

```

/*****
/*          D S P U T I L . C          */
**-----**
/* Task      : Provides functions for programming the SoundBlaster */
/*          DSP.          */
**-----**
/* Author     : Michael Tischer / Bruno Jennrich          */
/* Developed on : 3/20/1994          */
/* Last update  : 8/17/1995          */
**-----**
/* Note: If this module is compiled with the defined icon          */
/*      DSP_VERSIONONLY only the functions necessary for          */
/*      determining the version will be compiled          */
*****/
#ifndef __DSPUTIL_C          /* DSPUTIL.C can also be #Included! */
#define __DSPUTIL_C

/*-- Add Include files -----*/

#include <dos.h>
#include <io.h>
#include <stdio.h>
#include <errno.h>

#include "dsputil.h"
#include "dmautil.h"
#include "mixutil.h"
#include "irqutil.h"

#define _SB_NONE 0
#define _SB_1XX 1  /* DSP versions 1.00 to 1.99 */
#define _SB_2XX 2  /* DSP versions 2.00 to 2.99 */
#define _SB_3XX 3  /* DSP versions 3.00 to 3.99 */
#define _SB_4XX 4  /* DSP versions 4.00 to 4.99 */
#define _SB_XXX 5  /* DSP version > 4.99 */

/*-- Global Variables -----*/
static LPCHAR pDspNames[ ]= { "Unknown sound card",
                              "SB 1.5 or SB MCV",
                              "SB 2.0",
                              "SB Pro or SB Pro MCV",
                              "SB 16, SB 16 ASP",
                              "> SB16" };

SBBASE _FP DSPBASE;

#ifndef DSP_VERSIONONLY
VOID ( _FP *lpUserFunc )( LONG ) = NULL;
VOID ( _FP *lpDSP4UserFunc )( BYTE ) = NULL;
VOID ( _interrupt _FP *lpOldInt )() = NULL;

static LONG lIRQServed = 0L;
static LONG loldIRQServed = 0L;
#endif

/*-- Functions -----*/

/*****
/* dsp_SetBase : Set SB Base structure for use of dsp_???          */
/*          functions.          */
**-----**
/* Input : pSBBASE - Address of structure initialized by          */
/*          'SB_SetEnviron'.          */
/*          iReset   - Reset DSP          */
/* Output :          == 0 - DSP functions were initialized          */
/*          properly          */
/*          DSP_ERRRESET == 1 - Error resetting DSP          */
/*          DSP_ERRVERSION == 2 - Error determining version number */
/*          ==-1 - No SBBASE structure passed          */
/*          or iDspPort == -1          */
*****/
WORD dsp_SetBase( PSBBASE pSBBASE, WORD iReset )
{
    if( pSBBASE )
        if( pSBBASE->iDspPort != -1 )
        {

```

```

    DSPBASE = *pSBBASE;
    if( iReset ) if( dsp_Reset() ) return DSP_ERRRESET;
    if( dsp_GetVersion( pSBBASE ) ) return DSP_ERRVERSION;
    if( pSBBASE->uDspVersion >= DSP_4XX )
        if( pSBBASE->iDspDmaW == -1 )
            pSBBASE->iDspDmaW = pSBBASE->iDspDmaB;

#ifdef DSP_VERSIONONLY
    mix_SetBase( pSBBASE, FALSE );
#endif

    DSPBASE = *pSBBASE;
    return NO_ERROR;
}
return ERROR;
}

/*****
/* dsp_write : Pass data to DSP
**-----*/
/* Input : iVal - data byte or command to be sent
/* Output : == 0 - Could not send data
/*          <> 0 - Sent data to DSP
*****/
WORD dsp_Write( WORD iVal )
{ WORD cx = 65535U; /* Internal counter == 0 => transfer error! */
  /* Read Write-Status and wait until OK. Decrement counter */
  while( ( inp( DSPBASE.iDspPort + DSP_WRITESTATUS ) & 0x80 ) && cx )
      cx--;
      /* Counter != 0, then Write-Status within predetermined time */
      /* transfer of a data byte is allowed:
  if( cx ) outp( DSPBASE.iDspPort + DSP_WRITECMDATA, ( BYTE )iVal );
  return cx; /* Return counter as error indicator */
}

/*****
/* dsp_Read : Read data from DSP
**-----*/
/* Input : pVal - Address of 'int' that is to accept the
/*          read value ('Output parameter')
/* Output : == 0 - DSP does not provide data
/*          <> 0 - DSP provides data
*****/
/* Info : - Reading out DSP data only makes sense when a
/*          corresponding query command was sent beforehand.
