

```

*****
S E R U T I L . P A S
*****
}
{ Task      : Functions for direct access to
              serial port
}
*****
{ Author      : Michael Tischer / Bruno Jennrich
{ Developed on : 04/08/1994
{ Last update  : 04/07/1995
{*****
{$X+}                                     { Function results optional }
Unit SERUTIL;

Interface

Uses WIN;
Const
SER_COM1 = $3F8;                        { Base address COM1 }
SER_COM2 = $2F8;                        { Base address COM2 }

SER_IRQ_COM1 = 4;                       { IRQ 4 = vector $0C }
SER_IRQ_COM2 = 3;                       { IRQ 3 = vector $0B }

SER_TXBUFFER = $00;                     { Transmit register }
SER_RXBUFFER = $00;                     { Receive register }
SER_DIVISOR_LSB = $00;                   { Baud rate divisor LSB }
SER_DIVISOR_MSB = $01;                   { Baud rate divisor MSB }
SER_IRQ_ENABLE = $01;                    { Interrupt enable register }
SER_IRQ_ID = $02;                        { Interrupt ID register }
SER_FIFO = $02;                          { FIFO register }
SER_2Function = $02;                     { Alternate function register }
SER_LINE_CONTROL = $03;                  { Line control }
SER_MODEM_CONTROL = $04;                 { Modem control }
SER_LINE_STATUS = $05;                   { Line status }
SER_MODEM_STATUS = $06;                  { Modem status }
SER_SCRATCH = $07;                       { Scratch register }

{ IRQ enable register bits (enable/disable interrupts) }
SER_IER_RECEIVED = $01;                  { IRQ after data received }
SER_IER_SENT = $02;                      { IRQ after byte sent }
SER_IER_LINE = $04;                       { IRQ after line status change }
SER_IER_MODEM = $08;                      { IRQ after modem status change }

{ IRQ-ID - bits (What initiated IRQ?) }
SER_ID_PENDING = $01;                     { Is serial IRQ pending? }
SER_ID_MASK = $06;                        { ID is coded in bits 1 and 2 }
SER_ID_LINESTATUS = $06;                  { Line status (error or break) }
SER_ID_RECEIVED = $04;                     { Data received }
SER_ID_SENT = $02;                         { Byte was sent }
SER_ID_MODEMSTATUS = $00;                 { CTS, DSR, RI or RLSD change }

{ Bit assignment in FIFO register (16550A UART or later) }
SER_FIFO_ENABLE = $01;
SER_FIFO_RESETPRECEIVE = $02;
SER_FIFO_RESETPRETRANSMIT = $04;

{ FIFO bits (number of bytes in FIFO after which IRQ occurs) }
SER_FIFO_TRIGGER0 = $00;                  { Normal }
SER_FIFO_TRIGGER4 = $40;                  { 4 bytes }
SER_FIFO_TRIGGER8 = $80;                  { 8 bytes }
SER_FIFO_TRIGGER14 = $C0;                 { 14 bytes }

{ Line control register bits (transmission parameters) }
SER_LCR_WordLEN = $03;                    { Number of bits being transmitted }
SER_LCR_5BITS = $00;
SER_LCR_6BITS = $01;
SER_LCR_7BITS = $02;
SER_LCR_8BITS = $03;
SER_LCR_2STOPBITS = $04;                  { 2 or 1.5 stop bits }
SER_LCR_1STOPBIT = $00;                   { 1 stop bit }

SER_LCR_NOPARITY = $00;                    { Disable parity check }
SER_LCR_ODDPARITY = $08;                   { Odd parity }
SER_LCR_EVENPARITY = $18;                  { Even parity }
SER_LCR_PARITYSET = $28;                   { Parity bit always set }
SER_LCR_PARITYCLR = $38;                   { Parity bit always cleared }
SER_LCR_PARITYMSK = $38;

SER_LCR_SENDBREAK = $40;                   { Send break as long as bit is set }
SER_LCR_SETDIVISOR = $80;                  { For access to baud rate divisor }

