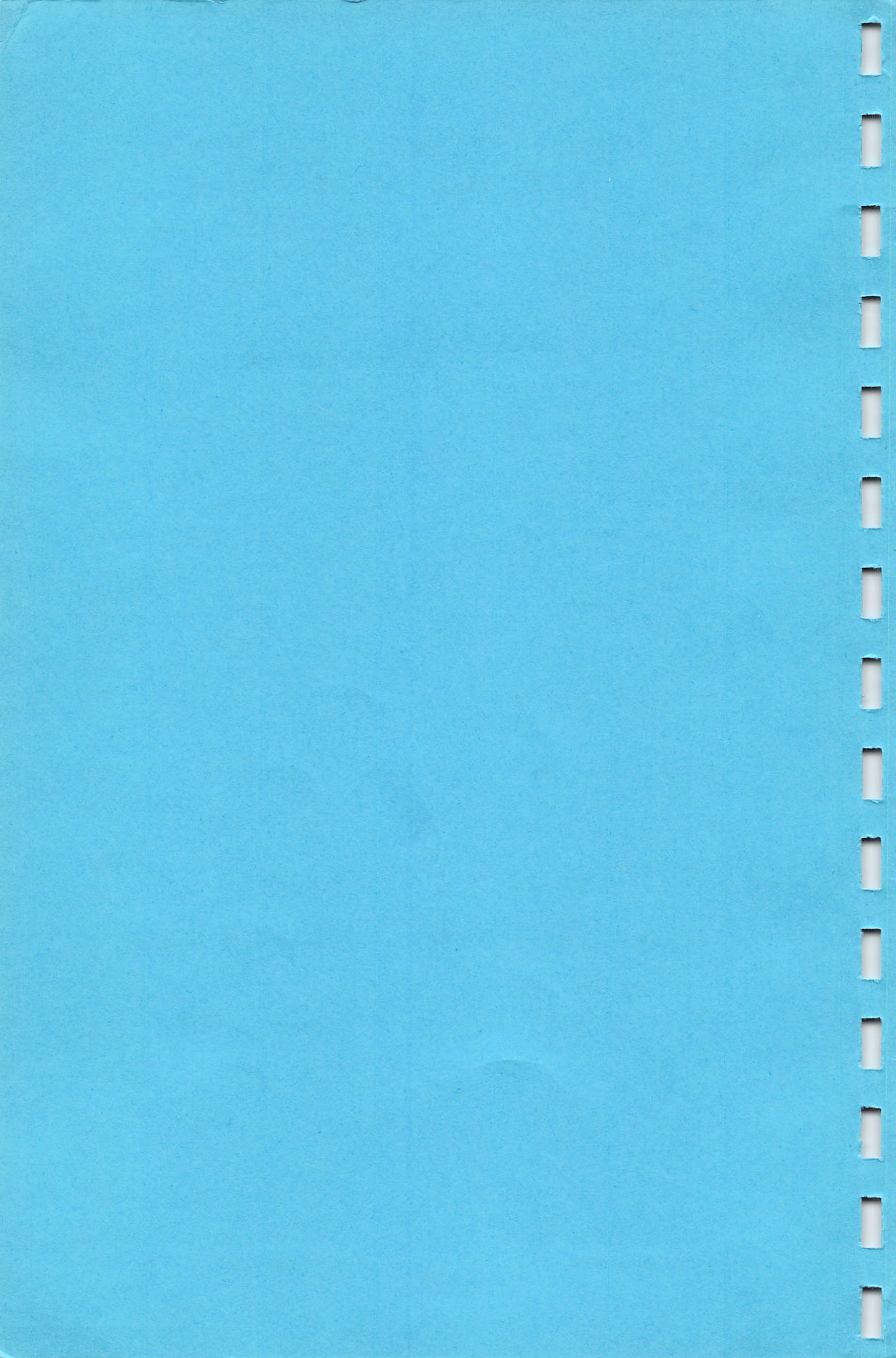
DSP Solutions Audio API Library v1.20

Interface Library Programmer's Manual

© DSP Solutions, Incorporated, 1993
August 12, 1993



Software Driver Source Code Licensing Agreement and Limited Warranty

DSP Solutions, Inc. warrants that its product will substantially conform to the performance specifications contained in this programmer's manual, provided that the product is used on the computer hardware and with an operating system for which it was designed. In the event of any defects in manufacture or failure to conform to published specifications, DSP Solutions will, at its option, repair or replace the defective product at no charge. Defective product must be received at a DSP Solutions facility within ninety (90) days from date of purchase, accompanied by proof of purchase. Shipping charges to DSP Solutions shall be prepaid by purchaser.

The foregoing warranty is exclusive and in lieu of all other warranties, express or implied, including but not limited to implied warranty for merchantability or fitness for a particular purpose. DSP Solutions expressly disclaims any and all liability for any incidental, special, or consequential damages arising from the use or inability to use its product, software, or documentation. In no case shall the liability of DSP Solutions exceed the cost of the product.

Some States do not allow the limitation or exclusion of implied warranties or general liability, so the above limitation or exclusion may not apply, sole by operation of the law.

Software License Agreement

PLEASE READ THIS AGREEMENT BEFORE OPENING THE ENVELOPE THAT CONTAINS THE DISKETTES. IF YOU DO NOT WISH TO COMPLY WITH THE TERMS AND CONDITIONS PRESENTED HERE, RETURN THE ENTIRE PRODUCT IN ITS ORIGINAL PACKAGING TO YOUR SUPPLIER.

By opening the envelope, you agree to the following terms and conditions:

www.DSPS.com

You may use this software to create software programs to be used exclusively with DSP Solutions Audio Adapters. The use of this software to create programs not capable of use with DSP Solutions Audio Adapters is not permitted under the terms of this License. The unauthorized use of this software shall be deemed a breach of the copyright in the software owned