

```

***** D S P U T I L . P A S *****
{
-----*
Task      : Provides functions for programming the
            Sound Blaster DSP.
-----*
Author     : Michael Tischer / Bruno Jennrich
Developed on  : 03/20/1994
Last update : 04/05/1995
-----*
Note: If this module is compiled with the defined symbol
      DSP_VERSIONONLY only those functions required for
      determining the version will be compiled!
*****}

{$X+} { extended syntax, evaluation of function results optional }

Unit DSPUTIL;

Interface
Uses SBUTIL, DOS;

Type

DSPRECPLAY = record
    iStereo      : Boolean;
    iBits        : Integer;
    uFrequency    : Word;
End;

UserFunc      = Procedure( l : Longint ); { Callback routines }
DSP4UserFunc  = Procedure( b : Byte );
OldIntproc    = Procedure;

const
    DSP_ERRRESET      = 1;           { Error codes }
    DSP_ERRVERSION    = 2;
    DSP_ERRFRQ        = 3;
    DSP_ERR4DACFRQ    = 4;
    DSP_ERR4ADCFRQ    = 5;
    DSP_ERRSPEAKER    = 6;
    DSP_ERRTRANSSIZE  = 7;
    DSP_ERRILLSIZE    = 8;
    DSP_ERR8DAC       = 9;
    DSP_ERR8ADC       = 10;
    DSP_ERRPLAY       = 11;
    DSP_ERRRECORD     = 12;

    FRQ1_MONO_ADC     = 13000;      { Highest AD sampling frequencies }
    FRQ2p_MONO_ADC    = 13000;      { DSP Version 2.01+ }
    FRQ2p_HIMONO_ADC  = 15000;
    FRQ3_MONO_ADC     = 23000;
    FRQ3_HIMONO_ADC   = 44100;
    FRQ3_STEREO_ADC   = 22050;
    FRQ4_ADC          = 44100;

    FRQ1_MONO_DAC     = 23000;      { Highest DA sampling frequencies }
    FRQ2p_MONO_DAC    = 23000;      { DSP Version 2.01+ }
    FRQ2p_HIMONO_DAC  = 44100;
    FRQ3_MONO_DAC     = 23000;
    FRQ3_HIMONO_DAC   = 44100;
    FRQ3_STEREO_DAC   = 22050;
    FRQ4_DAC          = 44100;

    DSP_1XX = $100;
    DSP_200 = $200;
    DSP_201 = $210;
    DSP_3XX = $300;
    DSP_4XX = $400;           { Version numbers }

    { DSP register offsets }
    DSP_WRITESTATUS = $0C;      { Bit 7 indicates whether write operations are allowed }
    DSP_WRITECMDATA = $0C;      { Write data }

    DSP_READSTATUS  = $0E;      { Bit 7 indicates whether read operations are allowed }
    DSP_IRQACK      = $0E;      { Interrupt acknowledge }
    DSP_READDATA    = $0A;      { Read data }

    DSP_RSET        = $06;      { Reset port }

    DSP_GETVER      = $E1;      { DSP commands }
    DSP_8DA         = $10;
    DSP_8DMADAC     = $14;
    DSP_8DMAAUTODAC = $1C;

    DSP_8AD         = $20;

```

```

_DSP_8DMAADC      =    $24;
_DSP_8DMAAUTOADC  =    $2C;

_DSP_SETTIMECONSTANT = $40;
_DSP_SETTRANSSIZE   = $48;

_DSP_DMAPAUSE      = $D0;
_DSP_DMACONTINUE   = $D4;

_DSP2p_8DMAHIAUTODAC = $90;
_DSP2p_8DMAHIDAC    = $91;

_DSP2p_8DMAHIAUTOADC = $98;
_DSP2p_8DMAHIADC     = $99;

_DSP_SPEAKERON      = $D1;
_DSP_SPEAKEROFF     = $D3;
_DSP2p_SPEAKERSTATUS = $D8;
_DSP4_EXIT16DMA     = $D9;
_DSP4_EXIT8DMA      = $DA;
_DSP2p_EXITAUTOINIT = $DA;

_DSP3_MONOADC       = $A0;                                { DSP3 }
_DSP3_STEREOADC     = $A8;

_DSP4_DACFR         = $41;
_DSP4_ADCFR         = $42;

_DSP4_CMDADC        = $08;                                { Command bit for recording }
_DSP4_CMDAUTOINIT   = $04;                                { Bit for Autoinit mode }
_DSP4_CMDFIFO       = $02;
_DSP4_CMDDAC        = $00;
_DSP4_CMD8DMA       = $C0;                                { Hi-Nibble for 8 and 16 bit }
_DSP4_CMD16DMA      = $B0;                                { commands }

_DSP4_MODESTEREO    = $20;
_DSP4_MODEMONO      = $00;
_DSP4_MODESIGNED    = $10;
_DSP4_MODEUNSIGNED  = $00;

_SB_NONE = 0;
_SB_1XX  = 1;                                { DSP Version 1.00 to 1.99 }
_SB_2XX  = 2;                                { DSP Version 2.00 to 2.99 }
_SB_3XX  = 3;                                { DSP Version 3.00 to 3.99 }
_SB_4XX  = 4;                                { DSP Version 4.00 to 4.99 }
_SB_XXX  = 5;                                { DSP Version > 4.99 }

ON        = TRUE;
OFF       = FALSE;

Procedure DummyUserFunc( l : Longint );
Procedure DummyDSP4UserFunc( b : Byte );

Function _dos_open( FileName: String ) : Integer;

Procedure _dos_close( handle : integer );

Function _dos_read(handle : integer; Buf : Pointer; cnt : Integer) : word;

Function _dos_write(handle:integer; Buf : Pointer; cnt : Integer) : word;

Function _dos_filesize( handle : integer ) : longint;

Function dsp_SetBase( var SBBASE : SBBASE;
                     iReset : Boolean ) : Word;

Function dsp_Write( iVal : Word ) : Word;

Function dsp_Read( var Val : Byte) : Word;

Function dsp_Reset : Word;

Function dsp_GetVersion(var SBBASE : SBBASE) : Word;

Function dsp_AdjustFrq( var Frq      : Word;
                       iADC      : Boolean;
                       var Stereo : Boolean ) : Boolean;

Function dsp4_DACFrq( uFrq : Word ) : Word;

Function dsp4_ADCFrq( uFrq : Word ) : Word;

Function dsp_SetFrq( var Frq : Word ) : Word;

Function dsp_CanStereo : Boolean;