{ Modem control register bits (signal control) }
SER_MCR_DTR = $01;                        { Set DTR signal }
SER_MCR_RTS = $02;                        { Set RTS signal }
SER_MCR_UNUSED = $04;
SER_MCR_IRQENABLED = $08;                  { Issue IRQs to IRQ controller }

```

```

SER_MCR_LOOP      = $10;                                { Self-test }

{ Line status register bits (transmission error) }
SER_LSR_DATA RECEIVED = $01;    { Receive data word (5 - 8 bits) }
SER_LSR_OVERRUN ERROR = $02;    { Previous data word lost }
SER_LSR_PARITY ERROR = $04;     { Parity error }
SER_LSR_FRAMING ERROR = $08;    { Start/stop bit error }
SER_LSR_BREAK DETECT = $10;     { Break detected }
SER_LSR_ERROR MSK = ( SER_LSR_OVERRUN ERROR or SER_LSR_PARITY ERROR or
SER_LSR_FRAMING ERROR or SER_LSR_BREAK DETECT );
SER_LSR_THREMPY = $20;
SER_LSR_TSREMPY = $40;

{ Modem status register bits (which signals are set) }
{ Delta... bits indicate whether status of corresponding }
{ signals has changed since the last read on }
{ modem status register. }
SER_MSR_DCTS = $01;    { Delta CTS (status in CTS) }
SER_MSR_DDSR = $02;    { Delta DSR (status in DSR) }
SER_MSR_DRI = $04;     { Delta RI (status in RI) }
SER_MSR_DCD = $08;     { Delta CD (status in CD) }
SER_MSR_CTS = $10;     { Clear To Send set }
SER_MSR_DSR = $20;     { Data Set Ready set }
SER_MSR_RI = $40;      { Ring Indicator set }
SER_MSR_CD = $80;      { Carrier Detect set }

NOSER = 0;
INS8250 = 1;          { National Semiconductor UART's }
NS16450 = 2;
NS16550A = 3;
NS16C552 = 4;

SER_MAXBAUD = 115200; { Maximum baud rate }

SER_SUCCESS = 0;
SER_ERR SIGNALS = $0300;
SER_ERR TIMEOUT = $0400;

Function ser_UARTType( iSerPort : Integer ) : Integer;

Function ser_Init( iSerPort : Integer;
lBaudRate : longint;
bParams : Byte ) : Integer;

Procedure ser_FIFOLevel( iSerPort : Integer; bLevel : Byte );

Function ser_IsDataAvailable( iSerPort : Integer ) : Boolean;

Function ser_IsWritingPossible( iSerPort : Integer ) : Boolean;

Function ser_IsModemStatusSet( iSerPort : Integer;
bTestStatus : Byte ) : Boolean;

Procedure ser_SetModemControl( iSerPort : Integer;
bNewControl : Byte );

Function ser_WriteByte( iSerPort : Integer;
bData : Byte;
uTimeout : Word;
bSigMask,
bSigVals : Byte ) : Integer;

Function ser_ReadByte( iSerPort : Integer;
var Data : Byte;
uTimeout : Word;
bSigMask,
bSigVals : Byte ) : Integer;

Function ser_WritePacket( iSerPort : Integer;
pData : pointer;
iLen : Integer;
uTimeout : Word;
bSigMask,
bSigVals : Byte ) : Integer;

Function ser_ReadPacket( iSerPort : Integer;
pData : pointer;
iLen : Integer;
uTimeout : Word;
bSigMask,
bSigVals : Byte ) : Integer;

Procedure ser_CLRIRQ( iSerPort : Integer );

Procedure ser_SETIRQ( iSerPort : Integer );

```

```

Function ser_SetIRQHandler( iSerPort,
                           iSerIRQ   : Integer;
                           lpHandler : Pointer;
                           bEnablers : Byte) : Pointer;

Procedure ser_RestoreIRQHandler( iSerPort,
                                iSerIRQ   : Integer;
                                lpHandler : Pointer);

Procedure ser_PrintError( var Win : WINDOW; e : Integer );

Procedure ser_PrintModemStatus( var Win : WINDOW; iSerPort : Integer );

Procedure ser_PrintLineStatus( var Win : WINDOW; iSerPort : Integer);

Function ser_GetBaud( iSerPort : Integer ) : Longint;

Procedure ser_PrintCardSettings( var Win : WINDOW; iSerPort : Integer );

```

Implementation

```

