

Pascal listing: PLINKP.PAS

```
{*****}
{                                     P L I N K P . P A S                                     }
{-----}
{ Task           : Transfers files over the parallel interface. }
{-----}
{ Author        : Michael Tischer }
{ Developed on   : 09/27/91 }
{ Last update    : 04/07/95 }
{*****}
{$M 65520, 0, 655360 }
```

```
uses dos, crt;                                { Add DOS and CRT units }
```

```
{== Constants =====}
```

```
const ONESEC      = 18;                        { One second }
    TENSEC        = 182;                      { Ten seconds }
    TO_DEFAULT    = TENSEC;                  { Time out default value }

    MAXBLOCK      = 4096;                     { 4K maximum block size }
```

```
{-- Constants for transfer protocol -----}
```

```
    ACK           = $00;                      { Acknowledge }
    NAK           = $FF;                      { Non-Acknowledge }
    MAX_TRY       = 5;                        { Maximum number of tries }
```

```
{-- Tokens for communication between sender and receiver -----}
```

```
    TOK_DATSTART  = 0;                        { Start of file data }
    TOK_DATNEXT   = 1;                        { Next file data block }
```

```

TOK_DATEND      = 2;                                { End file data transfer }
TOK_ENDIT       = 3;                                { End program }
TOK_ESCAPE      = 4;                                { <Esc> pressed on remote computer }

{-- Codes for LongJump call -----}

LJ_OKSENDER     = 1;                                { All data sent successfully }
LJ_OKRECD       = 2;                                { All data received successfully }
LJ_TIMEOUT      = 3;                                { Time out: No response }
LJ_ESCAPE       = 4;                                { <Esc> pressed on local computer }
LJ_REMESCAPE    = 5;                                { <Esc> pressed on remote computer }
LJ_DATA         = 6;                                { Communication error }
LJ_NOLINK       = 7;                                { No link }
LJ_NOPAR        = 8;                                { No interface }
LJ_PARA         = 9;                                { Invalid call parameters }

{== Type definitions =====}

type BHEADER = record                                { Block transfer header }
    case boolean of
        true  : ( Token : byte;
                  Len    : word );
        false : ( HArr  : array[ 0..2 ] of byte );
    end;

JMPBUF = record                                     { Information needed for }
    BP,                                             { the return is stored here }
    SP,
    CS,      { Do not change the order of these elements! }
    IP : word;
end;

```

```

        DBlock = array[ 1..MAXBLOCK ] of byte;                { Data block }

{== Global variables =====}

var InpPort      : word;                { Input port address }
    OutPort      : word;
    Escape       : boolean;            { <Esc> not pressed }
    Timeout      : word;                { Selected time out value }
    TO_Count     : word;                { Counter for time out }
    ReturnToEnd  : JMPBUF;              { Return address for end }
    BlockBuf     : DBLOCK;              { Buffer for passing a block }
    FiVar        : file;                { File variable for file being processed }

{== Declaration of assembler functions =====}

{$L plinkpa.obj }                { Link assembler module }

function getb : byte; external;
procedure putb( TpVar : byte ); external;
procedure intr_install(escape_flag, timeout_count : pointer); external;
procedure intr_remove; external;
procedure EscapeDirect( Disconnect : boolean ); external;

{*****}
{ SetJmp : Determines when a call to LongJump should follow. }
{-----}
{ Input   : JB : A data structure of type JumpBuf, needed for passing }
{           information after the return }
{ Output  : NOJMP on return from this function; any other value on }
{           return from a call to the LongJump procedure }
{*****}

```

```

{$F+}                                { Setjmp and Longjmp must be coded as FAR }

function SetJump( var JB : JMPBUF ) : integer;

type WordP = ^word;                  { Pointer to a word }

begin
  JB.BP := WordP( ptr( SSeg, Sptr+2) )^;
  JB.IP := WordP( ptr( SSeg, Sptr+4) )^;
  JB.CS := WordP( ptr( SSeg, Sptr+6) )^;

  { SP must point to the position for the new BP register contents    }
  { after the LongJump, as well as the stack's return address        }

  JB.SP := Sptr + 12 - 6 - 6;

  SetJump := -1;                     { Shows that this is no LongJump call }
end;

{ ***** }
{ LongJump : A procedure dependent GOTO that continues program        }
{ execution in the program line called by SetJump.                  }
{ ----- }
{ Input      : JB           : Jump buffer which redirects program    }
{                  execution through SetJump                        }
{ RetCode    : Function result which indicates SetJump target      }
{ ***** }

procedure LongJump( JB : JMPBUF; RetCode : integer );

type WordP = ^word;                  { Pointer to a word }