```

```

Function dsp_IsHIMONODACFrq( uFrq : Word ) : Boolean;

Function dsp_MaxBits : Integer;

Function dsp_SetSpeaker( iState : Boolean ) : Word;

Function dsp_SetTransferSize( uSize : Word ) : Word;

Procedure dsp_IrqHandler; interrupt;

Procedure dsp_RestoreIrqHandler;

Procedure dsp_InitIrqHandler;

Procedure dsp_SetUserIRQ( lpFunc : UserFunc );

Procedure dsp4_SetUserIRQ( lpFunc : DSP4UserFunc );

Function dsp_WaitForNextIRQ( lpDoSomething : userfunc;
                             lpar          : Longint ) : Boolean;

Procedure dsp_InitWaitForIRQ;

Function dsp_8DAC( bVal : Byte ) : Word;

Function dsp_8ADC( var Val : Byte ) : Word;

Function dsp_PLAY( pBuffer      : pointer;
                  uSize, uDelay : Word ) : Word;

Function dsp_RECORD( pBuffer      : pointer;
                   uSize, uDelay : Word ) : Word;

Procedure dsp_InitBuffers;

Function dsp_FileOpen( Filename  : string;
                     var Handle : Integer ) : Integer;

Function dsp_FileClose( var Handle : Integer ) : Integer;

Function dsp_ReadHeader( var Handle : Integer;
                       var DRP      : DSPRECPLAY ) : Word;

Function dsp_WriteHeader( var Handle : Integer;
                        var DRP      : DSPRECPLAY ) : Word;

Function dsp_ReadBuffer( var Handle  : Integer;
                       lpBuffer  : pointer;
                       iHalfSize : Integer ) : Word;

Function dsp_WriteBuffer( var Handle  : Integer;
                        lpBuffer  : pointer;
                        iHalfSize : Integer ) : Word;

Procedure dsp_ClearBuffer( lpBuffer : pointer; uMemSize : Word );

Procedure dsp_DoRECPLAY( var Handle  : Integer;
                       iADC      : Boolean;
                       iSource   : Integer;
                       var DRP    : DSPRECPLAY;
                       iSecs     : Integer;
                       lpBuffer  : pointer;
                       uMemSize : Word );

```

Implementation

```

Uses MIXUTIL, IRQUTIL, DMAUTIL;

```

```

type SampleArrayB = array[0..65534] of byte; { Array with sample data }
type SampleArrayBPtr = ^SampleArrayB; { Pointer at this array }
type BPTR = ^BYTE;

```

```

{-- Global Variables -----}

```

```

Const
  pDspNames : Array[0..5] of string = ( 'Unknown sound card',
                                         'SB 1.5 or SB MCV',
                                         'SB 2.0',
                                         'SB Pro or SB Pro MCV',
                                         'SB 16, SB 16 ASP',
                                         '> SB16' );

```

```

{ Which buffer part should be loaded/saved next ? }
iBufCnt : Integer = 0;

```

```

var DSPBASE : SBASE;

{$ifndef DSP_VERSIONONLY}

lpUserFunc      : UserFunc;
lpDSP4UserFunc  : DSP4UserFunc;
lpOldInt        : OldIntProc;

const
  lIRQServed      : Longint = 0;
  lOldIRQServed   : Longint = 0;
{$endif}

{-- Functions -----}

{*****}
{ _dos_open : Open file with DOS (read and write) }
{*****}
{-----*}
{ Input : FileName : Name of file to be opened }
{ Output : valid Filehandle or -1 }
{-----*}
{ Note : The Pascal file functions cannot be used, }
{ because they basically only refer to objects on the }
{ heap. However, the DMA buffer cannot be }
{ allocated by the heap. }
{*****}
Function _dos_open( FileName: String ) : Integer;
var regs : Registers;
Begin
  regs.ah := $3D;
  regs.al := $2; { read and write }
  FileName := FileName + #0;
  regs.ds := LongInt( @FileName[1] ) shr 16; { Segment }
  regs.dx := LongInt( @FileName[1] ) and $FFFF; { Offset }

  msdos( regs );
  if ( regs.Flags and 1 ) <> 0 then
    _dos_open := -1
  else _dos_open := regs.ax;
End;

{*****}
{ _dos_close : Close DOS file }
{*****}
{-----*}
{ Input : handle : Handle of file opened previously }
{ by _dos_open. }
{*****}
}
Procedure _dos_close( handle : integer );
var regs : Registers;
Begin
  if handle >= 0 then
    Begin
      regs.ah := $3E;
      regs.bx := handle;
      msdos( regs );
    End;
  End;
End;

{*****}
{ _dos_read : Read data from DOS file }
{*****}
{-----*}
{ Input : handle : Handle of DOS file }
{ Buf : Untyped pointer at read buffer }
{ cnt : Number of characters to be read }
{*****}
Function _dos_read( handle : integer; Buf : Pointer; cnt : Integer ) : word;
var regs : Registers;
Begin
  if handle >= 0 then
    Begin
      regs.ah := $3F;
      regs.bx := handle;
      regs.cx := cnt;
      regs.ds := LongInt( Buf ) shr 16; { Segment }
      regs.dx := LongInt( Buf ) and $FFFF; { Offset }
      msdos( regs );
      if ( regs.Flags and 1 ) <> 0 then
        _dos_read := 0
      else _dos_read := regs.ax;
    End;
  End;
End;

{*****}
{ _dos_write: Write data to DOS file }
{*****}
{-----*}
{ Input : handle : Handle of DOS file }

```

```

{
    Buf      : Untyped pointer at write data
    cnt      : Number of characters to be written
}
*****}
Function _dos_write(handle : integer; Buf : Pointer; cnt : Integer): word;
var regs : Registers;
Begin
    if handle >= 0 then
    Begin
        regs.ah := $40;
        regs.bx := handle;
        regs.cx := cnt;
        regs.ds := LongInt( Buf ) shr 16;           { Segment }
        regs.dx := LongInt( Buf ) and $FFFF; { Offset }
        msdos( regs );
        if (regs.Flags and 1) <> 0 then
            _dos_write := 0
        else _dos_write := regs.ax;
    End;
End;

{*****}
{ _dos_filesize: Get length of a DOS file }
{*****}
{-----*}
{ Input : handle : Handle of DOS file }
{ Output : Length of file }
{-----*}
{ Note : To determine the length of a file, use the following }
{ procedure: }
{ 1.) Note current file pointer }
{ 2.) Move file pointer to end of file and }
{ return current position }
{ 3.) Move file pointer back to old position }
{*****}
Function _dos_filesize( handle : integer ) : longint;
var regs : Registers;
    actposl, actposh : WORD; { current file pointer position }
Begin
    _dos_filesize := 0; { identify errors first }
    if handle >= 0 then
    Begin
        regs.ah := $42; { read current position }
        regs.al := 1; { current file position }
        regs.bx := handle;
        regs.cx := 0; { no offset }
        regs.dx := 0;
        msdos( regs );
        if regs.Flags and 1 = 0 then
        Begin
            actposl := regs.ax;
            actposh := regs.dx;
        End
        else exit;

        regs.ah := $42; { read position at end of file }
        regs.al := 2; { position relative to end of file }
        regs.bx := handle;
        regs.cx := 0; { Offset = 0 => end of file }
        regs.dx := 0;
        msdos( regs );
        if regs.Flags and 1 = 0 then
        Begin
            _dos_filesize := longint( regs.dx ) shl 16 +
                            longint ( regs.ax );
        End
        else exit;

        regs.ah := $42; { restore old position }
        regs.al := 0; { relative to beginning of file }
        regs.bx := handle;
        regs.cx := actposh; { no offset }
        regs.dx := actposl;
        msdos( regs );
        if regs.Flags and 1 <> 0 then
        Begin
            _dos_filesize := 0;
            exit;
        End;
    End;
End;

{*****}
{ DummyUserFunc : Dummy function for procedure type UserFunc }
{*****}
Procedure DummyUserFunc( l : Longint );
begin

