```
   current=&X01001000
```

This time, to set bit 6 low, we need to put a 0 in bit 6 position of *bitmask*, and put a 1 in the other bit positions, then AND it with *current*.

```
   current=&X01001000
AND bitmask=&X10111111
    result=&X00001000
```

This time, each bit of *current* is compared with the corresponding bit of *bitmask*, and that bit of *result* is only set to 1 if they are both 1. If either (or both) bits are 0, then the *result* bit is 0. It should be fairly clear then that a 0 in *bitmask* will force a *result* bit of 0, but a 1 in *bitmask* will preserve the *current* bit value (at least, I *hope* it's clear!).

The only thing left to do then is to send the new value out to the port, and update *current*.

In BASIC the above operation would look like this:-

```
180 ...
190 ...
200 bitmask=&X10111111
210 result=current AND bitmask
220 OUT &FAFA,result
230 current=result
240 ...
250 ...
```

Once again, you can alter several bits at a time; just put a 0 in each bit position of *bitmask* that you want to set low.

GATHER YOUR BITS IN

To get data into the computer from the port we use the INP command. eg:-

```
datin=INP(&FAFA)
```

This will read-in the data from PORT 3 and put it into variable *datin*.
All of the bit values will be read, inputs *and* outputs. Reading the value of an input or output doesn't affect the state of it, by the way.
To test for a particular bit being high or low we need to use a logical operator again, namely AND. To see how this works, let's suppose we have just issued the instruction:-

```
datin=INP(&FAFA)
```

....and this has given us the value:-

```
datin=&X10010110
```

Now, to check the value of bit 2 we need another bit-mask again; this time with bit 2 set to 1, and all other bits set to 0. eg:-

```
     datin=&X10010110
AND bitmask=&X00000100
    result=&X00000100
```

If result is greater than zero then that bit is 1 (high). In BASIC then:-

```
180 ...
190 ...
200 datin=INP(&FAFA)
```

```
210 bitmask=&X00000100
220 result=bitmask AND datin
230 IF result>0 THEN PRINT "Bit 2 is
high" ELSE PRINT "Bit 2 is low"
240 ...
250 ...
```

Various bits of *datin* can then be checked by setting different bits of *bitmask* high, and then ANDing them again. If you only wanted to check one bit, then the above program could be shortened to:-

```
180 ...
190 ...
200 IF INP(&FAFA) AND &X00000100 THEN
PRINT "Bit 2 is high" ELSE PRINT "Bit
2 is low"
210 ...
220 ...
```

These routines could be called on a regular basis by using the EVERY command:-

```
180 ...
190 ...
200 EVERY 50 GOSUB 220
210 GOTO 210
220 IF INP(&FAFA) AND &X00000100 THEN
PRINT "Bit 2 is high" ELSE PRINT "Bit
2 is low"
230 RETURN
```

The routine at line 220 will be called every second. The value after the EVERY command determines the time interval between calls in fiftieths of a second. To stop the routine from being called, assign a variable to REMAIN(0). eg. a=REMAIN(0)

JUST WAIT A BIT

One more command that you might find useful is WAIT. The syntax is:-

WAIT <port address>,<mask>

This causes the computer to wait until one or more bits of the mask number goes high at the specified I/O port address. For instance, if we want to wait for bit 2 of PORT 3 (&FAFA) to go high, then we would use:-

```
WAIT &FAFA,&X00000100
```

BASIC will wait indefinitely until this condition occurs. Several bits can be specified, and BASIC will wait until one or more of those bits goes high, although it will not indicate which bit (or bits) has gone high.

WAIT BE BUGGED!

Actually, WAIT should be able to do a little more, as the full syntax is:-

WAIT <port addr>,<mask>,[<inversion>]

The inversion part is optional, but I can't get it to work! I don't know if it's just my computer or if it's a bug in the BASIC, although my CPC6128 is the 1985 version, perhaps the later one was fixed. Anyway, if you want to see if yours works properly then first of all switch-off and unplug everything from the expansion port of your CPC. Now switch on again and type in this command:

```
PRINT INP(&FAFA)
```

You should get the value 255 returned to your screen. Now type:

```
WAIT &FAFA,255
```

We already know that 255 is on I/O address &FAFA, so BASIC will return immediately with the Ready prompt and the cursor blob. So far so good.
Now for the real test; enter the following command:

```
WAIT &FAFA,255,0
```

BASIC will either return immediately with the Ready prompt and cursor blob (if it's working correctly), or (if the WAIT command is bugged) it will wait until you reset the computer.
I would be interested to know if anyone gets this to work properly.
If your full version of the WAIT command *does* work, then here's how to use it.
First of all, decide which I/O port address to use; for instance &FAFA.
Then, decide which bits you want to check. Let's suppose you want to look at bits 0,1,2,5, and 6.
Next, decide what value you want to test for in each bit. For this example we'll look for the following:-