Uses IRQUTIL;

```

```

type SerBuf      = array[0..65534] of byte;
SerBufPtr = ^SerBuf;

```

```

{*****}
{ ser_UARTType : Determine type of UART chip }
{*****}
{-----*-----}
{ Input : iSerPort      - base port of interface being tested }
{ Output : 0 (NOSER)    - no UART chip found }
{      1 (INS8250)      - INS8250 or INS8250-B chip }
{      2 (NS16450)      - INS8250A, INS82C50A, NS16450, NS16C450 }
{      3 (NS16550A)     - NS16550A chip }
{      4 (NS16C552)     - NS16C552 chip }
{*****}
Function ser_UARTType( iSerPort : Integer ) : Integer;

var b          : Byte;
    UartDetect : integer;

Begin
    UartDetect := -1; { -1 indicates not yet initialized }

    {- Check base capabilities ----- }
    port[iSerPort + SER_LINE_CONTROL] := $AA; { Divisor latch set }
    if port[iSerPort + SER_LINE_CONTROL] <> $AA then
        UartDetect := NOSER
    else
        Begin
            port[iSerPort + SER_DIVISOR_MSB] := $55; { Specify divisor }
            if port[iSerPort + SER_DIVISOR_MSB] <> $55
            then UartDetect := NOSER
            else
                { Clear divisor latch }
                Begin
                    port[iSerPort + SER_LINE_CONTROL] := $55;
                    if port[iSerPort + SER_LINE_CONTROL] <> $55 then
                        UartDetect := NOSER
                    else
                        Begin
                            port[iSerPort + SER_IRQ_ENABLE] := $55;
                            if port[iSerPort + SER_IRQ_ENABLE] <> $05 then
                                UartDetect := NOSER
                            else
                                Begin
                                    port[iSerPort + SER_FIFO] := 0; { Clear FIFO and IRQ }
                                    port[iSerPort + SER_IRQ_ENABLE] := 0;
                                    if port[iSerPort + SER_IRQ_ID] <> 1 then
                                        UartDetect := NOSER
                                    else
                                        Begin
                                            port[iSerPort + SER_MODEM_CONTROL] := $F5;
                                            if port[iSerPort + SER_MODEM_CONTROL] <> $15 then
                                                UartDetect := NOSER
                                            end;
                                        end;
                                    end;
                                end;
                            end;
                        end;
                    end;
                end;
            end;
        end;
    end;

    if UartDetect = -1 then { Not yet filtered out? }
        Begin { Looping }
            port[iSerPort + SER_MODEM_CONTROL] := SER_MCR_LOOP;
            b := port[iSerPort + SER_MODEM_STATUS];
            if ( port[iSerPort + SER_MODEM_STATUS] and $F0 ) <> 0 then
                UartDetect := NOSER
            else
                Begin

```

```

port[iSerPort + SER_MODEM_CONTROL] := $1F;
if ( port[iSerPort + SER_MODEM_STATUS] and $F0 ) <> $F0 then
    UartDetect := NOSER
else
    Begin
        port[iSerPort + SER_MODEM_CONTROL] := SER_MCR_DTR or
                                             SER_MCR_RTS;
        { Scratch register detected? }
        port[iSerPort + SER_SCRATCH] := $55;
        if port[iSerPort + SER_SCRATCH] <> $55 then
            UartDetect := INS8250
        else
            { FIFO detected ? }
            Begin
                port[iSerPort + SER_SCRATCH] := 0;