*****/
WORD dsp_Read( PBYTE pVal )
{ WORD cx = 65535U; /* Internal counter == 0 => transfer error! */
  /* Read Read-Status and wait until OK. Decrement counter */
  while( !( inp( DSPBASE.iDspPort + DSP_READSTATUS ) & 0x80 ) && cx )
      cx--;
      /* Counter != 0, then Read-Status within predetermined time OK:
      /* reading a data byte is allowed:
  if( cx ) *pVal = ( BYTE )inp( DSPBASE.iDspPort + DSP_READDATA );
  return cx; /* Return counter as error indicator */
}

/*****
/* dsp_Reset : Reset DSP to original status
**-----*/
/* Output : NO_ERROR( 0 ) - Reset executed properly
/*          ERROR ( -1 ) - Reset not executed
*****/
/* Info : - Reset Sequence - Send 1 first, then 0 to Reset-Port of
/*          DSP then DSP readout and wait for
/*          reception of 0xAA.
*****/
WORD dsp_Reset( VOID )
{ BYTE bVal = 0;

    /* Write 1, then 0 to Reset-Port */
    outp( DSPBASE.iDspPort + DSP_RESET, 1 );
    inp( DSPBASE.iDspPort + DSP_RESET ); /* wait at least 3
    inp( DSPBASE.iDspPort + DSP_RESET ); /* microseconds
    inp( DSPBASE.iDspPort + DSP_RESET );
    outp( DSPBASE.iDspPort + DSP_RESET, 0 );
    inp( DSPBASE.iDspPort + DSP_RESET );

```

```

inp( DSPBASE.iDspPort + DSP_RESET );
inp( DSPBASE.iDspPort + DSP_RESET );
dsp_Read( &bVal ); /* Read value from DSP */
return (bVal == 0xAA) ? NO_ERROR : ERROR; /* Value = 0xAA => OK */
}

/*****
 * dsp_GetVersion : Get DSP version number
 *-----*/
/* Input : pSBBASE - Address of Sbase structure to be
 *          initialized
 * Output : NO_ERROR ( 0 ) - no error
 *          DSP_ERRVERSION ( 2 ) - transfer error
 *-----*/
/* Info : - Along with the uDsp version field, the name of the
 *          detected SoundBlaster card is also specified
 *****/
WORD dsp_GetVersion( PSBBASE pSBBASE)
{
    BYTE bV1, bV2;
    if( dsp_Write(DSP_GETVERSION) && dsp_Read(&bV1) && dsp_Read(&bV2) )
    { /* Make version word */
        pSBBASE->uDspVersion = MAKEWORD( bV1, bV2 );
        if((bV1>=1)&&(bV1<=4)) pSBBASE->pDspName = pDspNames[ bV1 ];
        else pSBBASE->pDspName = pDspNames[ _SB_NONE ];
        return NO_ERROR;
    }
    return DSP_ERRVERSION; /* Error transferring DSP data */
}

#ifndef DSP_VERSIONONLY /* Use only version control */
/*****
 * dsp_AdjustFrq : Adjust frequency to sound card
 *-----*/
/* Input : pFrq - Address of INT containing the desired
 *          frequency
 *          iADC - Record (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
 *          can be used on the current SoundBlaster card, otherwise
 *          it sets the greatest possible frequency.
 *          If an attempt is made to enable stereo mode on a card that
 *          doesn't allow it, the value FALSE is returned.
 *          Some of the high frequencies can only be used by the
 *          HI-SPEED commands.
