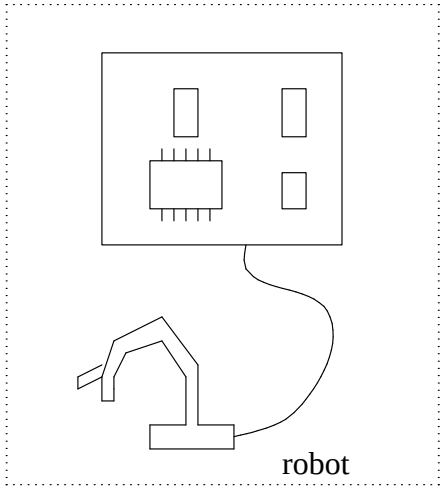
Controlling hardware from a microkernel

- consider a microkernel which controls a robot:



- the software has to tell the robot where to move

Controlling hardware from a microkernel

- depending on the hardware interface this could be complex or easy!
  - typically a robot may require two values: a vector and a time in which to arrive at the destination
  - the software would write these values to a *device register*
  - the software may have to wait until the robot has finished its current maneuver before issuing another request

Device control (continued)

- hardware interface might indicate an error
  - unable to get to the required position
  - requires a recalibration

Device control (continued)

- generally device interfaces have similar properties

- device interfaces typically consist of three classes of registers
  - data registers (vector and time for robot)
  - control registers (recalibrate, abort, start, issue interrupt when finished)
  - status registers (command ok?, error?, device ready?)

Clock device on the IBM-PC

- a clock device is important for any microkernel
- it allows the system to take actions at specific times
  - software can utilize a single timer to implement many **virtual** timers (see later on for a full description of virtual timers).

Clock device on the IBM-PC

- typically microkernel software will *service* demands every time period. Obvious example is a graphic clock:
  - ```
    MakeVirtualTimer(1, MoveSecondHand);
    MakeVirtualTimer(60, MoveMinuteHand);
    MakeVirtualTimer(3600, MoveHourHand);
    ```
    ```

Example: floppy disk motor

- to be turned on before a read or write
- to be turned off after read or write
- it takes 1 second to spin floppy disk motor at full speed from standstill
  - must not forget to turn off floppy disk motor otherwise we ruin floppy disk! (Floppy disk drive heads rub the surface of a floppy disk)

Example: floppy disk motor

- use timer to:
  - wait 1 second before performing read/write after turning motor on
  - (inefficient to always turn off floppy disk motor at end of write)
  - we could wait for 3 seconds and then turn off motor
  - if another floppy disk read/write occurs before 3 seconds then we don’t need to wait for motor to get up to speed.
  - after subsequent read/write operations we *reprime* or *reactivate* the 3 second timer.

Clock device registers in IBM-PC

- at the low level the microkernel software has to ask the clock device to give us an interrupt at a specific time
  - in above examples - a one second granularity would suffice

8253 (Programmable Interval Timer)

- maybe configured to provide 3 separate count down timers
  - give a timer a value, it counts down and when value reaches zero it issues an interrupt

- device registers are:
  - `ControlWordReg` at (port `043H`) (8 bits long)
  - `Counter0` at (port `040H`) (8 bits long)

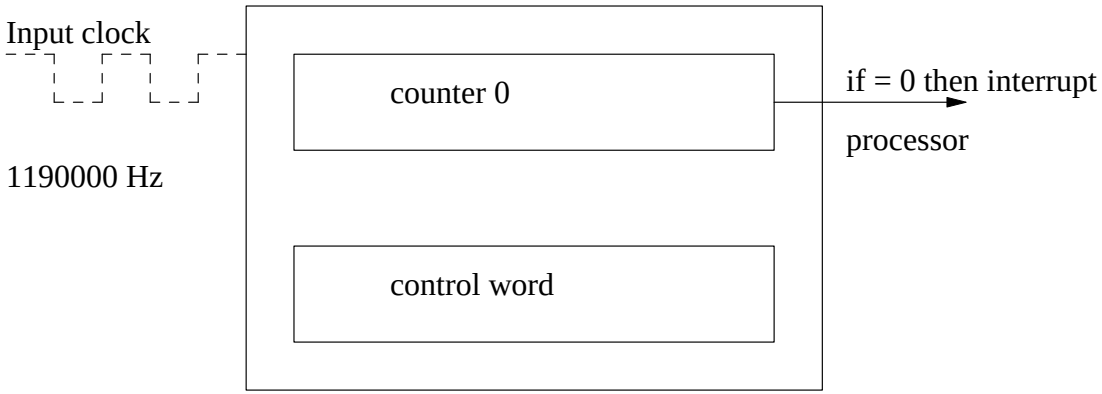
ControlWordReg

d7	d6	d5	d4	d3	d2	d1	d0
sc1	sc0	rl1	rl2	m2	m1	m0	bcd

- **sc** counter number 0..3
- **rl** read or load (3 = read or load lsb first then msb) (internally the counter is 16 bits long - but we have to give it 8 bits at a time)
- **m** different modes

8253

- we are only interested in count down and interrupt (mode 0).



8253

- on the IBM-PC when a clock counter value reaches 0 interrupt 32 is raised. (IRQ 0)

- the value in the clock counter is decremented 1190000 times a second
  - so if you wanted 20 interrupts every second your would put a value of 59500 into the clock counter
  - 1190000/59500 = 20

8253 (continued)

- what is the largest interrupt period that can be programmed?

- $1190000/2^{16} = 1190000/65536 = 18.157$
  - 18.157 interrupts per second = 18.157 Hz
  - period = $\frac{1}{Hz} = \frac{1}{18.157} = 0.0550$ second

Software control of the 8253

- to control the 8253 through C we could devise two routines:
  - `ClockDevice_StartCount` - initializes counter0 with a value
  - `ClockDevice_LoadCount` - returns the value of counter0

Software control of the 8253