                port[iSerPort + SER_FIFO] := $CF;
                if ( port[iSerPort + SER_IRQ_ID] and $C0 ) <> $C0 then
                    UartDetect := NS16450
                else
                    Begin
                        port[iSerPort + SER_FIFO] := 0;
                        { Alternate function register detected? }
                        port[iSerPort + SER_LINE_CONTROL] := SER_LCR_SETDIVISOR;
                        port[iSerPort + SER_2Function] := $07;
                        if port[iSerPort + SER_2Function] <> $07 then
                            Begin
                                port[iSerPort + SER_LINE_CONTROL] := 0;
                                UartDetect := NS16550A;
                            End
                        else
                            Begin
                                port[iSerPort + SER_LINE_CONTROL] := 0; { Reset }
                                port[iSerPort + SER_2Function] := 0;
                                UartDetect := NS16C552;
                            End;
                        End;
                    End;
                End;
            End;
        End;
    End;
End;
ser_UARTType := UartDetect;
End;

{*****}
{ ser_Init : Initialize serial port }
{*****}
{-----*}
{ Input : iSerPort - base port of interface }
{          being initialized. }
{          lBaud    - baud rate ( 1 - 115200 ) }
{          bParams  - bit mask of remaining parameters }
{                   (s. SER_LCR... bits) }
{ Output : TRUE  - port initialized successfully }
{          FALSE - no port found }
{*****}

Function ser_Init( iSerPort : Integer;
                  lBaudRate : longint;
                  bParams   : Byte ) : Integer;

var uDivisor : Word;
    b       : Byte;
    uART     : Integer;

Begin
    uART := ser_UARTType( iSerPort );
    if uart = NOSER then
        Begin
            ser_Init := NOSER;
            exit;
        end;
    end;

    uDivisor := ( SER_MAXBAUD div lBaudRate );

    {-- Divide baud rate -----}
    port[iSerPort + SER_LINE_CONTROL] := { Enable divisor access }
    port[SER_LINE_CONTROL] or SER_LCR_SETDIVISOR;

    port[iSerPort + SER_DIVISOR_LSB] := LO( uDivisor );
    port[iSerPort + SER_DIVISOR_MSB] := HI( uDivisor );

    port[iSerPort + SER_LINE_CONTROL] := { Disable divisor access }
    port[SER_LINE_CONTROL] and not SER_LCR_SETDIVISOR;

    {-- Set other parameters only after resetting baud rate latch, --}
    {-- because this operation clears all --}
    {-- port parameters! --}

    port[iSerPort + SER_LINE_CONTROL] := bParams;

```

```

{ Read a byte, to reverse possible error }
b := port[iSerPort + SER_TXBUFFER];
ser_Init := uART;
End;

{*****}
{ ser_FIFOLevel : Set FIFO buffer size }
{*****}
{-----*}
{ Input : 0 - FIFO buffer size = 0, disable }
{ and reset (1 byte) }
{ SER_FIFO_TRIGGER4/8/14 - size = 4, 8 or 14 bytes }
{*****}
Procedure ser_FIFOLevel( iSerPort : Integer; bLevel : Byte );

Begin
  if bLevel <> 0 then
    port[iSerPort + SER_FIFO] := bLevel or SER_FIFO_ENABLE
  else
    port[iSerPort + SER_FIFO] := SER_FIFO_RESETRRECEIVE or
    SER_FIFO_RESESTRANSMIT;
End;

{*****}
{ ser_IsDataAvailable : Is data available to be read? }
{-----*}
{ Input : iSerPort - base port of interface being checked. }
{ Output : = 0 : No byte available to be read }
{ <> 0 : Byte is available }
{-----*}
{ Info : A byte is sent bit by bit, and becomes }
{ a complete byte again only when the receiving port }