```

```

end;

{*****}
{ DummyDSP4UserFunc : Dummy function for procedure type }
{ DSP4UserFunc }
{*****}
Procedure DummyDSP4UserFunc( b : byte );
begin
end;

{*****}
{ dsp_SetBase : Set SB base structure for use of dsp_??? }
{ functions. }
{*****}
{-----*}
{ Input : SBBASE - structure initialized by SB_SetEnviron. }
{ iReset - reset DSP }
{ Output : == 0 - DSP functions were initialized }
{ DSP_ERRRESET == 1 - Error resetting DSP }
{ DSP_ERRVERSION == 2 - Error getting version number }
{ == -1 - No SBBASE structure passed }
{ or iDspPort == -1 }
{*****}
Function dsp_SetBase( var SBBASE : SBBASE; iReset : Boolean ) : Word;

Begin
  if @SBBASE <> NIL then
    if SBBASE.iDspPort <> -1 then
      Begin
        DSPBASE := SBBASE;
        if iReset then if dsp_Reset <> 0 then
          Begin
            dsp_SetBase := DSP_ERRRESET;
            Exit;
          End;

        if dsp_GetVersion( SBBASE ) <> 0 then
          Begin
            dsp_SetBase := DSP_ERRVERSION;
            Exit;
          End;

        if SBBASE.uDspVersion >= DSP_4XX then
          if SBBASE.iDspDmaW = -1 then
            SBBASE.iDspDmaW := SBBASE.iDspDmaB;

        {$ifndef DSP_VERSIONONLY}
        mix_SetBase( SBBASE, FALSE );
        {$endif}

        DSPBASE := SBBASE;
        dsp_SetBase := NO_ERROR;
      End
    else dsp_SetBase := ERROR;
  End;

{*****}
{ dsp_write : Pass data to DSP }
{-----*}
{ Input : iVal - data byte or command to be sent }
{ Output : == 0 - Data could not be transferred }
{ <> 0 - Data sent to DSP }
{*****}
Function dsp_Write( iVal : Word ) : Word;

Const cx : Word = 65535;

Begin
  { Internal counter == 0 => Transfer error! }
  { Read Write-Status and wait until OK. Decrement counter }
  While ( ( port[DSPBASE.iDspPort + DSP_WRITESTATUS] and $80 ) <> 0 )
    and ( cx <> 0 ) ) do
    Dec(cx);
    { Counter != 0, then Write-Status within default time }
    { Transfer of data byte is allowed : }
    if cx > 0 then port[DSPBASE.iDspPort + DSP_WRITECMDATA] := Byte (iVal);
    dsp_write := cx;
    { Return counter as error indicator }
  End;

{*****}
{ dsp_Read : Read data from DSP }
{-----*}
{ Input : Val - Byte variable that is to receive }
{ read value (output parameter) }
{ Output : == 0 - DSP doesn't provide any data }
{ <> 0 - DSP provides data }
{-----*}

```

```

{ Info : - Reading out DSP data only makes sense if
  an appropriate query command was sent beforehand.
}
*****}
Function dsp_Read( var Val : Byte ) : Word;

Const cx : Word = 65535;

Begin
  { Internal counter == 0 => Transfer error! }
  { Read Read-Status and wait until OK. Decrement counter }
  while (not ( ( port[DSPBASE.iDspPort + DSP_READSTATUS] and $80) <> 0)
    and ( cx <> 0 ) ) ) do
    Dec(cx);
    { Counter != 0, then Read-Status within default time OK: }
    { Reading of a data byte is allowed: }
    if cx > 0 then Val := Byte ( port[DSPBASE.iDspPort + DSP_READDATA] );
    dsp_Read := cx;
    { Return counter as error indicator }
End;

*****}
{ dsp_Reset : Reset DSP to output status }
{-----}
{ Output : NO_ERROR( 0 ) - Reset properly executed }
{          ERROR ( -1 ) - Reset not executed }
{-----}
{ Info : - Reset sequence - First 1, then send 0 to Reset Port of DSP }
{          then DSP-readout and wait }
{          to receive $AA. }
*****}
Function dsp_Reset : Word;

Const bVal : Byte = 0;

var dummy : Byte;

Begin
  { Write 1 to Reset-Port first, then write 0 }
  port[DSPBASE.iDspPort + DSP_RSET] := 1;
  dummy := port[DSPBASE.iDspPort + DSP_RSET]; { Wait at least }
  dummy := port[DSPBASE.iDspPort + DSP_RSET]; { 3 microseconds }
  dummy := port[DSPBASE.iDspPort + DSP_RSET];
  port[DSPBASE.iDspPort + DSP_RSET] := 0;
  dummy := port[DSPBASE.iDspPort + DSP_RSET]; { Wait at least }
  dummy := port[DSPBASE.iDspPort + DSP_RSET]; { 3 microseconds }
  dummy := port[DSPBASE.iDspPort + DSP_RSET];
  dsp_Read( bVal ); { Read value of DSP }
  if bVal = $AA then dsp_Reset := NO_ERROR
    else dsp_Reset := ERROR; { Value = $AA => OK }
End;

*****}
{ dsp_GetVersion : Get DSP version number }
{-----}
{ Input : SBBASE - SB base structure to be initialized }
{ Output : NO_ERROR ( 0 ) - no error }
{          DSP_ERRVERSION ( 2 ) - Transfer error }
{-----}
{ Info : - Along with the uDspVersion field, the name of }
{          the detected Sound Blaster card is specified }
*****}
Function dsp_GetVersion( var SBBASE : SBBASE ) : Word;

var bv1, bv2 : Byte;

Begin
  if ( ( dsp_Write(DSP_GETVER) <> 0 )
    and ( dsp_Read( bv1 ) <> 0 )
    and ( dsp_Read( bv2 ) <> 0 ) ) then
    Begin
      { Form version word }
      SBBASE.uDspVersion := bv1 * 256 + bv2;
      if ( ( bv1 >= 1 ) and ( bv1 <= 4 ) ) then
        SBBASE.pDspName := pDspNames[bv1]
      else
        SBBASE.pDspName := pDspNames[_SB_NONE];
      dsp_GetVersion := NO_ERROR;
    End
  else
    { Error with DSP data transfer }
    dsp_GetVersion := DSP_ERRVERSION;
End;

{ $ifndef DSP_VERSIONONLY } { Use only version control }
*****}
{ dsp_AdjustFrq : Adjust frequency to sound card }
{-----}
{ Input : Frq - WORD variable containing the desired }
{          frequency }
{ iADC - Recording (TRUE) or playback (FALSE) }