```

```

begin
  WordP( ptr( SSeg, JB.SP ) )^ := JB.BP;
  WordP( ptr( SSeg, JB.SP+2 ) )^ := JB.IP;
  WordP( ptr( SSeg, JB.Sp+4 ) )^ := JB.CS;

  {-- Load return code into the AX register and emulate -----}
  {-- SetJmp's function result -----}

  inline( $8b / $46 / $06 );           { mov ax,[bp+6] }

  inline( $8b / $ae / $fa / $ff );     { mov bp,[bp-6] }

  { mov    sp,bp           ;This command invokes the compiler      }
  { pop    bp              ;automatically and restores the         }
  { ret    6               ;stack                                   }
end;

{$F-}

{ ***** }
{ GetPortAdr: Initializes a parallel interface's port addresses in }
{               the global variables INPPORT and OUTPORT.          }
{ Input         : LPTNUM = Parallel interface number (1-4)        }
{ Output        : TRUE if interface is valid                      }
{ Global vars.  : InpPort/W, OutPort/W                             }
{ Info         : The base addresses of up to 4 parallel interfaces }
{               lie in the four memory words starting at 0040:0008. }
{ ***** }

function GetPortAdr( LptNum : integer ) : boolean;

begin

```

```

{-- Read port addresses from BIOS variable segment -----}
OutPort := MemW[ $0040: 6 + LptNum * 2 ];
if ( OutPort <> 0 ) then                                { Interface available? }
begin                                                  { Yes }
    InpPort := OutPort + 1;                            { Input register address }
    GetPortAdr := TRUE;                                { End error-free }
end
else
    GetPortAdr := FALSE;                                { Error: Interface not available }
end;

```

```

{*****}
{ Port_Init      : Initializes the registers needed for transfer. }
{ Input          : SENDER = TRUE if sender, FALSE if receiver }
{ Output         : TRUE if register initializes successfully }
{ Global vars.   : InpPort/R, OutPort/R }
{ Info           : The asymmetry (send 00010000, wait for 00000000) }
{                 occurs because of signal inversion. Normally the }
{                 input and output registers contain the desired }
{                 values, but initialization is needed after an }
{                 aborted transfer, or when restarting. }
{*****}

```

```

function Port_Init( Sender : boolean ) : boolean;

```

```

begin
    EscapeDirect( TRUE );                                { Release thru Escape time out }
    if ( Sender ) then                                    { Device = Sender? }
    begin
        TO_Count := Timeout * 5;                          { Start time out counter }
        PutB( $10 );                                       { Send: 00010000b }
        while ( ( GetB <> $00 ) and ( TO_Count > 0 ) ) do { Wait for 0 }

```

```

        ;
    end
else
    { Device = Receiver }
begin
    TO_Count := Timeout * 5;           { Start time out counter }
    while ( ( GetB <> $00 ) and ( TO_Count > 0 ) ) do { Wait for 0 }
        ;
        PutB( $10 );                   { Send: 00010000b }
    end;
    EscapeDirect( FALSE );             { No time out released on Escape }
    Port_Init := ( TO_Count > 0 );     { End initialization }
end;

```

```

{*****}
{ SendABYTE      : Sends a byte to the remote computer in two parts, }
{                  then checks the result.                             }
{ Input          : B2Send = Byte to be sent                           }
{ Output         : Transfer successful? (0 = error, -1 = O.K.)         }
{ Global vars.   : Timeout/R, InpPort/R, OutPort/R (in macros)         }
{*****}

```

```
function SendABYTE( B2Send : byte ) : boolean;
```

```
var RcvdB : byte;           { Received byte }
```

```

begin
    {-- Send lower nibble -----}

    TO_Count := Timeout;           { Initialize time out counter }
    PutB( B2Send and $0F );        { Sending, clear BUSY }
    while ( ( ( GetB and 128 ) = 0 ) and ( TO_Count > 0 ) ) do
        ;
    end;