 *****/
WORD dsp_AdjustFrq( PWORD pFrq, INT iADC, PINT pStereo )
{
    if( iADC )
    {
        if( DSPBASE.uDspVersion >= DSP_4XX )
        {
            *pFrq = *pFrq > FRQ4_ADC ? FRQ4_ADC : *pFrq;
            return TRUE;
        }
        if( DSPBASE.uDspVersion >= DSP_3XX )
        {
            if( *pStereo )
                *pFrq = *pFrq > FRQ3_STEREO_ADC ? FRQ3_STEREO_ADC : *pFrq;
            else
                *pFrq = *pFrq > FRQ3_HIMONO_ADC ? FRQ3_HIMONO_ADC : *pFrq;
            return TRUE;
        }
        if( DSPBASE.uDspVersion >= DSP_201 )
        {
            *pFrq = *pFrq > FRQ2p_HIMONO_ADC ? FRQ2p_HIMONO_ADC : *pFrq;
            if( *pStereo )
            {
                *pStereo = FALSE; return FALSE;
            }
            return TRUE;
        }
        if( DSPBASE.uDspVersion >= DSP_1XX )
        {

```

```

        *pFrq = *pFrq > FRQ1_MONO_ADC ? FRQ1_MONO_ADC : *pFrq;
        if( *pStereo )
        {
            *pStereo = FALSE; return FALSE;
        }
        return TRUE;
    }
}
else
{
    if( DSPBASE.uDspVersion >= DSP_4XX )
    {
        *pFrq = *pFrq > FRQ4_DAC ? FRQ4_DAC : *pFrq;
        return TRUE;
    }

    if( DSPBASE.uDspVersion >= DSP_3XX )
    {
        if( *pStereo )
            *pFrq = *pFrq > FRQ3_STEREO_DAC ? FRQ3_STEREO_DAC : *pFrq;
        else
            *pFrq = *pFrq > FRQ3_HIMONO_DAC ? FRQ3_HIMONO_DAC : *pFrq;
        return TRUE;
    }

    if( DSPBASE.uDspVersion >= DSP_201 )
    {
        *pFrq = *pFrq > FRQ2p_HIMONO_DAC ? FRQ2p_HIMONO_DAC : *pFrq;
        if( *pStereo )
        {
            *pStereo = FALSE; return FALSE;
        }
        return TRUE;
    }

    if( DSPBASE.uDspVersion >= DSP_1XX )
    {
        *pFrq = *pFrq > FRQ1_MONO_DAC ? FRQ1_MONO_DAC : *pFrq;
        if( *pStereo )
        {
            *pStereo = FALSE; return FALSE;
        }
        return TRUE;
    }
}
return FALSE; /* DSP-Version < 1.00 */
}

/*****
/* 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 only available for DSP4.00 and above! */
/*****
WORD dsp4_DACFrq( WORD uFrq )
{
    if( dsp_Write( DSP4_DACFRQ ) &&
        dsp_Write( HIBYTE( uFrq ) ) &&
        dsp_Write( LOBYTE( uFrq ) ) ) return NO_ERROR;
    return DSP_ERR4DACFRQ;
}

/*****
/* 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 errorr */
/*-----**/
/* Info : - This function only available for DSP4.00 and above! */
/*****
WORD dsp4_ADCFrq( WORD uFrq )
{
    if( dsp_Write( DSP4_ADCFRQ ) &&

```

```

        dsp_Write( HIBYTE( uFrq ) ) &&
        dsp_Write( LOBYTE( uFrq ) ) ) return NO_ERROR;
    return DSP_ERR4ADCFRQ;
}

/*****
/* dsp_SetFrq : Set input/output frequency of DSP
**-----**
/* Input   : pFrq - Address of WORD that contains the input/output
/*           frequency in Hertz [Hz]. After the call for this
/*           function this WORD contains the frequency that is
/*           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 increments.
/*         This function returns the frequency that is actually
/*         being used.
*****/
WORD dsp_SetFrq( PWORD pFrq )
{
    BYTE bTC;
    bTC = (BYTE)( 256L - ( ( 1000000L + ( *pFrq / 2 ) ) / *pFrq ) );
    if( dsp_Write( DSP_SETTIMECONSTANT ) && dsp_Write( bTC ) )
    {
        *pFrq = ( WORD ) ( 1000000L / (256L - bTC ) );
        return NO_ERROR;
    }
    return ERROR;
}

/*****
/* dsp_CanStereo: Supports current sound card stereo mode?
**-----**
/* Output : TRUE - Stereo mode supported (starting with DSP3.00)
/*         FALSE - Stereo mode not supported.
*****/
WORD dsp_CanStereo( void )
{
    return ( DSPBASE.uDspVersion >= DSP_3XX ) ? TRUE : FALSE;
}

/*****
/* dsp_HIMONOADCFrq : Is specified frequency a HISPEED frequency?
**-----**
/* Output : Frequency for mono recordings
*****/
WORD dsp_IsHIMONOADCFrq( WORD uFrq )
{
    if( DSPBASE.uDspVersion <= DSP_1XX ) return FALSE;
    if( DSPBASE.uDspVersion <= DSP_201 )
        return ( uFrq <= FRQ2p_HIMONO_ADC );
    if( DSPBASE.uDspVersion <= DSP_3XX )
        return ( uFrq <= FRQ3_HIMONO_ADC );
    if( DSPBASE.uDspVersion <= DSP_4XX )
        return ( uFrq <= FRQ4_ADC );
    return TRUE;
}

/*****
/* dsp_HIMONODACFrq : Is specified frequency a HISPEED frequency?