by DSP Solutions. You may use this software on a single computer, but may transfer it to another computer as long as it is used on only one computer at a time. You may copy the software for backup purposes only. You may merge the software into another program for your use on a single computer. You may include the software in your applications and distribute them at your discretion. You may NOT distribute any portion of the software not integrated into your application.

You may transfer this software and license to another party if the other party agrees to accept the terms and conditions of this license, and you either transfer or destroy all copies in any format in your possession. This includes all portions of the software contained or merged into other programs.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE SOFTWARE, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE.

This license is effective until terminated. You may terminate it at any time by destroying the software together with all copies, modifications, and merged portions in any form. The license also terminates if you fail to comply with the terms and conditions of this agreement.

This software and accompanying documentation are protected by United States Copyright law and also by International Treaty provisions. Any use of the software in violation of these laws constitutes termination of the license and can be prosecuted.

Table Of Contents

Introduction 1
Audio Data Formats 3
The DIGIAPI Library 7
The Sample Programs 8
- PLAY.C 9
- RECORD.C 11
- CALLBACK.C 14
- CALLIDLE.C 17
- MUSIC.C 19
- EMULATE.C 22

Programming Considerations 25
Utility Functions 28
- DSReset 29
- DSGetStatus 30
- DSUninstall 31
- DSQuery 32

Digitized Audio Control Functions 34
- DSGetByteCount 35
- DSSetBuffer 36
- DSSetGain 37
- DSPause 38
- DSResume 39
- DSSetCallback 40
- DSSetCallIdle 43

Audio Playback/Record Functions 45
- DSBPlay 46
- DSBPlayStereo 48
- DSBRec 49

Compatibility Functions 51
- Digispeech Format Functions 52
  - DSFPlayDigi 53
  - DSBPlayDigi 54
- CVSD Format Functions 55
  - DSFRecCVSD 56
  - DSFPlayCVSD 57
  - DSFRecUCVSD 58
  - DSFPlayUCVSD 59
  - DSBPlayCVSD 60
  - DSBRecCVSD 61
- PCM Format Functions 62
  - DSFPlayPCM 63
  - DSBPlayOKI 64