```

```

if ( TO_Count = 0 ) then                                { Time out error? }
    longjmp( ReturnToEnd, LJ_TIMEOUT );                 { Cancel transfer }

RcvdB := ( GetB shr 3 ) and $0F;                         { Bits 3-6 in 0-3 }

{-- Send upper nibble -----}

TO_Count := Timeout;                                     { Initialize time out counter }
PutB( ( B2Send shr 4 ) or $10 );                         { Sending, set BUSY }
while ( ( ( GetB and 128 ) <> 0 ) and ( TO_Count > 0 ) ) do

    if ( TO_Count = 0 ) then                             { Time out error? }
        longjmp( ReturnToEnd, LJ_TIMEOUT );             { Cancel transfer }

    RcvdB := RcvdB or ( ( GetB shl 1 ) and $F0 );        { Bits 3-6 in 4-7 }
    SendAByte := ( B2Send = RcvdB );                    { Byte sent correctly? }
end;

```

```

{*****}
{ ReceiveAByte : Receives a two part byte from remote computer, and }
{               sends returned parts for testing.                     }
{ Input        : None                                                }
{ Output       : Received byte                                        }
{ Global vars. : Timeout/R, InpPort/R, Outport/R (in macros)        }
{*****}

```

```
function ReceiveAByte : byte;
```

```

var LoNib,
    HiNib : byte;                                     { Received nibbles }

```

```
begin
```



```

{-- Receive and re-send lowest nibble -----}

TO_Count := Timeout;           { Initialize time out counter }
while ( ( ( GetB and 128 ) = 0 ) and ( TO_Count > 0 ) ) do
;

if ( TO_Count = 0 ) then           { Time out error? }
    longjmp( ReturnToEnd, LJ_TIMEOUT );      { Cancel transfer }

LoNib := ( GetB shr 3 ) and $0F;      { Bits 3-6 in 0-3 }
PutB( LoNib );                        { Re-send }

{-- Receive and re-send highest nibble -----}

TO_Count := Timeout;           { Initialize time out counter }
while ( ( ( GetB and 128 ) <> 0 ) and ( TO_Count > 0 ) ) do
;

if ( TO_Count = 0 ) then           { Time out error? }
    longjmp( ReturnToEnd, LJ_TIMEOUT );      { Cancel transfer }

HiNib := ( GetB shl 1 ) and $F0;      { Bits 3-6 in 4-7 }
PutB( ( HiNib shr 4 ) or $10 );      { Re-sending, set BUSY }

ReceiveAByte := ( LoNib or HiNib );      { Received byte }
end;

{ ***** }
{ SendABlock: Sends a data block }
{ Input      : TOKEN   = Token for receiver }
{             TRANUM  = Number of bytes to be transferred }
{             DPTR    = Pointer to buffer containing data }

```

```
{ Output      : None, jumps immediately to an error handler if
{               an error occurs.
{ ***** }
```

```
procedure SendABlock( Token  : byte;
                     TraNum : word;
                     Dptr   : pointer );
```

```
var header      : BHEADER;      { Header for placing tokens and numbers }
    RcvrEscape  : byte;         { <Esc> pressed on remote? }
    ok          : boolean;      { Error flag }
    i           : word;         { Loop counter }
    try         : word;         { Remaining number of tries }
    DbPtr       : ^DBlock;      { Pointer to data block }
```

```
begin
  if ( Escape ) then           { <Esc> pressed on local computer? }
  begin
    Token := TOK_ESCAPE;      { Yes --> Send Escape token }
    TraNum := 0;
  end;
```

```
{-- Send header first -----}
```

```
header.Token := Token;        { Create header }
header.Len := TraNum;
```

```
try := MAX_TRY;
repeat
  ok := TRUE;                { Make MAX_TRY attempts at access }
  for i := 0 to 2 do         { Send on error-free transfer }
    ok := ok and SendAByte( Header.HArr[ i ] );      { Send a byte }
```

```

if ( ok ) then
    ok := ok and SendABYTE( ACK )                { Send confirmation }
else
    ok := ok and SendABYTE( NAK );                { Send confirmation }
    if ( not ok ) then                            { Error? }
        dec( try );                               { Yes --> Try it again }
until ( ( ok ) or ( try = 0 ) );

if ( try = 0 ) then                               { Could the header be sent successfully? }
    longjmp( ReturnToEnd, LJ_DATA );              { No --> Cancel transfer }

if ( Token = TOK_ESCAPE ) then                   { Was an Escape message sent? }
    longjmp( ReturnToEnd, LJ_ESCAPE );            { Yes --> Cancel transfer }

{-- Send the data block itself -----}

if ( TraNum > 0 ) then                            { Length > 0? }
    begin
        DbPtr := DPTR;
        try := MAX_TRY;
        repeat
            ok := TRUE;                            { Send on error-free transfer }
            for i := 1 to TraNum do
                ok := ok and SendABYTE( DbPtr^[ i ] );
            if ( ok ) then
                ok := ok and SendABYTE( ACK )        { Send confirmation }
            else
                ok := ok and SendABYTE( NAK );        { Send confirmation }
            if ( not ok ) then                        { Error? }
                dec( try );                           { Yes --> Try again }
        until ( ( ok ) or ( try = 0 ) );
        if ( try = 0 ) then                          { Could data block be sent successfully? }