**-----**
/* Output : Frequency for Mono playback
*****/
WORD dsp_IsHIMONODACFrq( WORD uFrq )
{
    if( DSPBASE.uDspVersion <= DSP_1XX ) return FALSE;
    if( DSPBASE.uDspVersion <= DSP_201 )
        return ( uFrq <= FRQ2p_HIMONO_DAC );
    if( DSPBASE.uDspVersion <= DSP_3XX )
        return ( uFrq <= FRQ3_HIMONO_DAC );
    if( DSPBASE.uDspVersion <= DSP_4XX )
        return ( uFrq <= FRQ4_DAC );
    return TRUE;
}

```

```

/*****
/* dsp_MaxBits : Determine highest sample resolution */
/*-----*/
/* Output : Maximum number of bits in a sample (8, 16) */
/*****
int dsp_MaxBits( VOID )
{
    return ( DSPBASE.uDspVersion >= DSP_4XX ) ? 16 : 8;
}

/*****
/* dsp_SetSpeaker : Switch DSP output on/off. */
/*-----*/
/* Input : iState - == 0 Switch DSP output off */
/*          <> 0 DSP Switch output on */
/* Output : NO_ERROR( 0 ) - Speaker switched */
/*          DSP_ERRSPEAKER ( 6 ) - transfer error */
/*****
WORD dsp_SetSpeaker( WORD iState )
{
    return dsp_Write( iState ? DSP_SPEAKERON : DSP_SPEAKEROFF ) ?
        NO_ERROR : DSP_ERRSPEAKER;
}

/*****
/* 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_ERRTRANSSIZE( 7 ) - transfer error */
/*****
WORD dsp_SetTransferSize( WORD uSize )
{
    if( uSize ) /* Continue only if number != 0 is passed */
    {
        uSize--; /* number - 1 is passed */
        if( dsp_Write( DSP_SETTRANSFERSIZE ) && /* Send DSP command */
            dsp_Write( LOBYTE( uSize ) ) && /* send LO byte */
            dsp_Write( HIBYTE( uSize ) ) ) /* send HI byte */
        {
            return NO_ERROR;
        }
        return DSP_ERRTRANSSIZE;
    }
    return DSP_ERRILLSIZE;
}

/*****
/* dsp_IrqHandler : Demo DSP Interrupt Handler */
/*-----*/
/* Info : - This is an ordinary interrupt handler. */
/*          To simplify the programming of a custom handler, */
/*          in this handler all the necessary steps have been taken */
/*          to confirm the interrupts. Afterwards, a user function */
/*          can be called. */
/*****
VOID _interrupt _FP dsp_IrqHandler( )
{
    lIRQServed++; /* Increase number of previous calls */

    if( DSPBASE.uDspVersion >= 0x0400 ) /* DSP version >= 4.00 */
    {
        BYTE who; /* Which chip triggered IRQ ? */

        who = ( BYTE )mix_Read( MIX4_IRQSOURCE );

        /* Call user function if available */
        if( lpDSP4UserFunc ) lpDSP4UserFunc( who );

        /* 8 bit DMA or Midi transfer */
        if (who & MIX4_IRQ8DMA ) inp( DSPBASE.iDspPort + 0x0e );
        /* 16 bit DMA */
        if (who & MIX4_IRQ16DMA ) inp( DSPBASE.iDspPort + 0x0f );
        /* MPU-401 UART */
        if (who & MIX4_IRQMPU ) inp( DSPBASE.iMpuPort );
    }

    /* IRQ-Acknowledge all other DSPs. */
    /* Only causes: Midi and 8 bit DMA */
    else inp( DSPBASE.iDspPort + 0x0e );
}

```

```

    if( lpUserFunc ) lpUserFunc( lIRQServed );

    /* Signal end of IRQ to interrupt controller */
    irq_SendEOI( DSPBASE.iDspIrq );
}

/*****
/* dsp_RestoreIrqHandler : Restore original IRQ handler.
*****/
VOID dsp_RestoreIrqHandler( VOID )
{
    irq_SetHandler( DSPBASE.iDspIrq, lpOldInt );
}

/*****
/* dsp_InitIrqHandler : Set up custom DSP-IRQ handler.
*****/
VOID dsp_InitIrqHandler( VOID )
{
    lpOldInt = irq_SetHandler( DSPBASE.iDspIrq, dsp_IrqHandler );
}

/*****
/* dsp_SetUserIRQ : Specify user-defined IRQ function.
*****/
/* Input : lpFunc - Address of user function.
/* Info : - The function specified here is called by custom
/* DSP-Handler. PROTOTYPE: void far User( LONG irqcnt ){
/* The number of previous IRQ calls is passed to the user
/* function. The call of the custom function is interrupted
/* by 'dsp_SetUserIRQ(NULL);'
*****/
VOID dsp_SetUserIRQ( VOID ( _FP *lpFunc)( LONG ) )
{
    lpUserFunc = lpFunc;
}

/*****
/* dsp4_SetUserIRQ : Specify user-defined IRQ function.