```

```

iStereo - Stereo mode (TRUE/FALSE)
}
Output : TRUE - Mono/Stereo valid
        FALSE - no stereo mode
}
}
*-----*
Info : This function checks whether the given frequency on the
current Sound Blaster card can be used
or else sets the highest possible frequency.
If you attempt to enable stereo mode on a card
that doesn't permit this, FALSE is returned.
Some of the high frequencies and only be used by
HI-SPEED commands.
}
*****}

```

```

Function dsp_AdjustFrq( var Frq      : Word;
                       iADC       : Boolean;
                       var Stereo  : Boolean ) : Boolean;

Begin
if iADC then
Begin { Recording }
if DSPBASE.uDspVersion >= DSP_4XX then
Begin
if Frq > FRQ4_ADC then Frq := FRQ4_ADC;
dsp_AdjustFrq := TRUE;
Exit;
End;

if DSPBASE.uDspVersion >= DSP_3XX then
Begin
if Stereo then
if Frq > FRQ3_STEREO_ADC then Frq := FRQ3_STEREO_ADC
else
if Frq > FRQ3_HIMONO_ADC then Frq := FRQ3_HIMONO_ADC;
dsp_AdjustFrq := TRUE;
Exit;
End;

if DSPBASE.uDspVersion >= DSP_201 then
Begin
if Frq > FRQ2p_HIMONO_ADC then Frq := FRQ2p_HIMONO_ADC;
if Stereo then
Begin
Stereo := FALSE; dsp_AdjustFrq := FALSE;
Exit;
End;
dsp_AdjustFrq := TRUE;
Exit;
End;

if DSPBASE.uDspVersion >= DSP_1XX then
Begin
if Frq > FRQ1_MONO_ADC then Frq := FRQ1_MONO_ADC;
if Stereo then
Begin
Stereo := FALSE; dsp_AdjustFrq := FALSE;
Exit;
End;
dsp_AdjustFrq := TRUE;
Exit;
End;
End
else { Playback }
Begin
if DSPBASE.uDspVersion >= DSP_4XX then
Begin
if Frq > FRQ4_DAC then Frq := FRQ4_DAC;
dsp_AdjustFrq := TRUE;
Exit;
End;

if DSPBASE.uDspVersion >= DSP_3XX then
Begin
if Stereo then
if Frq > FRQ3_STEREO_DAC then Frq := FRQ3_STEREO_DAC
else
if Frq > FRQ3_HIMONO_DAC then Frq := FRQ3_HIMONO_DAC;
dsp_AdjustFrq := TRUE;
Exit;
End;

if DSPBASE.uDspVersion >= DSP_201 then
Begin
if Frq > FRQ2p_HIMONO_DAC then Frq := FRQ2p_HIMONO_DAC;
if Stereo then
Begin
Stereo := FALSE;
dsp_AdjustFrq := FALSE;

```



```

Exit;
End;
dsp_AdjustFrq := TRUE;
Exit;
End;

if DSPBASE.uDspVersion >= DSP_1XX then
Begin
if Frq > FRQ1_MONO_DAC then Frq := FRQ1_MONO_DAC;
if Stereo then
Begin
Stereo := FALSE;
dsp_AdjustFrq := FALSE;
Exit;
End;
dsp_AdjustFrq := TRUE;
Exit;
End;
End;
dsp_AdjustFrq := FALSE; { DSP version < 1.00 }
End;

*****
{ dsp4_DACFrq : Set output frequency for DSP4.00 }
*-----*
{ Input : uFrq - output frequency to be set in Hertz [Hz] }
{ Output : NO_ERROR( 0 ) - Frequency set }
{ DSP_ERR4DACFRQ ( 4 ) - Transfer error }
*-----*
{ Info : - This function available beginning with DSP4.00! }
*****

Function dsp4_DACFrq( uFrq : Word ) : Word;

Begin
if( ( dsp_Write( DSP4_DACFR ) <> 0 ) and
( dsp_Write( HI( uFrq ) ) <> 0 ) and
( dsp_Write( LO( uFrq ) ) <> 0 ) ) then
dsp4_DACFrq := NO_ERROR
else
dsp4_DACFrq := DSP_ERR4DACFRQ;
End;

*****
{ dsp4_ADCFrq : Set input frequency of DSP4.00 }
*-----*
{ Input : uFrq - input frequency to be set in Hertz }
{ Output : NO_ERROR( 0 ) - Frequency set }
{ DSP_ERR4ADCFRQ ( 5 ) - Transfer error }
*-----*
{ Info : - This function available beginning with DSP4.00! }
*****

Function dsp4_ADCFrq( uFrq : Word ) : Word;

Begin
if( ( dsp_Write( DSP4_ADCFR ) <> 0 ) and
( dsp_Write( HI( uFrq ) ) <> 0 ) and
( dsp_Write( LO( uFrq ) ) <> 0 ) ) then
dsp4_ADCFrq := NO_ERROR
else
dsp4_ADCFrq := DSP_ERR4ADCFRQ;
End;

*****
{ dsp_SetFrq : Set input/output frequency of DSP }
*-----*
{ Input : Frq - Word variable containing the input/output }
{ frequency in Hertz [Hz]. After this function has been }
{ called, this word contains the }
{ frequency actually being used. }
{ Output : NO_ERROR( 0 ) - Frequency set }
{ DSP_ERRFRQ ( 3 ) - Transfer error }
*-----*
{ Info : - With DSP3.XX the frequency can be set in 256 steps/paces. }
{ The frequency actually being used }
{ is returned by this function. }
*****

Function dsp_SetFrq( var Frq : Word ) : Word;

var BTC : Byte;

Begin
{ Time constant }
bTC := Byte ( 256 - ( ( 1000000 + ( Frq div 2 ) ) div Frq ) );
if ( ( dsp_Write( DSP_SETTIMECONSTANT ) <> 0 ) and
( dsp_Write( bTC ) <> 0 ) ) then
Begin