```

```

        longjmp( ReturnToEnd, LJ_DATA );      { No --> Cancel transfer }
    end;

    {-- Read ESCAPE byte from receiver -----}

    try := MAX_TRY;
    repeat
        RcvrEscape := ReceiveAByte;           { Read remote Escape }
        dec( try );
    until ( ( RcvrEscape = byte( true ) ) or
            ( RcvrEscape = byte( false ) ) );

    if ( try = 0 ) then                        { Was the Escape status received? }
        longjmp( ReturnToEnd, LJ_DATA );      { No --> Cancel transfer }

    if ( RcvrEscape = byte( true ) ) then { <Esc> from remote computer? }
        longjmp( ReturnToEnd, LJ_REMESCAPE ); { Yes --> Cancel transfer }
    end;

    {*****}
    { ReceiveABlock: Receives a data block. }
    { Input      : TOKEN  = Pointer to variable containing tokens }
    {              LEN    = Pointer to variable containing length }
    {              DPTR   = Pointer to buffer containing data }
    { Output     : None, jumps immediately to an error handler if }
    {               an error occurs. }
    { Info       : Buffer must be allocated using MAXBLOCK, since block }
    {                 length cannot usually be anticipated. }
    {*****}

    procedure ReceiveABlock( var Token : byte;
                             var Len   : word;

```

```

                                Dptr  : pointer );

var header      : BHEADER;    { Header for storing tokens and numbers }
    ok          : boolean;    { Error flag }
    i           : word;       { Loop counter }
    try         : word;       { Remaining number of tries }
    EscapeStatus : boolean;
    ByteBuffer   : byte;
    DbPtr        : ^DBlock;    { Pointer to data block }

begin
  {-- Receive header first -----}

  try := MAX_TRY;
  repeat
    for i:= 0 to 2 do
      Header.HArr[ i ] := ReceiveAByte;

    ByteBuffer := ReceiveAByte;
    if ( ByteBuffer <> ACK ) then { All bytes received successfully? }
      dec( try );                { Yes --> No need to try again }
      until ( ( try = 0 ) or ( ByteBuffer = ACK ) );

  if ( try = 0 ) then            { Header received successfully? }
    longjmp( ReturnToEnd, LJ_DATA );    { No --> Cancel transfer }

  Token := Header.Token;
  Len := Header.Len;
  if ( Token = TOK_ESCAPE ) then { Sender Escape? }
    longjmp( ReturnToEnd, LJ_REMESCAPE ); { Yes --> Cancel transfer }

  {-- Header was O.K., now receive the data block itself -----}