*****/
/* Input : lpFunc - Address of user function.
/* Info : - The function specified here is called by the custom DSP
/* handler. PROTOTYPE: void far DSP4User( BYTE who ){
/* The number of previous IRQ calls is passed to the user
/* function. The call of the custom function is interrupted
/* by 'dsp4_SetUserIRQ(NULL);'
*****/
VOID dsp4_SetUserIRQ( VOID ( _FP *lpFunc)( BYTE ) )
{
    lpDSP4UserFunc = lpFunc;
}

/*****
/* dsp_WaitForNextIRQ : Wait for next interrupt triggered by DSP
*****/
/* Input : pDoSomething - Address of a function to be executed while
/* waiting.
/* lPar - Parameter to be passed to the function
/* Output : TRUE : Since the last call of WaitForNextIRQ more than
/* 1 IRQ call has passed.
/* FALSE : Since the last call of WaitForNextIRQ only
/* 1 IRQ call has passed.
*****/
WORD dsp_WaitForNextIRQ( VOID ( _FP *lpDoSomething)( LONG ), LONG lPar )
{ WORD iRet;

    iRet = ( lIRQServed - loldIRQServed > 1L );
    loldIRQServed = lIRQServed; /* save current counter status */
    /* Execute loop until IRQ counter changes */
    while( loldIRQServed == lIRQServed ) /* Call function */
        if( lpDoSomething ) lpDoSomething( lPar );

    return iRet;
}

/*****

```

```

/* dsp_InitWaitForIRQ : Adjust Interrupt counter */
/*****
VOID dsp_InitWaitForIRQ( VOID )
{
    /* old counter status = current counter status */
    loldIRQServed = lIRQServed;
}

/*****
/* dsp_8DAC : Direct 8 Bit "Digital->Analog Conversion" (Output) */
/*-----*/
/* Input   : bVal - 8 bit value to be output */
/* Output   : NO_ERROR ( 0 ) - Able to output byte */
/*          : DSP_ERR8DAC( 9 ) - transfer error */
/*****
WORD dsp_8DAC( BYTE bVal )
{
    return ( dsp_Write( DSP_8DAC ) && dsp_Write( bVal ) ) ?
        NO_ERROR : DSP_ERR8DAC;
}

/*****
/* dsp_8ADC : Direct 8 Bit "Analog->Digital Conversion" (Samples) */
/*-----*/
/* Input   : pVal - Address of byte to receive converted value */
/* Output   : NO_ERROR ( 0 ) - Able to read byte */
/*          : DSP_ERR8ADC( 10 ) - transfer error */
/*****
WORD dsp_8ADC( PBYTE pVal )
{
    return ( dsp_Write( DSP_8ADC ) && dsp_Read( pVal ) ) ?
        NO_ERROR : DSP_ERR8ADC;
}

/*****
/* dsp_PLAY : Play musical data of an array */
/*-----*/
/* Input   : pBuffer - Address of array containing musical 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 */
/*****
WORD dsp_PLAY( PBYTE pBuffer, WORD uSize, WORD uDelay )
{ WORD i,j;

    if( dsp_SetSpeaker( ON ) ) return DSP_ERRPLAY; /* Sound output on */
    for( i = 0; i < uSize; i++ ) /* Output every single byte... */
    {
        if( dsp_8DAC( pBuffer[ i ] ) ) return DSP_ERRPLAY;
        for( j = 0; j < uDelay; j++ ); /* ... and wait */
    }
    if( dsp_SetSpeaker( OFF ) ) return DSP_ERRPLAY;
    return NO_ERROR;
}

/*****
/* dsp_RECORD : Recording musical data in an array */
/*-----*/
/* Input   : pBuffer - Address of array that receives 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 */
/*****
WORD dsp_RECORD( PBYTE pBuffer, WORD uSize, WORD uDelay )
{ WORD i, j;

    /* DSP speaker must be switched off during recording! */
    if( dsp_SetSpeaker( OFF ) ) return DSP_ERRRECORD;
    for( i = 0; i < uSize; i++ )
    {
        /* Read... */
        if( dsp_8ADC( &pBuffer[ i ] ) ) return DSP_ERRRECORD;
        for( j = 0; j < uDelay; j++ ); /* ... and wait */
    }
    return NO_ERROR;
}

```



```

/* Which buffer part should be loaded/saved next ? */
INT iBufCnt = 0;

/*****
/* dsp_InitBuffers: Prepare buffer for DSP record/play          */
*****/
VOID dsp_InitBuffers( VOID )
{
    iBufCnt = 0;
}

/*****
/* dsp_FileOpen: Open file for DSP record/playback             */
*****/
/* Input   : pFile      - File name                             */
/*          : iADC       - Record (TRUE) / Playback (FALSE)     */
/*          : pHandle    - Address of INT for receiving the file handle */
/* Output   : 0          - no error                             */
/*          : <> 0       - occurred DOSERROR                     */
*****/
INT dsp_FileOpen( PCHAR pFile, INT iADC, PINT pHandle )