LPC Playback Functions 65
DSLPCStatus 66
DSFPlayLPC 67
DSFPlayULPC 68
DSBPlayLPC 69
DSBPlayULPC 70
Tone Generator Functions 71
DSToneStart 72
DSToneCmd 73
DSToneCmd - Set Attenuation 74
DSToneCmd - Set Frequency LSBs 75
DSToneCmd - Set Frequency MSBs 76
DSToneStop 77
Synthesizer Emulator Functions 78
DSGetEntry 79
DSEmulate 80
DSEmulate - SYNTH_INIT 81
DSEmulate - SYNTH_READ 82
DSEmulate - SYNTH_WRITE 83
DSEmulate - SYNTH_RESET 84
Synthesized Music Functions 85
DSInitSynthesizer 86
DSSetPatchTable 87
DSSetMusicMode 89
DSTranspose 90
DSMIDINoteOn 91
DSMIDINoteOff 92
DSMIDIPgmChange 93
High Level Functions 94
General 95
DriverInstalled 96
Digitized Audio 97
StartPlay 98
StartRecord 99
Tone Generator 100
Tone 101
PlayOneTone 102
PlayThreeTone 103
Synthesized Music 104
DSMusicInit 105
DSSetStatusAddress 106
DSMusicStatus 107
DSSetMusicRate 108

DSStartMusic 109
DSStopMusic 110
DSTerminateMusic 111
Appendix A - Data File Formats 112
Appendix B - DIGI.H & DIGIDRV.H 116
Appendix C - Detecting the Driver 121
Appendix D - SDK Files 123

Introduction

The DSP Solutions Audio API library provides a common high level language interface for the DSP Solutions LWDIGI and PDIGI audio drivers. The LWDIGI driver supports the DSP Solutions DS201 and DS201A Audio Adapter Units. The PDIGI driver supports the DS301 PORT•ABLE Sound and PORT•ABLE Sound Plus Units. Except where a distinction must be made, for ease of reading, this manual refers to both the DS201 and the DS201A as DS201A, the PORT•ABLE Sound and PORT•ABLE Sound Plus units as DS301, and the LWDIGI and PDIGI drivers as DIGI.

The functions in this manual apply to all DSP Solutions Audio Adapters, except where noted under the title bar of each function description. If the name of an Audio Adapter appears in square brackets, the function applies only to that unit (or units, if more than one specified). If no units are listed in the title bar, that function applies to all units.

The DSP Solutions drivers are Terminate-and-Stay-Resident (TSR) DOS programs. They can be loaded at the DOS command prompt prior to executing the application program, or they can be loaded via the AUTOEXEC.BAT file when the system is booted. In either case, the appropriate Audio Adapter Unit MUST be powered up and connected to the PC before executing the driver. If the driver does not detect the appropriate Audio Adapter Unit, it will not load.

A DSP Solutions driver consists of at least two files. The loader file (xx.EXE) scans the PC environment to determine appropriate loading conditions for the DS201A and driver. The driver file (xxx.DAT) is a TSR program loaded by the loader file. If more than one .DAT file is present, the appropriate file is chosen by the loader according to command line switches and the PC environment.

The DS301 PDIGI driver reads the DGSPEECH.INI file to determine the appropriate port, interrupt, and communications parameters for loading the .DAT file. The DGSPEECH.INI file can be located either in the same directory as the PDIGI.EXE file, or in the directory specified by the DGSPEECH environment variable. The contents of the DGSPEECH.INI file are created by the DGSETUP.EXE program, which tests the CPU, the parallel port, and other factors to determine the best communications parameters for the DS301 in that environment.

Audio Data Formats

DSP Solutions DIGI drivers support the following audio data formats.