BIT NUMBER	VALUE
0	1 (high)
1	1 (high)
2	0 (low)
5	1 (high)
6	0 (low)

This is represented by:-

```
&Xx01xx011  where x=don't care
```

We now need to make an inversion value by replacing the above 0's with 1's, and replacing the 1's with 0's:-

```
&Xx10xx100
```

We can now replace all the x's with 0's to give:-

```
&X01000100
```

This is our inversion value.
We now need to make the mask. This is actually another one of those AND bit-masks. Each bit that you want to test (as in the table above) should be set to 1 in the mask, and the other bits set to 0. Like so:-

```
&X01100111
```

This is our AND mask.
These values can now be inserted into the WAIT command:-

```
WAIT &FAFA,&X01100111,&X01000100
```

BASIC will now wait until one or more of the bits that we are testing are set to the required value; such as a 1 on bit 0, or a 1 on bit 1, or a 0 on bit 2 etc although once again, WAIT will not tell us *which* bit.

An alternative way to use the inversion value is if you just want to test for a *change of state* in the bits you are testing. To do this you will first need to know the current state of the port; you can use the INP command for this. eg:

```
current=INP(&FAFA)
```

Then use this as the inversion value, and set to 1 all the bits in the AND mask that you wish to test for a change in state. eg:

```
WAIT &FAFA,&X01100111,current
```

This will wait for a change in state of bit 0,1,2,5, or 6.

A BIT OF THE HARD STUFF

That's it as far as programming is concerned, so we might as well have a look at some more hardware ideas.

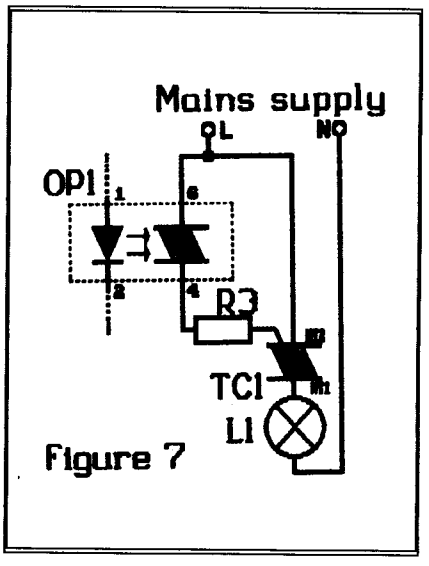


Figure 7

Figure 1 (last month's issue) shows an LED to indicate when an output is on (high). This LED could just as easily be an Infra-Red type, coupled to a photo-transistor via a length of fibre-optic cable. This allows signals to be sent along long cables without the problems of picking-up electrical noise etc. An added bonus is that the transmitting and receiving ends are very well electrically isolated from each other.

If you want the electrical isolation but without the fibre-optic cables, you could use an opto-isolator as shown in Figure 7.

This is suitable for switching mains loads without the need for relays (which can take a lot more pcb space, and will eventually wear-out).

The opto-isolator OP1 should have pins 1 & 2 wired in place of LED1 (see fig.1 again). This device has a triac which is switched-on by the light from the (internal) LED. This triac can then switch-on a much larger triac TC1 which drives the load L1 (a lamp, motor, heater or whatever).

For mains operation, TC1 should be rated at 400V or greater, and have a current rating of, say, twice the load current. The value of R3 will depend upon the sensitivity of TC1, but would typically be in the 1k0 to 2k0 range.

Suitable types for OP1 are:-

TRIAC OPTOISOLATOR	(from Maplin)
MOC 3020	(from RS)
S21MD3V	(from RS)

The pin-outs for all these types are the same as fig.7.

NOW FOR THE BOUNCY BITS

Figure 8 shows a switch "de-bounce" circuit made from a pair of cross-coupled TTL NAND gates.

When a switch is operated, the contacts may bounce for a short time before settling in their new position; this would result in the I/O port receiving a series of pulses during the bounce period. This circuit gives a clean switch-over from one level to the other without passing the bounce pulses through. The resistors should be around 1k0 to 2k0.

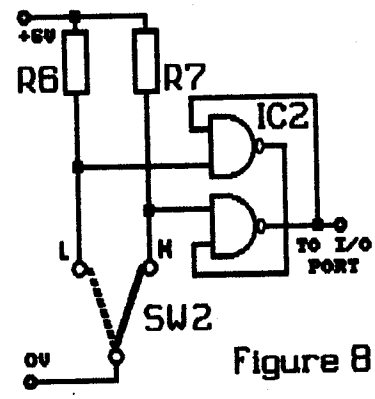


Figure 8