```

```

Frq := Word ( 1000000 div ( 256 - bTC ) );
dsp_SetFrq := NO_ERROR;
End
else
dsp_SetFrq := ERROR;
End;

{*****}
{ dsp_CanStereo: Does current sound card support stereo mode? }
{-----}
{ Output : TRUE - Stereo mode is supported (starting with DSP3.00) }
{ FALSE - Stereo mode is not supported. }
{*****}
Function dsp_CanStereo : Boolean;
Begin
dsp_CanStereo := ( DSPBASE.uDspVersion >= DSP_3XX ) ;
End;

{*****}
{ dsp_HIMONOADCFrq : Is specified frequency a HISPEED frequency? }
{-----}
{ Output : Frequency for mono recordings }
{*****}
Function dsp_IsHIMONOADCFrq( uFrq : Word ) : Boolean;
Begin
if DSPBASE.uDspVersion <= DSP_1XX then
dsp_IsHIMONOADCFrq := FALSE
else
if DSPBASE.uDspVersion <= DSP_201 then
dsp_IsHIMONOADCFrq := ( uFrq <= FRQ2p_HIMONO_ADC )
else
if DSPBASE.uDspVersion <= DSP_3XX then
dsp_IsHIMONOADCFrq := ( uFrq <= FRQ3_HIMONO_ADC )
else
if DSPBASE.uDspVersion <= DSP_4XX then
dsp_IsHIMONOADCFrq := ( uFrq <= FRQ4_ADC )
else
dsp_IsHIMONOADCFrq := TRUE;
End;

{*****}
{ dsp_HIMONODACFrq : Is specified frequency a HISPEED frequency? }
{-----}
{ Output : Frequency for mono playback }
{*****}
Function dsp_IsHIMONODACFrq( uFrq : Word ) : Boolean;
Begin
if DSPBASE.uDspVersion <= DSP_1XX then
dsp_IsHIMONODACFrq := FALSE
else
if DSPBASE.uDspVersion <= DSP_201 then
dsp_IsHIMONODACFrq := ( uFrq <= FRQ2p_HIMONO_DAC )
else
if DSPBASE.uDspVersion <= DSP_3XX then
dsp_IsHIMONODACFrq := ( uFrq <= FRQ3_HIMONO_DAC )
else
if DSPBASE.uDspVersion <= DSP_4XX then
dsp_IsHIMONODACFrq := ( uFrq <= FRQ4_DAC )
else
dsp_IsHIMONODACFrq := TRUE;
End;

{*****}
{ dsp_MaxBits : Get highest sample resolution }
{-----}
{ Output : Maximum number of bits for a sample (8, 16) }
{*****}
Function dsp_MaxBits : Integer;
Begin
if DSPBASE.uDspVersion >= DSP_4XX then dsp_MaxBits := 16
else dsp_MaxBits := 8;
End;

{*****}
{ dsp_SetSpeaker : Switch DSP output on/off. }
{-----}
{ Input : iState == 0 Switch off DSP output }
{ <> 0 Switch on DSP output }
{ Output : NO_ERROR( 0 ) - Speaker engaged }
{ DSP_ERRSPEAKER ( 6 ) - Transfer error }
{*****}
Function dsp_SetSpeaker( iState : Boolean ) : Word;

```

```

var return : Word;

Begin
  if iState then return := dsp_Write( DSP_SPEAKERON )
  else return := dsp_Write( DSP_SPEAKEROFF );
  if return <> 0 then dsp_SetSpeaker := NO_ERROR
  else dsp_SetSpeaker := DSP_ERRSPEAKER;
End;

{*****}
{ dsp_SetTransferSize : Send number of bytes to be transferred }
{ to DSP. }
{-----}
{ Input : uSize - Number of bytes in transfer }
{ Output : NO_ERROR( 0 ) - Number set }
{ DSP_ERRTRANSIZE( 7 ) - Transfer error }
{*****}
Function dsp_SetTransferSize( uSize : Word ) : Word;

Begin
  if uSize <> 0 then
    Begin
      { Continue only when number != 0 is passed }
      Dec( uSize ); { Number - 1 is passed }
      { Send DSP command }
      if ( ( dsp_Write( DSP_SETTRANSIZE ) <> 0 ) and
        ( dsp_Write( LO( uSize ) ) <> 0 ) and { Send LO-byte }
        ( dsp_Write( HI( uSize ) ) <> 0 ) ) { Send HI-byte }
      then dsp_SetTransferSize := NO_ERROR
      else dsp_SetTransferSize := DSP_ERRTRANSIZE;
    End
  else
    dsp_SetTransferSize := DSP_ERRILLSIZE;
End;

{*****}
{ dsp_IrqHandler : Demo DSP Interrupt-Handler }
{-----*}
{ Info : - This is a normal interrupt handler. }
{ To simplify the programming of a custom handler }
{ in this handler everything necessary is done }
{ to confirm the interrupts. Afterwards }
{ a user function can be called. }
{*****}
Procedure dsp_IrqHandler;

var who, i : Byte;

Begin
  Inc(lIRQServed); { Increase number of previous calls }
  if DSPBASE.uDspVersion >= $0400 then { DSP version >= 4.00 }
  Begin
    { Which chip triggered the IRQ ? }
    who := Byte ( mix_Read( MIX4_IRQSOURCE ) );

    if ( @lpDSP4UserFunc <> @DummyDSP4UserFunc ) then
      lpDSP4UserFunc( who ); { Call user functions }

    {-- 8 bit DMA or Midi transfer -----}
    if ( who and MIX4_IRQ8DMA ) <> 0 then
      i := port[DSPBASE.iDspPort + $0e];

    {-- 16 bit DMA -----}
    if ( who and MIX4_IRQ16DMA ) <> 0 then
      i := port[DSPBASE.iDspPort + $0f];

    {-- MPU-401 UART -----}
    if ( who and MIX4_IRQMPU ) <> 0 then
      i := port[DSPBASE.iMpuPort];
  End
  else
    { IRQ acknowledge all other DSPs. }
    { Only causes: Midi and 8 bit DMA }
    i := port[DSPBASE.iDspPort + $0e];

  if @lpUserFunc <> @DummyUserFunc then
    lpUserFunc( lIRQServed ); { Call user function }

    { Signal end of IRQ to interrupt controller }
    irq_SendEOI( DSPBASE.iDspIrq );
End;

{*****}
{ dsp_RestoreIrqHandler : Restore original IRQ handler. }
{*****}
Procedure dsp_RestoreIrqHandler;