```

```

if ( Len > 0 ) then                                { No data block? }
begin                                              { No }
    DbPtr := Dptr;
    try := MAX_TRY;

    repeat                                         { Receive data block byte for byte }
        for i := 1 to len do
            DbPtr^[ i ] := ReceiveAByte;

            ByteBuffer := ReceiveAByte;
            if ( ByteBuffer <> ACK ) then { Bytes received successfully? }
                dec( try );              { Yes --> No need to try again }
            until ( ( try = 0 ) or ( ByteBuffer = ACK ) );

            if ( try = 0 ) then                  { Block received successfully? }
                longjmp( ReturnToEnd, LJ_DATA );      { No --> Cancel transfer }
        end;

{-- Send current Escape status to the remote computer -----}

EscapeStatus := Escape;                          { Note status }

try := MAX_TRY;
repeat
    dec( try );
until ( SendAByte( byte( EscapeStatus ) ) or ( try = 0 ) );

if ( try = 0 ) then                                { Could Escape status be sent? }
    longjmp( ReturnToEnd, LJ_DATA );              { No --> Cancel transfer }

if ( EscapeStatus ) then                          { <Esc> pressed on this computer? }

```

```

    longjmp( ReturnToEnd, LJ_ESCAPE );          { Yes --> Cancel transfer }
end;

{ ***** }
{ SendAFile : Sends a file. }
{ Input      : NAME = Filename }
{ Output     : None }
{ ***** }

procedure SendAFile( Name : string );

var Status      : word;                      { Send status }
    WdsRead     : word;                      { Number of bytes read }
    BytSent     : longint;                   { Number of bytes sent }

begin
    write( copy( Name + '                      ', 1, 13 ) );
    assign( FiVar, Name );
    reset( FiVar, 1 );
    SendABlock( TOK_DATSTART, length( Name ) + 1, @Name ); { Send names }

    {-- Transfer file contents -----}

    BytSent := 0;
    repeat
        blockread( FiVar, BlockBuf, MAXBLOCK, WdsRead );    { Read block }
        if ( WdsRead > 0 ) then                                { End of block not reached? }
            begin {                                           { No }
                SendABlock( TOK_DATNEXT, WdsRead, @BlockBuf ); { Send block }
                inc( BytSent, WdsRead );
                write( #13, copy( Name + '                      ', 1, 13 ),
                    '(' , BytSent, ')' );
            end
        until WdsRead = 0;
    end repeat;
end;

```

```

        end;
until ( WdsRead = 0 );
writeln;

SendABlock( TOK_DATEND, 0, NIL );                { End transfer }

close( FiVar );                                  { Close file }
end;

{ ***** }
{ ReceiveAFile: Receives a file.                  }
{ Input      : None                               }
{ Output     : The last token received            }
{ ***** }

function ReceiveAFile : word;

var Status      : word;                          { Send status }
    LastBlkSize : word;                          { Size of last block }
    BytSent     : longint;
    Token       : byte;                          { Token received }
    Len         : word;                          { Block length received }
    i           : word;                          { Loop counter }
    Name        : string[ 13 ];                  { Filename }

begin
    ReceiveABlock( Token, Len, @BlockBuf );
    if ( Token = TOK_DATSTART ) then
        begin
            for i := 0 to BlockBuf[ 1 ] do
                Name[ i ] := chr( BlockBuf[ i + 1 ] );
            assign( FiVar, Name );
        end
    end
end

```



```

rewrite( FiVar, 1 );
write( copy( Name + '                                ', 1, 13 ) );

{-- Receive file contents -----}

BytSent := 0;
repeat
  ReceiveABlock( Token, Len, @BlockBuf );          { Receive block }
  if ( Token = TOK_DATNEXT ) then                  { Next data block? }
  begin                                             { Yes }
    blockwrite( FiVar, BlockBuf, Len );            { Write block }
    inc( BytSent, Len );
    write( #13, copy( Name + '                                ', 1, 13 ),
           '(', BytSent, ')' );
  end;
until ( TOKEN <> TOK_DATNEXT );
close( FiVar );                                   { Close file }
writeln;

end;
ReceiveAFile := Token;                           { Return error status }
end;

{*****}
{
  M A I N   P R O G R A M
}
{*****}

const ScnMesg : array[ 0..8 ] of string =
  ( 'DONE: All files sent successfully.',
    'DONE: All files received successfully.',
    'ERROR: Time out, remote system not responding.',
    'DONE: <Esc> key pressed.',
    'DONE: <Esc> key pressed on remote computer.',