{
    if( iADC ) return _dos_creat( pFile, _A_NORMAL, pHandle );
    else      return _dos_open ( pFile, O_RDWR, pHandle );
}

/*****
/* dsp_FileClose: Close DSP Record/playback file               */
*****/
/* Input   : iHandle    - File handle                           */
/* Output   : 0          - no error                             */
/*          : <> 0       - occurred DOSERROR                     */
*****/
INT dsp_FileClose( INT iHandle )
{
    return _dos_close( iHandle );
}

/*****
/* dsp_ReadHeader : Read header                                 */
*****/
/* Input   : iHandle    - File handle                           */
/*          : pDRP       - Address of RECPLAY structure to receive header */
/* Output   : Number of read characters                         */
*****/
INT dsp_ReadHeader( INT iHandle, PDSPRECPLAY pDRP )
{
    WORD n;
    _dos_read( iHandle, &pDRP[ 0 ], sizeof( DSPRECPLAY ), &n );
    return n;
}

/*****
/* dsp_WriteHeader : Write header                               */
*****/
/* Input   : iHandle    - File handle                           */
/*          : pDRP       - Address of RECPLAY structure containing the */
/*          :             header                                     */
/* Output   : Number of written characters                     */
*****/
INT dsp_WriteHeader( INT iHandle, PDSPRECPLAY pDRP )
{
    WORD n;
    _dos_write( iHandle, &pDRP[ 0 ], sizeof( DSPRECPLAY ), &n );
    return n;
}

/*****
/* dsp_ReadBuffer : Read next block for playback               */
*****/
/* Input   : iHandle    - File handle                           */
/*          : lpBuffer   - Address of memory                     */
/*          : iHalfSize  - (Size of memory) / 2                 */
/* Output   : Number of read characters                         */
*****/
/* Note    : This function automatically keeps track of whether the */
/*          : first or second half of the buffer needs to be read. To */
/*          : do this, all the loaded buffers are counted. The      */

```

```

/*          decision as to which part of the buffer will be loaded          */
/*          with new data is made based on the "evenness" of this          */
/*          counter.                                                         */
/*****
INT dsp_ReadBuffer( INT iHandle, LPBYTE lpBuffer, INT iHalfSize )
{ WORD n;
  _dos_read( iHandle,
    ( LPVOID )&lpBuffer[ ( iBufCnt & 1 ) ? iHalfSize : 0 ],
    iHalfSize, &n );
  iBufCnt++;
  return n;
}

/*****
/* dsp_WriteBuffer : Save next block of recording                          */
/*****
/* Input   : iHandle   - File handle                                     */
/*          : lpBuffer  - Memory address                               */
/*          : iHalfSize - (Size of memory) / 2                         */
/* Output  : Number of written characters                               */
/*****
/* Note : This function automatically keeps track of whether the         */
/*         first half or second half of the buffer needs to be saved.    */
/*****
INT dsp_WriteBuffer( INT iHandle, LPBYTE lpBuffer, INT iHalfSize )
{ WORD n;
  _dos_write( iHandle,
    ( LPVOID )&lpBuffer[ ( iBufCnt & 1 ) ? iHalfSize : 0 ],
    iHalfSize, &n );
  iBufCnt++;
  return n;
}

/*****
/* dsp_ClearBuffer : Clear buffer                                         */
/*****
/* Input  : lpBuffer - Address of buffer                                 */
/*          : uMemSize - Size of buffer                                  */
/*****
/* Note   : After the last block to be played has been loaded this       */
/*           function is called to avoid an "Echo"                       */
/*****
VOID dsp_ClearBuffer( LPBYTE lpBuffer, WORD uMemSize )
{ WORD i;
  for( i = 0; i < uMemSize; i++ ) lpBuffer[ i ] = 0;
}

/*****
/* dsp_DOREcPlay : Harddisk-Recorder / Player                            */
/*****
/* Input  : iHandle   - File handle of file containing data to be       */
/*          :           played or file to receive data.                  */