```

```

Begin
    irq_SetHandler( DSPBASE.iDspIrq, @lpOldInt );
End;

{*****}
{ dsp_InitIrqHandler : Install custom DSP IRQ handler. }
{*****}
Procedure dsp_InitIrqHandler;

Begin
    @lpOldInt := irq_SetHandler( DSPBASE.iDspIrq, @dsp_irqHandler );
End;

{*****}
{ dsp_SetUserIRQ : Specify user defined IRQ function. }
{*****}
{-----*}
{ Input : lpFunc - Address of user function. }
{ Info : - The function specified here is called by the custom }
{ DSP handler. }
{ The user function has the number of previous IRQ calls }
{ passed to it. }
{ By calling dsp_SetUserIRQ(DummyUserFunc) the program }
{ suppresses additional calls of the user function. }
{*****}
Procedure dsp_SetUserIRQ( lpFunc : UserFunc );

Begin
    lpUserFunc := lpFunc;
End;

{*****}
{ dsp4_SetUserIRQ : Specify user defined IRQ function. }
{*****}
{-----*}
{ Input : lpFunc - Address of user function. }
{ Info : - The custom DSP handler calls the function specified }
{ here. }
{ The user function has the number of previous IRQ calls }
{ passed to it. }
{ By calling dsp4_SetUserIRQ(DummyDSP4UserFunc) }
{ the program suppresses additional calls of the user function. }
{*****}
Procedure dsp4_SetUserIRQ( lpFunc : DSP4UserFunc );

Begin
    lpDSP4UserFunc := lpFunc;
End;

{*****}
{ dsp_WaitForNextIRQ : Wait for next interrupt triggered }
{ by the DSP }
{-----*}
{ Input : lpDoSomething - Address of a function to be executed while }
{ waiting. }
{ lPar - Parameter to be passed to }
{ the function. }
{ Output : TRUE : Since the last call for WaitForNextIRQ }
{ more than 1 IRQ call has occurred. }
{ FALSE : Since the last call for WaitForNextIRQ }
{ only 1 IRQ call has occurred. }
{*****}
Function dsp_WaitForNextIRQ( lpDoSomething : UserFunc;
    lpar : Longint ) : Boolean;

var iret : Boolean;

Begin
    iRet := ( lIRQServed - loldIRQServed > 1 );
    loldIRQServed := lIRQServed; { save current counter status }
    { Run loop until IRQ counter changes }
    while( loldIRQServed = lIRQServed ) do { Call function }
        if @lpDoSomething <> @DummyUserFunc then
            lpDoSomething( lPar );

    dsp_WaitForNextIRQ := iRet;
End;

{*****}
{ dsp_InitWaitForIRQ : Adjust InterruptCounter }
{*****}
Procedure dsp_InitWaitForIRQ;

Begin
    { old counter status = current counter status }
    loldIRQServed := lIRQServed;
End;

```

```

*****
{ dsp_8DAC : Direct 8 bit 'Digital->Analog conversion' (Output) }
-----
{ Input : bVal - 8 bit value to be output }
{ Output : NO_ERROR ( 0 ) - Could not output byte }
{ DSP_ERR8DAC( 9 ) - Transfer error }
*****
Function dsp_8DAC( bVal : Byte ) : Word;

Begin
  if ( ( dsp_Write( DSP_8DA ) <> 0 ) and ( dsp_Write( bVal ) <> 0 ) ) then
    dsp_8DAC := NO_ERROR
  else
    dsp_8DAC := DSP_ERR8DAC;
End;

*****
{ dsp_8ADC : Direct 8 bit 'Analog->Digital Conversion' (Sampling) }
-----
{ Input : Val - Byte variable that is to receive the converted }
{ value. }
{ Output : NO_ERROR ( 0 ) - Able to input byte }
{ DSP_ERR8ADC( 10 ) - Transfer error }
*****
Function dsp_8ADC( var Val : Byte ) : Word;

Begin
  if ( ( dsp_Write( DSP_8AD ) <> 0 ) and ( dsp_Read( Val ) <> 0 ) ) then
    dsp_8ADC := NO_ERROR
  else
    dsp_8ADC := DSP_ERR8DAC;
End;

*****
{ dsp_PLAY : Play back musical data of an array }
-----
{ Input : pBuffer - Address of array with the sample data }
{ uSize - Size of array }
{ uDelay - Value for delay loop (for Frequency) }
{ Output : NO_ERROR ( 0 ) - Able to output byte }
{ DSP_ERRPLAY ( 11 ) - Transfer error }
*****
Function dsp_PLAY( pBuffer : pointer;
                  uSize, uDelay : Word ) : Word;
var i, j : Word;
    BufPtr : SampleArrayBPtr;

Begin
  BufPtr := pBuffer;
  if dsp_SetSpeaker( ON ) <> 0 then dsp_PLAY := DSP_ERRPLAY
  else Begin { Sound output on }
    for i := 0 to uSize - 1 do { Output each single byte... }
      Begin
        if dsp_8DAC( BufPtr^[i] ) <> 0 then
          Begin
            dsp_PLAY := DSP_ERRPLAY;
            Exit;
          End;
        for j := 1 to uDelay do Begin End; { ... and wait }
      End;
    if dsp_SetSpeaker( OFF ) <> 0 then
      Begin
        dsp_PLAY := DSP_ERRPLAY; Exit;
      End;
    dsp_PLAY := NO_ERROR;
  End;
End;

*****
{ dsp_RECORD : Recording of musical data in an array }
-----
{ Input : pBuffer - Address of array recording musical data }
{ uSize - Size of array }
{ uDelay - Value for delay loop (for Frequency) }
{ Output : NO_ERROR ( 0 ) - Able to output byte }
{ DSP_ERRRECORD ( 12 ) - Transfer error }
*****
Function dsp_RECORD( pBuffer : pointer;
                   uSize, uDelay : Word ) : Word;

var i, j : Word;
    BufPtr : SampleArrayBPtr;

Begin
  BufPtr := pBuffer;

```

```

{ DSP speaker must be off during recording! }
if dsp_SetSpeaker( OFF ) <> 0 then
    dsp_RECORD := DSP_ERRRECORD
else Begin
    for i := 0 to uSize - 1 do
        Begin
            if dsp_8ADC( BufPtr^[i] ) <> 0 then
                Begin
                    dsp_RECORD := DSP_ERRRECORD; Exit;
                End;
            for j := 1 to uDelay do Begin End;
        End;
    End;
    dsp_RECORD := NO_ERROR;
End;

{ ***** }
{ dsp_InitBuffers: Prepare buffer for DSP recording/playback }
{ ***** }
Procedure dsp_InitBuffers;

Begin
    iBufCnt := 0;
End;

{ ***** }
{ dsp_FileOpen: Open file for DSP recording/playback }
{ ***** }
{
    Input : Filename - name of file
             Handle - handle of file
    Output : 0 - no error
             <> 0 - DOSERROR occurred
}
{ ***** }
Function dsp_FileOpen( Filename : string;
                     var Handle : Integer ) : Integer;

Begin
    Handle := _dos_open( FileName );
    dsp_FileOpen := integer( Handle = -1 );
End;

{ ***** }
{ dsp_FileClose: Close DSP recording/playback file }
{ ***** }
{
    Input : Handle - handle of file
    Output : 0 - no error
             <> 0 - DOSERROR occurred
}
{ ***** }
Function dsp_FileClose( var Handle : Integer ) : Integer;

Begin
    _dos_close( Handle );
End;

{ ***** }
{ dsp_ReadHeader : read header }
{ ***** }
{
    Input : Handle - handle of file
             pDRP - RECPLAY-Structure that is to receive header
    Output : Number of read characters
}
{ ***** }
Function dsp_ReadHeader( var Handle : Integer;
                       var DRP : DSPRECPLAY ) : Word;

Begin
    dsp_ReadHeader := _dos_read( Handle, @DRP, sizeof( DSPRECPLAY ) );
End;

{ ***** }
{ dsp_WriteHeader : Write header }
{ ***** }
{
    Input : Handle - handle of file
             DRP - RECPLAY-contains structure of header
    Output : Number of written characters
}
{ ***** }
Function dsp_WriteHeader( var Handle : Integer;
                       var DRP : DSPRECPLAY ) : Word;

Begin
    dsp_WriteHeader := _dos_write( Handle, @DRP, sizeof( DSPRECPLAY ) );
End;

{ ***** }
{ dsp_ReadBuffer : Read next block for playback }
{ ***** }
{
    Input : Handle - handle of file
             lpBuffer - Address of memory
             iHalfSize - (Size of memory) / 2
}