```

```

        'ERROR: Check hardware (cable, interface, jacks).',
        'ERROR: No contact with remote computer.',
        'ERROR: Interface you requested not found.',
        'ERROR: Invalid or unknown parameters.' );

var SRec      : SearchRec;           { Structure for directory search }
Sender       : boolean;              { Transfer mode (Send, Receive) }
sjStatus     : integer;              { Longjump code }
LptNum,      :                    { Interface number }
i,           :                    { Loop counter }
DirFound     : byte;                { For directory search }
dummy        : integer;
argv         : array[ 1..10 ] of string;      { Parameter }

begin
  write( #13#10'Parallel Interface File Transfer Program' );
  writeln( '      (c) 1992 by Michael Tischer' );
  write( '===== ' );
  writeln( '===== ' );

  Escape := false;
  Timeout := TO_DEFAULT;

  if ( paramstr( 1 ) = '?' ) then      { Display syntax only? }
    begin                             { Yes }
      writeln( 'Syntax: plinkp [/Pn] [/Tnn] [filename]' );
      writeln( '      |           | ' );
      writeln( '      Interface   Time out ' );
      halt( 0 );
    end;

```

```

sjStatus := setjmp( ReturnToEnd );           { Return address to end }
if ( sjStatus > 0 ) then                       { Longjmp called? }
begin                                         { Yes }
    Intr_remove;                           { Disable interrupt handler }
    writeln( #13#10#13#10, ScnMesg[ sjStatus - 1 ] );
    halt( 0 );
end;

Intr_Install( @Escape, @TO_Count );           { Interrupt handler init }

{-- Set default values and interpret command line -----}

Sender := FALSE;                             { Default is Receiver }
LptNum := 1;                                  { Default is LPT1 }

for i := 1 to paramcount do
begin
    argv[ i ] := paramstr( i );               { Note parameters }
    if ( argv[ i, 1 ] = '/' ) then            { Parameter }
    begin
        case ( upcase( argv[ i, 2 ] ) ) of
            'T' : begin
                delete( argv[ i ], 1, 2 );
                val( argv[ i ], Timeout, dummy );
                Timeout := ( Timeout * TENSEC ) div 10;
                if ( Timeout = 0 ) then
                    longjmp( ReturnToEnd, LJ_PARA );           { Invalid }
                end;
            'P' : begin
                LptNum := ord( argv[ i, 3 ] ) - 48;           { Interface }
                if ( ( LptNum = 0 ) or ( LptNum > 4 ) ) then
                    longjmp( ReturnToEnd, LJ_PARA );           { Invalid }
            end;
        end;
    end;
end;

```

```

        end;
        else longjmp( ReturnToEnd, LJ_PARA ); { Unknown parameters }
    end;
    argv[ i ] := ''; { Clear argument }
end
else { No parameters, must be a filename }
    Sender := TRUE; { Sender }
end;

{-- Start transfer -----}

if ( not GetPortAdr( LptNum ) ) then { Does the interface exist? }
    longjmp( ReturnToEnd, LJ_NOPAR ); { No --> Error }

if ( not Port_Init( Sender ) ) then { Create link }
    longjmp( ReturnToEnd, LJ_NOLINK ); { Impossible, error }

if ( Sender ) then { Sender? }
    begin
        writeln( 'Sending over LPT', LptNum, #13#10 );

        {-- Transfer all data -----}

        for i := 1 to paramcount do { Execute command line }
            begin
                if ( argv[ i ] <> '' ) then { Filename? }
                    begin { Yes }
                        findfirst( argv[ i ], AnyFile, SRec);
                        while ( DosError = 0 ) do { While found }
                            begin
                                if ( SRec.Attr <> Directory ) then
                                    SendAFile( SRec.Name ); { Transfer file }
                            end
                        end
                    end
                end
            end
        end
    end
end

```

```

                findnext( SRec );
            end;
        end;
        SendABlock( TOK_ENDIT, 0 , NIL );           { All files sent? }
        longjmp( ReturnToEnd, LJ_OKSENDER );
    end
else                                               { No --> Receiver }
begin
    writeln( 'Receiving over LPT', LptNum, #13#10 );
    while ( ReceiveAFile <> TOK_ENDIT ) do        { Receive data      }
        ;                                         { until END token  }
    longjmp( ReturnToEnd, LJ_OKRECD );
    end;
end.

```