/*****
/*          iADC       - Record (TRUE) or Playback (FALSE)              */
/*          : iASource  - Recording source (CD, LINE, MIC)               */
/*          :           -1: Use current mixer setting                    */
/*          : pDRP      - Header                                         */
/*          : iSecs     - Number of seconds to be recorded              */
/*          :           (ignored during playback)                       */
/*          : lpBuffer  - DMA-capable memory                             */
/*          : uMemSize  - Size of DMA memory                             */
/*****
VOID dsp_DoRECPLAY( INT iHandle,
                   INT iADC,
                   INT iSource,
                   PDSPRECPLAY pDRP,
                   INT iSecs,
                   LPBYTE lpBuffer,
                   UINT uMemSize )
{ INT iEncore;
  BYTE DSP_cmd;
  LONG lSamples;

  /* En - core!, En - core! */
  /* Executed command */
  /* Number of samples to record/play */

  pDRP->iStereo = pDRP->iStereo ? dsp_CanStereo() : FALSE;
  pDRP->iBits = ( pDRP->iBits == 16 ) ? dsp_MaxBits() : 8;
  dsp_AdjustFrq( &pDRP->uFrequency, iADC, &pDRP->iStereo );

```

```

dsp_InitIrqHandler();                /* Install DSP interrupt function */

dsp_InitBuffers();                  /* Initialize DMA double/Splitbuffering */
dsp_ClearBuffer( lpBuffer, uMemSize );

if( !iADC )                        /* Playback: Fill buffer */
{
    lSamples = filelength( iHandle ) - lseek( iHandle, 0L, SEEK_SET );
    dsp_ReadBuffer( iHandle, lpBuffer, uMemSize / 2 );
    dsp_ReadBuffer( iHandle, lpBuffer, uMemSize / 2 );
}
else lSamples = ( LONG ) pDRP->uFrequency * ( LONG )iSecs;

/*- DSP3xx -----*/
if( DSPBASE.uDspVersion < DSP_4XX )
{
    if( pDRP->iStereo ) mix3_PrepereForStereo( iADC ? ADC : DAC );

    if( iADC )                    /* Recording */
    {
        if( iSource >= 0 ) mix3_SetADCSrc( iSource );
        dsp_SetSpeaker( OFF );    /* When recording, always set DSP */
                                   /* speaker off */
                                   /* Mono or Stereo recording? */
        if( DSPBASE.uDspVersion >= DSP_3XX )
            dsp_Write( pDRP->iStereo ? DSP3_STEREOADC : DSP3_MONOADC );
                                   /* Use HI_SPEED DSP commands in stereo mode*/
                                   /* or with high speed transfers */

        DSP_cmd = ( BYTE )DSP_8DMAAUTOADC;
        if( DSPBASE.uDspVersion >= DSP_201 )
            if( pDRP->iStereo || dsp_IsHIMONODACFrq( pDRP->uFrequency ) )
                DSP_cmd = ( BYTE )DSP2p_8DMAHIAUTOADC;
    }
    else                          /* Playback */
    { /* DSP3XX */
        dsp_Write( DSP3_MONOADC );    /* Record mode always MONO */

        dsp_SetSpeaker( ON );        /* DSP speaker on */
        if( pDRP->iStereo )          /* Stereo playback */
        { char cBackUp; /* With stereo playback, first output 0x80 via */
                                   /* DSP using DMA. */
                                   /* replace first byte of DMA capable memory through 0x80 */
            cBackUp = lpBuffer[ 0 ];
            lpBuffer[ 0 ] = 0x80;
                                   /* Program DMA channel for outputting a byte*/
            dma_SetChannel( DSPBASE.iDspDmaB, lpBuffer, 1,
                            MODE_SINGLE | 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( NULL, 0 );
            lpBuffer[ 0 ] = cBackUp; /* Restore DMA buffer */
            DSP_cmd = DSP2p_8DMAHIAUTODAC;
                                   /* Stereo playback always HiSpeed */
        }
        else                      /* Mono playback */
        {
            DSP_cmd = ( BYTE )DSP_8DMAAUTODAC;    /* DSP_201 */
            if( dsp_IsHIMONODACFrq( pDRP->uFrequency ) )
                DSP_cmd = ( BYTE )DSP2p_8DMAHIAUTODAC;
        }
    }

    /* Program DMA for Record/Play */
    dma_SetChannel( DSPBASE.iDspDmaB, lpBuffer, uMemSize,
                    ( BYTE ) ( iADC ?