```
/*
 *  LoadCount - returns the value of counter0.
 */

unsigned int ClockDevice_LoadCount (void)
{
   unsigned int lo;
   unsigned int hi;

   /* Tell 8253 that we wish to Read or Write */
   /* to it. Mode 0 Counter 0.                */
   Out8(ControlWordReg, (unsigned char)((1<<4)|(1<<5)));
   /* Least Significant Byte */
   lo = PortIO_In8(Counter0) ;
   /* Most significant Byte  */a
   hi = PortIO_In8(Counter0) ;
   return hi*0x100 + lo;
}
```

Software control of the 8253

```
/*
 *  StartCount - initialize counter0 with Count.
 */

void Device_StartCount (unsigned int Count)
{
   /* Tell 8253 that we wish to Read or Write */
   /* to it. Mode 0 Counter 0.                */
   Out8(ControlWordReg, (unsigned char)((1<<4)|(1<<5)));
   /* Least Significant Byte */
   Out8(Counter0, (unsigned char) (Count % 0x100));
   /* Most significant Byte  */
   Out8(Counter0, (unsigned char) (Count / 0x100));
}
```

Definition of In8 and Out8

- where In8 and Out8 are defined as follows:

```
/*
 *  In8 - returns a BYTE from port, Port.
 */

unsigned char PortIO_In8 (unsigned int Port);

/*
 *  Out8 - sends a byte, Value, to port, Port.
 */

void PortIO_Out8 (unsigned int Port, unsigned char Value);
```

Interrupts

- remember that, typically, there are 4 different types of interrupt on a computer:
  - I/O interrupt - 8253 counter0 = 0
  - user code issuing a software interrupt - user code calling the operating system. Ie MS-DOS INT 021H.
  - illegal instruction (divide by zero, or an opcode which the processor does not recognize).
  - memory management fault interrupt (occurs when code attempts to read from non existent memory).

Interrupts

- an interrupt will cause the processor to execute some specialist code dedicated for this purpose
  - the processor can ignore the first class of interrupt (it can disable interrupts)
  - in the microkernel this is achieved by: `TurnInterrupts(Off)`
  - when `TurnInterrupts(On)` occurs then any pending interrupts will be serviced
  - it cannot ignore the later 3 classes of interrupt

What happens when counter0 equals 0?

- a hardware interrupt occurs.
- when 8253 issues an interrupt the processor will start to execute a specific routine dedicated to servicing this device

What happens when counter0 equals 0?

- typically the functionality of this routine might be:

```
ticks = 0;
seconds = 0;
while (TRUE) {
   /* wait for 1/20 of a second */
   ClockDevice_StartCount(59500) ;
   EnableProcessorInterrupts ;
   WaitForClockInterrupt ;
   DisableProcessorInterrupts ;
   ticks++;
   if (ticks == 20) {
      seconds++;
      ticks = 0;
   }
}
```

Sample interrupt service routine

- this simple routine maintains a seconds count
  - the routine **WaitForClockInterrupt** will transfer control to something else that can be usefully done.
  - when the hardware interrupt occurs the processor temporarily suspends what is doing and resumes execution at the **DisableProcessorInterrupts**

d7	d6	d5	d4	d3	d2	d1	d0
sc1	sc0	rl1	rl2	m2	m1	m0	bcd

The type BITSET

- to load a variable with the control word register you could write the following code:

```
VAR
   CopyOfControl: BITSET ;
BEGIN
   (* firstly get a copy of the ControlWordReg *)
   CopyOfControl := VAL(BITSET, In8(040H)) ;
```

- to find out whether bit 0 has a value of 1 you could write the following code:

```
IF 0 IN CopyOfControl
THEN
   (* bit0 value is 1 *)
ELSE
   (* bit0 value is 0 *)
END
```

The type BITSET

- the type `BITSET` is a set of bits. Each bit can only contain a binary value (0 or 1).
  - In Modula-2 under GNU/Linux the `BITSET` has 32 bits (as the i486 is a 32 bit architecture)
  - to set a bit say bit 5 to a value of 1

- ```
  INCL(CopyOfControl, 5)
  ```
  - set set a bit say bit 5 to a value of 0

- ```
  EXCL(CopyOfControl, 5)
  ```

Bit manipulation

- to set all bits to zero
-
```
CopyOfControl := {}
```
- or if you wanted to set bits `7, 5, 2` all to a value of `1` you could use:
-
```
CopyOfControl := {7, 5, 2}
```
- alternatively you could:
-

```
CopyOfControl := {} ;
INCL(CopyOfControl, 2) ;
INCL(CopyOfControl, 5) ;
INCL(CopyOfControl, 7) ;
```


Bit manipulation

- or

Bit manipulation

```
CopyOfControl := VAL(BITSET, 01010100B)
```

- which bits are set to a value of 1 in ControlWordReg after the following code has been executed?

```
CopyOfControl := {4, 3, 2, 1, 0} ;
EXCL(CopyOfControl, 0) ;
EXCL(CopyOfControl, 2) ;
EXCL(CopyOfControl, 4) ;
INCL(CopyOfControl, 5) ;
INCL(CopyOfControl, 7) ;
```

Bit manipulation

- answer? {7, 5, 3, 1}
- how would you test to see whether bit 3 is set?
-

```
IF 3 IN ControlWordReg
THEN
   (* bit3 is set *)
END
```