

Listing: IRQUTIL.PAS

```
{*****}
{          I R Q U T I L   . P A S          }
{*-----*}
{ Task           : Provides functions for hardware interrupt }
{               programming.                               }
{*-----*}
{ Author          : Michael Tischer / Bruno Jennrich       }
{ Developed on    : 03/12/1994                             }
{ Last update    : 10/08/1994                             }
{*****}
```

Unit IRQUTIL;

Interface

Const

```
{- IRQ controller port addresses -----}
MASTER_PIC      =   $20;                { Base address of Master-PIC }
SLAVE_PIC       =   $A0;                { Base address of Slave-PIC }
IRQ_MASK        =   $01;                { Offset to masking port }

{- IRQ Commands -----}
EOI              =   $20;                { Unspecified End of Interrupt }

MASTER_FIRST_VECTOR = $08;              { Software vectors of the hardware }
SLAVE_FIRST_VECTOR  = $70;              { Interrupts }

{-- Prototypes -----}
```

```

Procedure irq_Enable      ( iIRQ : Integer );
Procedure irq_Disable     ( iIRQ : Integer );
Procedure irq_SendEOI     ( iIRQ : Integer );
Function  irq_SetHandler( iIRQ : Integer; lpHandler : Pointer ) : Pointer;
Function  irq_ReadMask   ( iController : Integer ) : Byte;
Function  irq_ReadISR     ( iController : Integer ) : Byte;
Function  irq_ReadIRR     ( iController : Integer ) : Byte;

```

Implementation

Uses DOS;

```

{*****}
{  irq_Enable : Enable hardware interrupt  }
{-----*}
{  Input : iIRQ - Number of hardware interrupt (0-15), that is }
{           to be enabled                                     }
{-----*}
{  Info : IRQs are enabled and disabled by                     }
{           sending an OCW1 (Operation Control Word) named byte }
{           to the base port of the IRQ controller.           }
{*****}

```

```

Procedure irq_Enable( iIRQ : Integer );

```

```

var iPort : Integer;

```

Begin

```

{-- get port address of appropriate PIC first -----}
{ ( 0-7 = MASTER_PIC , 8-15 = SLAVE_PIC ) }
if iIRQ <= 7 then iPort := MASTER_PIC
                else iPort := SLAVE_PIC ;
iPort := iPort + IRQ_MASK;      { Choose masking port }

```

```

iIRQ := iIRQ and $0007;      { Get PIC interrupt number (0-7) }
                              { Clear bit -> Interrupt enabled }
port[iPort] := port[iPort] and not ( 1 shl iIRQ );
End;

{*****}
{ irq_Disable : Suppress hardware interrupt }
{ *-----* }
{ Input : iIRQ - Number of hardware interrupt (0-15) to be }
{           forbidden }
{*****}
Procedure irq_Disable( iIRQ : Integer );

var iPort : Integer;
Begin
  {-- get port address of appropriate PIC first -----}
  { ( 0-7 = MASTER_PIC , 8-15 = SLAVE_PIC ) }
  if iIRQ <= 7 then iPort := MASTER_PIC
    else iPort := SLAVE_PIC ;

  iPort := iPort + IRQ_MASK;      { Choose masking port }

  iIRQ := iIRQ and $0007;      { Get PIC interrupt number (0-7) }
                              { Set Bit -> Interrupt locked }
  port[iPort] := port[iPort] or ( 1 shl iIRQ );
End;

{*****}
{ irq_SendEOI : Signal "End Of Interrupt" }
{ *-----* }
{ Input : iIRQ : Number of hardware interrupt (0-15) that has been }
{           completely processed. }
{*****}

```

```

{ Info : - An interrupt cannot be triggered a second time until
  after the previous interrupt has been completely
  processed. To let the PICs know that an interrupt handler
  has been terminated, an EOI command must be sent to the
  PICs.
}
{*****}

```

```

Procedure irq_SendEOI( iIRQ : Integer );

```

```

Begin

```

```

  {-- With IRQ 8 - 15 inform Slave as well --}
  if iIRQ > 7 then port[ SLAVE_PIC] := EOI;
  port[ MASTER_PIC] := EOI;    { Always signal EOI to MASTER }
End;

```

```

{*****}
{ irq_SetHandler : Install new interrupt handler
  *-----*
  Input : iIRQ      - Number of interrupt the new handler
                  is to receive.
          lpHandler - Address of new handler
  Output : Address of old handler.
}
{*****}

```

```

Function irq_SetHandler( iIRQ : Integer; lpHandler : Pointer) : Pointer;

```

```

var lpOldHandler : Pointer;
    iVect        : Integer;

```

```

Begin

```

```

  {-- Get interrupt vector of hardware interrupt -----}
  { IRQ 0 - 7 = Vectors $08 - $0F }
  { IRQ 8 - 15 = Vectors $70 - $77 }
  if iIRQ <= 7 then

```

```

    iVect := ( MASTER_FIRST_VECTOR + iIRQ )
else
    iVect := ( SLAVE_FIRST_VECTOR + ( iIRQ and $7 ) );

irq_Disable( iIRQ );    { Disable hardware and software interrupt }
asm cli end;

GetIntVec( iVect,lpOldhandler );          { Save old handler }
SetIntVec( iVect,lpHandler );              { Set new handler }

asm sti end;                          { Allow software interrupts }

{- In case a handler was passed, allow corresponding --}
{- hardware interrupt again                                --}
if lpHandler <> NIL then
    irq_Enable( iIRQ );

    irq_SetHandler := lpOldHandler; {Return address of old handler}
End;

{*****}
{ irq_ReadMask : Read masks of an IRQ controller }
{*****}
{
  *-----*
  Input : iController - Base port of IRQ controller ($20/$A0)
  Output : Masking of IRQs serviced by IRQ controller.
           Set Bits : IRQ is not triggered.
  *-----*
  Info : The maskings of a PIC can be determined by simply
         reading out the mask register. ($21/$A1)
  *****}
Function irq_ReadMask( iController : Integer ) : Byte;

```

```

Begin
  irq_ReadMask := port[iController + IRQ_MASK];
End;

```

```

{*****}
{  irq_ReadISR : Read status register of an IRQ controller  }
{-----*}
{  Input : iController - Base port of IRQ controller ($20/$A0) }
{  Output : Status register of IRQs serviced by IRQ controller. }
{           Set Bits : the appropriate IRQ routine is either }
{           executed now or interrupted by an IRQ with }
{           higher priority. }
{-----*}
{  Info : Before the IRQ controller outputs the contents of the }
{           ISR register, the OCW3 named byte must be passed to the }
{           controller. }
{*****}
Function irq_ReadISR( iController : Integer ) : Byte;

```

```

Begin
  {-- OCW3: $0B = No operation , no poll, read ISR -----}
  port[iController] := $0B;
  irq_ReadISR := port[iController ];
End;

```

```

{*****}
{  irq_ReadIRR : Reading request register of an IRQ controller }
{-----*}
{  Input : iController - Base port of the IRQ controller ($20/$A0) }
{  Output : Request register IRQs serviced by IRQ controller. }
{           Set Bits : execute the appropriate IRQ routine next }
{           unless a request with higher priority }
{-----*}

```

```

{
    is pending.
}
{*-----*}
{ Info : Before the IRQ controller outputs the contents of the
{   ISR-Registers, the OCW3 named byte must be sent to the
{   controller.
{ *****
Function irq_ReadIRR( iController : Integer ) : Byte;
Begin
    {-- OCW3: $0A = No operation , no poll, read IRR -----}
    port[iController] := $0A ;
    irq_ReadIRR := port[iController];
End;

End.

```