{ has combined the individual bits. This is what is }
{ being checked by this function. }
{*****}
Function ser_IsDataAvailable( iSerPort : Integer ) : Boolean;

Begin
  ser_IsDataAvailable := ( port[iSerPort + SER_LINE_STATUS]
    and SER_LSR_DATA RECEIVED ) <> 0;
End;

{*****}
{ ser_IsWritingPossible : Can port send next byte? }
{-----*}
{ Input : iSerPort - base port of interface being checked. }
{ Output : = 0 : Byte cannot be sent. }
{ <>0 : Port ready to send. }
{-----*}
{ Info : A serial port should not be used }
{ to send a byte in the following cases: }
{ 1. A received byte has not yet been "retrieved" }
{ by the port. }
{ 2. An old send request has not yet been completed. }
{*****}
Function ser_IsWritingPossible( iSerPort : Integer ) : Boolean;

Begin
  ser_IsWritingPossible := ( port[iSerPort + SER_LINE_STATUS]
    and SER_LSR_TSREEMPTY ) <> 0;
End;

{*****}
{ ser_IsModemStatusSet : Check input line status }
{-----*}
{ Input : iSerPort - base port of interface. }
{ bTestStatus - bit pattern of lines being tested }
{ (CTS, DSR, RI, CD) }
{*****}
Function ser_IsModemStatusSet( iSerPort : Integer;
  bTestStatus : Byte ) : Boolean;

Begin
  ser_IsModemStatusSet := ( port[iSerPort + SER_MODEM_STATUS]
    and bTestStatus ) = bTestStatus;
End;

{*****}
{ ser_SetModemControl : Set signal lines for communication with }
{ modem etc. }
{-----*}
{ Input : iSerPort - base port of interface. }
{ bNewControl - new status of DTR, RTS etc. lines }
{*****}
Procedure ser_SetModemControl( iSerPort : Integer; bNewControl : Byte );

Begin
  port[iSerPort + SER_MODEM_CONTROL] := bNewControl;

```

```

End;

{*****}
{ ser_WriteByte : Send a byte }
{*****}
{-----*}
{ Input : iSerPort - base port of interface through which }
{               a byte is being sent. }
{       bData    - byte being sent }
{       uTimeout - number of passes through loop }
{               after which a timeout error occurs }
{               if the send was unsuccessful. (If }
{               iTimeout = 0 the system waits forever.) }
{       bSigMask - bit mask of signal lines being tested }
{               (RTS, CTS, CD, RI) }
{       bSigVals - signal line status }
{               after applying above mask. }
{ Output : = 0 - byte was sent }
{         <> 0 - error }
{*****}
Function ser_WriteByte( iSerPort      : Integer;
                       bData         : Byte;
                       uTimeout      : Word;
                       bSigMask, bSigVals : Byte ) : Integer;

Begin
  if uTimeout <> 0 then
    { Timeout loop }
    Begin
      While( not ser_IsWritingPossible( iSerPort )
            and ( uTimeout <> 0 ) )
      do Dec( uTimeout );
      if uTimeout = 0 then Begin
        ser_WriteByte := SER_ERRTIMEOUT;
        Exit;
      End;
    End
  else { Wait! }
  Repeat
    Until ser_IsWritingPossible( iSerPort );

  {-- Test signal lines -----}
  if ( port[iSerPort + SER_MODEM_STATUS] and bSigMask ) = bSigVals then
    Begin
      { Pass byte being sent to port }
      port[iSerPort + SER_TXBUFFER] := bData;
      { Return port error }
      ser_WriteByte := port[iSerPort + SER_LINE_STATUS] and
        SER_LSR_ERRORMSK;
    End
  else
    ser_WriteByte := SER_ERRSIGNALS;
End;

{*****}
{ ser_ReadByte : Receive byte }
{*****}
{-----*}
{ Input : iSerPort - base port of interface through which }
{               a byte is being received. }
{       Data      - byte variable accepting }
{               the received byte. }
{       uTimeout  - number of passes through loop }