                                ( BYTE ) (MODE_SINGLE|MODE_AUTOINIT|MODE_WRITE) :
                                ( BYTE ) (MODE_SINGLE|MODE_AUTOINIT|MODE_READ) ) );

    dsp_SetFrq( &pDRP->uFrequency );
                                   /* Trigger an IRQ after each buffer half: */
    dsp_SetTransferSize( uMemSize / 2 );
    dsp_Write( DSP_cmd );          /* Send DSP command */
}

```

```

}
else
/*- DSP 4.XX -----*/
{ BYTE dsp4_Mode = 0;
  INT iDivisor = 2;
  if( iADC )
    switch( iSource )
    {
      case CD:
        mix4_SetADCSourceL( CD_L, TRUE ); /* With mono recordings */
        mix4_SetADCSourceR( CD_R, TRUE ); /* only left ADC is used */
        break;
      case LINE:
        mix4_SetADCSourceL( LINE_L, TRUE );
        mix4_SetADCSourceR( LINE_R, TRUE );
        break;
      case MIC:
        mix4_SetADCSourceL( MIC, TRUE );
        mix4_SetADCSourceR( MIC, TRUE );
        break;
    }
  if( !iADC ) mix4_SetVolume( VOICE, 255, 255 );
  else if( !pDRP->iStereo ) mix4_PrepareForMonoADC();
  /* Program DMA channel for 8 or 16 bits */
  dma_SetChannel( ( pDRP->iBits == 16 ) ? DSPBASE.iDspDmaW : DSPBASE.iDspDmaB,
    lpBuffer, uMemSize,
    ( BYTE )( iADC ?
      ( MODE_SINGLE | MODE_AUTOINIT | MODE_WRITE ) :
      ( MODE_SINGLE | MODE_AUTOINIT | MODE_READ ) ) );
  /* Set input/output frequency */
  if( iADC ) dsp4_ADCFrq( pDRP->uFrequency );
  else      dsp4_DACFrq( pDRP->uFrequency );

  /* Record/Playback specified by command byte */
  DSP_cmd = ( iADC ) ? DSP4_CMDADC : DSP4_CMDDAC;
  DSP_cmd |= DSP4_CMDAUTOINIT | DSP4_CMDFIFO;
  DSP_cmd |= ( pDRP->iBits == 16 ) ? DSP4_CMD16DMA : DSP4_CMD8DMA;

  dsp4_Mode = ( pDRP->iStereo ) ? DSP4_MODESTEREO : DSP4_MODEMONO;
  dsp4_Mode |= 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( pDRP->iBits == 16 ) iDivisor *= 2;
  dsp_Write( LOBYTE( ( uMemSize / iDivisor ) - 1 ) );
  dsp_Write( HIBYTE( ( uMemSize / iDivisor ) - 1 ) );
}

dsp_InitWaitForIRQ(); /* Initialize wait for IRQ */
do
{
  /* Wait for next IRQ */
  if( dsp_WaitForNextIRQ( NULL, 0 ) ) printf("Data lost!\n");
  printf(".");
  if( iADC )
  {
    iEncore = FALSE; /* do not continue? */
    if( dsp_WriteBuffer( iHandle, lpBuffer, uMemSize / 2 ) )
      if( lSamples > 0 )
      {
        iEncore = TRUE;
        lSamples -= uMemSize / 2;
      }
  }
  else iEncore = dsp_ReadBuffer( iHandle, lpBuffer, uMemSize / 2 );
} while( iEncore );
dsp_ClearBuffer( lpBuffer, uMemSize );

/* HIspeed commands can only be interrupted by a reset */
if( ( DSP_cmd == DSP2p_8DMAHIAUTODAC ) ||
    ( DSP_cmd == DSP2p_8DMAHIAUTOADC ) )
{
  dsp_RestoreIrqHandler(); /* install old IRQ handler */
  dsp_Reset();
}

```

```

else
{
    if( ( !pDRP->iStereo ) && ( DSPBASE.uDspVersion>= DSP_4XX ) )
        mix4_PrepateForMonoADC();

    if( ( DSP_cmd & 0xF0 ) == DSP4_CMD8DMA )
        dsp_Write( DSP4_EXIT8DMA );          /* End after current block */
    else
        if( ( DSP_cmd & 0xF0 ) == DSP4_CMD16DMA )
            dsp_Write( DSP4_EXIT16DMA );      /* End after current block */
        else
        {
            if( ( pDRP->iStereo ) && ( DSPBASE.uDspVersion <= DSP_3XX ) )
                mix3_RestoreFromStereo();
            dsp_SetSpeaker( OFF );              /* DSP speaker off */
            dsp_Write( DSP2p_EXITAUTOINIT );    /* End after current block */
        }
        dsp_WaitForNextIRQ( NULL, 0 );          /* Wait for end */
        dsp_RestoreIrqHandler();                /* install old IRQ handler */
    }
}
#endif
#endif
/* #ifndef DSP_VERSIONONLY */
/* #define __DSPUTIL_C */

```