```

```

{ Output : Number of read characters }
}
-----*}
{ Note : This function automatically monitors whether the first }
{ or second half of the buffer must be read. To do this, }
{ the program counts all loaded buffers. Which part of }
{ the buffer is loaded with new data is determined based on the }
{ 'evenness' of this counter. }
}
*****}
Function dsp_ReadBuffer( var Handle : Integer;
                        lpBuffer : pointer;
                        iHalfSize : Integer ) : Word;
var BufPtr : BPTR;
Begin
    BufPtr := lpBuffer;
    if ( iBufCnt and 1 ) <> 0 then
        Inc( BufPtr, iHalfSize );
    BufPtr^:= 255;
    dsp_ReadBuffer := _dos_read( Handle, BufPtr, iHalfSize );
    Inc( iBufCnt );
End;

{ *****}
{ dsp_WriteBuffer : Save next block of recording }
}
-----*}
{ Input : Handle - Handle of file }
{ lpBuffer - Address of memory }
{ iHalfSize - (Size of memory) / 2 }
{ Output : Number of written characters }
}
-----*}
{ Note : This function automatically monitors whether the first }
{ or second half of the buffer must be saved. }
}
*****}
Function dsp_WriteBuffer( var Handle : Integer;
                        lpBuffer : pointer;
                        iHalfSize : Integer ) : Word;
var BufPtr : BPTR;
Begin
    BufPtr := lpBuffer;
    if ( iBufCnt and 1 ) <> 0 then Inc( BufPtr, iHalfSize );
    dsp_WriteBuffer := _dos_write( Handle, BufPtr, iHalfSize );
    Inc( iBufCnt );
End;

{ *****}
{ dsp_ClearBuffer : Clear buffer }
}
-----*}
{ Input : lpBuffer - Address of buffer }
{ uMemSize - Size of buffer }
}
-----*}
{ Note : The program calls this function }
{ after loading the last block to be played back to avoid an }
{ 'echo'. }
}
*****}
Procedure dsp_ClearBuffer( lpBuffer : pointer; uMemSize : Word );
var i : Word;
    l : word;
    BufPtr : SampleArrayBPTR;
Begin
    BufPtr := lpBuffer;
    for l := 0 to uMemSize - 1 do
        BufPtr^[l] := 0;
    End;

{ *****}
{ dsp_DoRecPlay : Harddisk-Recorder / Player }
}
-----*}
{ Input : Handle - Handle of file that either contains the data }
{ to be played back, or is to record data. }
{ iADC - Recording (TRUE) or playback (FALSE) }
{ iASource - Source of recording (CD, LINE, MIC) }
{ -1: use current mixer setting }
{ DRP - Header }
{ iSecs - Number of seconds to be recorded }
{ (ignored during playback) }
{ lpBuffer - DMA capable memory }
{ uMemSize - size of DMA memory }
}
*****}
Procedure dsp_DoRECPLAY( var Handle : Integer;
                        iADC : Boolean;
                        iSource : Integer;
                        var DRP : DSPRECPLAY;
                        iSecs : Integer;
                        lpBuffer : pointer;
                        uMemSize : Word );

```

```

var iEncore      : Boolean;
    DSP_cmd,
    m,
    cBackup      : Byte;
    lSamples      : Longint;
    BufPtr        : SampleArrayBPtr;

Const
    dsp4_Mode : Byte      = 0;
    iDivisor   : Integer = 2;

Begin
    BufPtr := lpBuffer;
    if DRP.iStereo then
        DRP.iStereo := dsp_CanStereo;

    if DRP.iBits = 16 then
        DRP.iBits := dsp_MaxBits else DRP.iBits := 8;

    dsp_AdjustFrq( DRP.uFrequency, iADC, DRP.iStereo );
    dsp_InitIrqHandler;           { Install DSP interrupt function }
    dsp_InitBuffers;             { Initialize DMA Double/SplitBuffering }
    dsp_ClearBuffer( lpBuffer, uMemSize );

    if not iADC then              { Playback: fill buffer }
        Begin
            lSamples := _dos_filesize( Handle );
            dsp_ReadBuffer( Handle, lpBuffer, uMemSize div 2 );
            dsp_ReadBuffer( Handle, lpBuffer, uMemSize div 2 );
        End
    else
        lSamples := DRP.uFrequency * iSecs;

    {- DSP3xx -----}
    if DSPBASE.uDspVersion < DSP_4XX then
        Begin
            if DRP.iStereo then
                if iADC then mix3_PrepereForStereo( ADC)
                    else mix3_PrepereForStereo( DAC);

            if iADC then           { Recording }
                Begin
                    if iSource >= 0 then mix3_SetADCSrc( iSource );
                    dsp_SetSpeaker( OFF ); { With recordings, DSP speaker always off }
                                           { Mono or stereo recording? }

                    if DSPBASE.uDspVersion >= DSP_3XX then
                        if DRP.iStereo then dsp_Write( DSP3_STEREOADC )
                            else dsp_Write( DSP3_MONOADC );
                        { In stereo mode or with high-speed transfers }
                        { use HI_SPEED DSP commands. }

                    DSP_cmd := Byte ( DSP_8DMAAUTOADC );
                    if DSPBASE.uDspVersion >= DSP_201 then
                        if ( DRP.iStereo or
                            dsp_IsHIMONOADCfrq( DRP.uFrequency ) ) then
                            DSP_cmd := Byte ( DSP2p_8DMAHIAUTOADC );

                End
            else                     { Playback }
                Begin { DSP3XX }
                    dsp_Write( DSP3_MONOADC );           { Recording mode always MONO }

                    dsp_SetSpeaker( ON );                { DSP speaker on }
                    if DRP.iStereo then                  { Stereo playback }
                        Begin { With stereo playback, output $80 via DSP first }
                            { using DMA. }
                            { replace first byte of DMA capable memory with $80 }
                            cBackup := BufPtr^[0];
                            BufPtr^[0] := $80;
                            { Program DMA channel for outputting a byte... }
                            dma_SetChannel( DSPBASE.iDspDmaB, @lpBuffer, 1,
                                MODE_SINGLE or MODE_READ );

                            dsp_Write(DSP_8DMADAC); { ...and output a byte via DSP }
                            dsp_Write( 0 );
                            dsp_Write( 0 );
                                { Wait for end of DMA transfer }
                            dsp_WaitForNextIRQ( DummyUserFunc, 0 );
                            BufPtr^[0] := cBackup; { Create DMA buffer again }
                            DSP_cmd := DSP2p_8DMAHIAUTODAC; {Stereo playback always HiSpeed}
                        End
                    else { Mono playback }
                        Begin
                            DSP_cmd := Byte ( DSP_8DMAAUTODAC ); { DSP_201 }
                            if dsp_IsHIMONODACfrq( DRP.uFrequency ) then
                                DSP_cmd := Byte ( DSP2p_8DMAHIAUTODAC );
                            End;
                End
            End
        End
    End