{               after which a timeout error occurs }
{               if the receive was unsuccessful. (If }
{               iTimeout = 0 the system waits forever.) }
{       bSigMask  - bit mask of signal lines being tested }
{               (RTS, CTS, CD, RI) }
{       bSigVals  - signal line status after }
{               applying above mask. }
{ Output : = 0 - byte was sent }
{         != 0 - error }
{*****}
Function ser_ReadByte( iSerPort      : Integer;
                      var Data      : Byte;
                      uTimeout      : Word;
                      bSigMask, bSigVals : Byte ) : Integer;

Begin
  if uTimeout <> 0 then
    { Timeout loop }
    Begin
      while( not ser_IsDataAvailable( iSerPort )
            and ( uTimeout <> 0 ) )
      do Dec( uTimeout );
      if uTimeout = 0 then
        Begin
          ser_ReadByte := SER_ERRTIMEOUT;
          Exit;
        End;
      End
    else { Wait! }
    Repeat

```

```

Until ser_IsDataAvailable( iSerPort );

{-- Test signal lines -----}
if ( port[iSerPort+SER_MODEM_STATUS] and bSigMask ) = bSigVals then
  Begin
    { Read received byte from port }
    Data := port[iSerPort + SER_RXBUFFER];
    ser_ReadByte := port[iSerPort + SER_LINE_STATUS] and
      SER_LSR_ERRORMSK;
  End
else
  ser_ReadByte := SER_ERRSIGNALS;
End;

{*****}
{ ser_WritePacket : Send data packet }
{-----*}
{ Input : iSerPort - base port of interface through which }
{           data is being sent. }
{           pData   - address of data being sent }
{           iLen    - >= 0 : Number of bytes being sent. }
{                   < 0 : Buffer size = strlen( pData ) }
{           uTimeout - number of passes through loop }
{                   after which a timeout error occurs }
{                   if the send was unsuccessful. (If }
{                   iTimeout = 0 the system waits forever.) }
{           bSigMask - bit mask of signal lines being tested }
{                   (RTS, CTS, CD, RI) }
{           bSigVals - signal line status after }
{                   applying above mask. }
{ Output : = 0 - byte was sent }
{         <> 0 - error }
{*****}

Function ser_WritePacket( iSerPort : Integer;
  pData      : pointer;
  iLen       : Integer;
  uTimeout   : Word;
  bSigMask,
  bSigVals   : Byte ) : Integer;

var i, e      : Integer;
    BufPtr : SerBufPtr;

Begin
  BufPtr := pData;
  if iLen < 0 then { If length not given then }
    Begin
      { search for first ZERO-byte in packet }
      iLen := 0;
      while BufPtr^[iLen] <> 0 do
        Inc(iLen);
      End;
    End;

  {-- Go through packet and send each byte individually -----}
  for i := 0 to iLen - 1 do
    Begin
      e := ser_WriteByte( iSerPort, BufPtr^[i], uTimeout,
        bSigMask, bSigVals );

      if e <> 0 then
        Begin
          ser_WritePacket := e;
          Exit;
        End;
      End;
    End;
  ser_WritePacket := SER_SUCCESS;
End;

{*****}
{ ser_ReadPacket : Receive data packet }
{-----*}
{ Input : iSerPort - base port of interface through which }
{           data is being received. }
{           pData   - address of data being sent }
{           iLen    - size of receive buffer }
{           uTimeout - number of passes through loop }
{                   after which a timeout error occurs }
{                   if the send was unsuccessful. (If iTimeout }
{                   = 0 the system waits forever.) }
{           bSigMask - bit mask of signal lines being tested }
{                   (RTS, CTS, CD, RI) }
{           bSigVals - signal line status after }
{                   applying above mask. }
{ Output : = 0 - byte was sent }
{         <> 0 - error }
{*****}

Function ser_ReadPacket( iSerPort : Integer;
  pData      : pointer;
  iLen       : Integer;
  uTimeout   : Word;

```



```

Begin
{-- No more IRQs to IRQ controller -----}
{-- Set handler and clear all "enablers" }

ser_CLRIRQ( iSerPort );
ser_SetIRQHandler( iSerPort, iSerIRQ, lpHandler, 0 );
irq_Disable( iSerIRQ );      { Also disable IRQs by the controller }
End;

{*****}
{ ser_PrintError : Output error message }
{-----*}
{ Input : Win - window where output should appear }
{          e - error code }
{*****}
Procedure ser_PrintError( var Win : WINDOW; e : Integer );

Begin
  case e of
    SER_LSR_DATA RECEIVED:
      win_print( Win, 'Old data!'#10 );

    SER_ERRTIMEOUT:
      win_print( Win, 'Timeout error!' );

    SER_ERRSIGNALS:
      win_print( Win, 'Signal lines!' );

  else
    begin
      if ( e and SER_LSR_OVERRUNERROR ) <> 0 then
        win_print( Win, 'Overrun error!'#10 );

      if ( e and SER_LSR_PARITYERROR ) <> 0 then
        win_print( Win, 'Parity error!'#10 );

      if ( e and SER_LSR_FRAMINGERROR ) <> 0 then
        win_print( Win, 'Framing error!'#10 );

      if ( e and SER_LSR_BREAKDETECT ) <> 0 then
        win_print( Win, 'Break detect!'#10 );

    end;
  end;
End;

{*****}
{ ser_PrintModemStatus : Display status of signal lines }
{-----*}
{ Input : Win - window where output should appear }
{          iSerPort - base port of interface whose }
{                   line statuses are being displayed. }
{*****}
Procedure ser_PrintModemStatus( var Win : WINDOW; iSerPort:Integer );

var b : Byte;

Begin
  b := port[iSerPort + SER_MODEM_STATUS];
  if ( b and SER_MSR_DCTS ) <> 0 then
    win_print( Win, 'DCTS : ' )
  else
    win_print( Win, ' CTS : ' );

  if ( b and SER_MSR_CTS ) <> 0 then
    win_print( Win, '[X]'#10 )
  else
    win_print( Win, '[ ]'#10 );

  if ( b and SER_MSR_DDSDR ) <> 0 then
    win_print( Win, 'DDSDR : ' )
  else
    win_print( Win, ' DSR : ' );

  if ( b and SER_MSR_DSR ) <> 0 then
    win_print( Win, '[X]'#10 )
  else
    win_print( Win, '[ ]'#10 );

  if ( b and SER_MSR_DRI ) <> 0 then
    win_print( Win, 'DRI : ' )
  else
    win_print( Win, ' RI : ' );

  if ( b and SER_MSR_RI ) <> 0 then
    win_print( Win, '[X]'#10 )
  else

```

```

win_print( Win, '[' ]'#10 );

if ( b and SER_MSR_DCD ) <> 0 then
    win_print( Win, 'DCD  : ' )
else
    win_print( Win, ' CD  : ' );

if ( b and SER_MSR_CD ) <> 0 then
    win_print( Win, '[X]'#10 )
else
    win_print( Win, '[ ]'#10 );
End;

{*****}
{ ser_PrintLineStatus : Display port status }
{*****}
{-----*}
{ Input : Win          - window where output should appear }
{           iSerPort - base port of interface whose }
{                   internal statuses are being displayed. }
{*****}
Procedure ser_PrintLineStatus( var Win : WINDOW; iSerPort : Integer );

var b : Byte;

Begin
    b := port[iSerPort + SER_MODEM_STATUS];
    if ( b and SER_LSR_DATARECEIVED ) <> 0 then
        win_print( Win, 'Data received [X]'#10 )
    else
        win_print( Win, 'Data received [ ]' );

    if ( b and SER_LSR_OVERRUNERROR ) <> 0 then
        win_print( Win, 'Overrun error [X]'#10 )
    else
        win_print( Win, 'Overrun error [ ]'#10 );

    if ( b and SER_LSR_PARITYERROR ) <> 0 then
        win_print( Win, 'Parity error  [X]'#10 )
    else
        win_print( Win, 'Parity error  [ ]'#10 );