```



```

End;

{-- Program DMA for recording/playback -----}
if iADC then
    dma_SetChannel( DSPBASE.iDspDmaB, lpBuffer, uMemSize,
                    Byte ( MODE_SINGLE or MODE_AUTOINIT or MODE_WRITE ) )
else
    dma_SetChannel( DSPBASE.iDspDmaB, lpBuffer, uMemSize,
                    Byte ( MODE_SINGLE or MODE_AUTOINIT or MODE_READ ) );

dsp_SetFrq( DRP.uFrequency );
    { An IRQ is to be triggered after each buffer half: }
dsp_SetTransferSize( uMemSize div 2 );
dsp_Write( DSP_cmd );           { Send DSP command }
End
else

{-- DSP 4.XX -----}
Begin
    if iADC then
        case iSource of
            CD:
                Begin
                    mix4_SetADCSOURCEL( CD_L, TRUE ); { With mono recordings }
                    mix4_SetADCSourceR( CD_R, TRUE ); { only left ADC is used }
                End;

            LINE:
                Begin
                    mix4_SetADCSOURCEL( LINE_L, TRUE );
                    mix4_SetADCSourceR( LINE_R, TRUE );
                End;

            MIC:
                Begin
                    mix4_SetADCSOURCEL( MIC, TRUE );
                    mix4_SetADCSourceR( MIC, TRUE );
                End;
            End;

        if not iADC then
            mix4_SetVolume( VOICE, 255, 255 )
        else
            if not DRP.iStereo then mix4_PrepareForMonoADC;
                { Program DMA channel for 8 or 16 bits }
            if iADC then m := Byte (MODE_SINGLE or MODE_AUTOINIT or MODE_WRITE )
                else m := Byte (MODE_SINGLE or MODE_AUTOINIT or MODE_READ );

            if DRP.iBits = 16 then
                dma_SetChannel( DSPBASE.iDspDmaW, lpBuffer, uMemSize, m)
            else
                dma_SetChannel( DSPBASE.iDspDmaB, lpBuffer, uMemSize, m);
                    { Set input/output frequency }

            if iADC then dsp4_ADCFrq( DRP.uFrequency )
                else dsp4_DACFrq( DRP.uFrequency );

                { Recording/playback is specified by command byte }
            if iADC then
                DSP_cmd := DSP4_CMDADC
            else
                DSP_cmd := DSP4_CMDDAC;

            DSP_cmd := DSP_cmd or DSP4_CMDAUTOINIT or DSP4_CMDFIFO;
            if DRP.iBits = 16 then DSP_cmd := DSP_cmd or DSP4_CMD16DMA
                else DSP_cmd := DSP_cmd or DSP4_CMD8DMA;
            if DRP.iStereo then dsp4_Mode := DSP4_MODESTEREO
                else dsp4_Mode := DSP4_MODEMONO;
            dsp4_Mode := dsp4_Mode or DSP4_MODESIGNED;

                { Send command, mode and transfer length to DSP }
            dsp_Write( DSP_cmd );
            dsp_Write( dsp4_Mode );

                { Half buffer in words or bytes }
            if ( DRP.iBits = 16 ) then iDivisor := iDivisor * 2;
            dsp_Write( LO( ( uMemSize div iDivisor ) - 1 ) );
            dsp_Write( HI( ( uMemSize div iDivisor ) - 1 ) );
        End;

dsp_InitWaitForIRQ;           { Initialize wait for IRQ }

Repeat
    if dsp_WaitForNextIRQ( DummyUserFunc, 0 ) then
        Writeln( 'Data lost!' );
        write( '.' );
    if iADC then

```

```

Begin
    iEncore := FALSE;           { don't continue? }
    if dsp_WriteBuffer( Handle, lpBuffer, uMemSize div 2 ) <> 0 then
        if lSamples > 0 then
            Begin
                iEncore := TRUE;
                lSamples := lSamples - uMemSize div 2;
            End;
        End
    else
        iEncore := dsp_ReadBuffer( Handle, lpBuffer, uMemSize div 2 ) <> 0;
Until not iEncore;

dsp_ClearBuffer( lpBuffer, uMemSize );

    { HISpeed commands can only be interrupted by resetting }
if ( ( DSP_cmd = DSP2p_8DMAHIAUTODAC ) or
    ( DSP_cmd = DSP2p_8DMAHIAUTOADC ) ) then
    Begin
        dsp_RestoreIrqHandler;           { install old IRQ handler }
        dsp_Reset;
    End
else
    Begin
        if ( not DRP.iStereo and ( DSPBASE.uDspVersion >= DSP_4XX ) ) then
            mix4_PrepareForMonoADC;

        if ( ( DSP_cmd and $F0 ) = DSP4_CMD8DMA ) then
            dsp_Write( DSP4_EXIT8DMA )    { Exit after current block }
        else
            if ( ( DSP_cmd and $F0 ) = DSP4_CMD16DMA ) then
                dsp_Write( DSP4_EXIT16DMA ) { Exit after current block }
            else
                Begin
                    if ( DRP.iStereo and ( DSPBASE.uDspVersion <= DSP_3XX ) ) then
                        mix3_RestoreFromStereo;
                        dsp_SetSpeaker( OFF );           { DSP speaker off }
                        dsp_Write(DSP2p_EXITAUTOINIT); { Exit after current block }
                    End;
                    dsp_WaitForNextIRQ( DummyUserFunc, 0 ); { wait for end }
                    dsp_RestoreIrqHandler; { install old IRQ handler }
                End;
    End;
End;
{$Endif}                                { #ifndef DSP_VERSIONONLY }

{-- UNIT Initialization -----}

begin
    lpUserFunc      := DummyUserFunc;
    lpDSP4UserFunc := DummyDSP4UserFunc;
End.

```