    if ( b and SER_LSR_FRAMINGERROR ) <> 0 then
        win_print( Win, 'Framing error [X]'#10 )
    else
        win_print( Win, 'Framing error [ ]'#10 );

    if ( b and SER_LSR_BREAKDETECT ) <> 0 then
        win_print( Win, 'Break detect  [X]'#10 )
    else
        win_print( Win, 'Break detect  [ ]'#10 );

    if ( b and SER_LSR_THREMPY ) <> 0 then
        win_print( Win, 'THR empty    [X]'#10 )
    else
        win_print( Win, 'THR empty    [ ]'#10 );

    if ( b and SER_LSR_TSREMPY ) <> 0 then
        win_print( Win, 'TSR empty    [X]'#10 )
    else
        win_print( Win, 'TSR empty    [ ]'#10 );
End;

{*****}
{ ser_GetBaud : Get current baud rate for port }
{*****}
{-----*}
{ Input : iSerPort - Base address of port whose }
{                   baud rate is being determined. }
{ Output : baud rate }
{*****}
Function ser_GetBaud( iSerPort : Integer ) : Longint;

var uDivisor : Word;
    bSettings : Byte;

Begin
    asm cli End; { Disable interrupts }
    bSettings := port[iSerPort + SER_LINE_CONTROL];

    {-- Read baud rate divisor -----}
    port[iSerPort + SER_LINE_CONTROL] := bSettings or SER_LCR_SETDIVISOR;
    uDivisor := port[iSerPort + SER_DIVISOR_MSB] * 256 +
                port[iSerPort + SER_DIVISOR_LSB];

    port[iSerPort + SER_LINE_CONTROL] := bSettings;
    asm sti End; { Re-enable interrupts }
    if uDivisor <> 0 then

```

```

ser_GetBaud:= SER_MAXBAUD div uDivisor
else
    ser_GetBaud := 0;
End;

{*****}
{ ser_PrintCardSettings : Display transmission parameters }
{ for port }
{*****}
{-----*}
{ Input : Win          - window where output should appear }
{           iSerPort   - base port of interface whose }
{                   transmission parameters are being displayed. }
{*****}
Procedure ser_PrintCardSettings( var Win : WINDOW; iSerPort : Integer );

var lBaudRate : Longint;
    bSettings : Byte;
    WordLen   : Integer;
    s         : string;

Begin
    case ser_UARTType( iSerPort ) of
        NOSER:
            Begin
                win_print (Win, ' No port detected!'#10 );
                Exit;
            End;

        INS8250:
            win_print( Win, ' INS8250 UART chip'#10 );

        NS16450:
            win_print( Win, ' NS16450 UART chip'#10 );

        NS16550A:
            win_print( Win, ' NS16550A UART chip'#10 );

        NS16C552:
            win_print( Win, ' NS16C552 UART chip'#10 );
    End;

    lBaudRate := ser_GetBaud( iSerPort );
    str( lBaudRate, s );
    win_print( Win, ' Baud rate '+s+#10 );

    bSettings := port[iSerPort + SER_LINE_CONTROL];
    WordLen := 5 + ( bSettings and SER_LCR_WordLEN );
    str( WordLen, s );
    win_print( Win, ' Data bits : '+s+#10 );

    if ( bSettings and SER_LCR_2STOPBITS ) <> 0 then
        if WordLen = 5 then
            s := '1.5'
        else
            s := '2'
        else
            s := '1';

    win_print( Win, ' Stop bits : '+s+#10 );
    win_print( Win, ' Parity : ' );

    case bSettings and SER_LCR_PARITYMSK of
        SER_LCR_ODDPARITY:
            win_print( Win, 'odd' );
        SER_LCR_EVENPARITY:
            win_print( Win, 'even' );
        SER_LCR_PARITYSET:
            win_print( Win, 'always set' );
        SER_LCR_PARITYCLR:
            win_print( Win, 'always cleared' );
        else
            win_print( Win, 'none' );
    End;

    win_print( Win, #10 );
    if ( bSettings and SER_LCR_SENDBREAK ) <> 0 then
        win_print(Win, 'Send break signal'#10);
End;

End.

```