DIGISPEECH Standard
DIGISPEECH Fixed
DIGISPEECH Realtime [DS301]

These proprietary compression formats are best used for high quality, natural-sounding voice reproduction. Audio is sampled at 16 bits, 8000 samples per second (sps) and compressed to average data rates of 1100, 1625, and 1650 bytes per second (bps), respectively.

Digispeech Standard and Fixed mode are supported for playback only. Recording of these formats is accomplished using the DSP Solutions Encoding Work Station product.

Digispeech Realtime compression format is similar to the Digispeech Fixed format, but allows real-time recording in addition to playback. It produces very good quality voice reproduction with approximately 4 to 1 compression.

PCM Linear
PCM μLaw
PCM aLaw
PCM Linear 16 bit [DS301]
PCM Stereo [DS301]

PCM Linear is the current industry standard for digitized audio. It consists of raw, non-compressed 8-bit digitized audio samples. It's advantages are that it provides relatively good sound reproduction with moderate storage space requirements, and is supported by nearly every sound device on the market. The μLaw and aLaw formats are United States and European telecommunications industry standards that sample data with logarithmic scaling, reducing background hiss.

LWDIGI supports recording at 8000 sps, and playback at 8000-11025,22050, and 44100 sps. PDIGI supports recording at 8000 and 11025 sps, 8 or 16 bit, and playback at 4000-44100 sps, 8 or 16 bit. PCM µLaw and aLaw are 8-bit only formats.

The DS201 and DS201A do not support aLaw recording. The DS201 supports only foreground playback and recording at 8000 sps.

The DS301 supports stereo playback of PCM Linear, PCM µLaw, and PCM Linear 16-bit formats at 11025, 22050, and 44100 sps.

CVSD

CVSD is a low to moderate quality compression format included for compatibility with the IBM PS/2 Speech Adapter, and used mainly in the education market. The DIGI drivers support both recording and playback of audio in CVSD format at six different rates, from 1,800 bps to 4,800 bps. It is recommended that Digispeech Realtime or DVI ADPCM compressions be used for new designs.

OKI ADPCM [DS201A][DS301]

This compression format is compatible with the 4-bit ADPCM format used by OKI Semiconductor MSM8205 and MSM5218 ADPCM Speech Synthesis integrated circuits. The sampling rate can be set to any value from 4,000 to 44,100 sps (DS301) or 8,000 to 11,025 sps (DS201A).

The DS201A supports OKI ADPCM playback only. Recording of this format is accomplished using the DSP Solutions Encoding Work Station product or the DS301.

DVI ADPCM [DS301]

This compression format is compatible with Intel corporation's DVI audio compression standard, and produces very good quality voice and music reproduction with approximately 4 to 1 compression. The sampling rate can be set to any value from 4,000 to 44,100 sps.

SB ADPCM 2-bit [DS301]
SB ADPCM 2.6 bit [DS301]
SB ADPCM 4 bit [DS301]

These compression formats are compatible with Creative Lab's Sound Blaster ADPCM formats. The sampling rate can be set to any value from 4,000 to 44,100 sps. These formats produce poor to moderate quality sound reproduction, and are included in the DS301 only for compatibility purposes. It is recommended that Digispeech Realtime or DVI ADPCM compressions be used for new designs.

LPC Speech Synthesis [DS201A][DS301]

LPC speech synthesis uses a very low data rate to produce poor but recognizable speech. It is included in the DS201A and DS301 for compatibility with the IBM PS/2 Speech Adapter. The DIGI drivers emulate the IBM PS/2 Speech Adapter LPC playback functions. The DIGI drivers implement the following LPC related functions of PS/2 Speech Adapter API (interrupt 4DH): AX=0200H, AX=0202H, AX=0300H, AX=0302H. More information about LPC functions can be found in "PS/2 Speech Adapter Technical Reference" by IBM, 68X2207 S68X-2207-00, January 1987.

Three Voice Tone Generator [DS201][DS201A]

The DS201 and DS201A Audio Adapters contain a three voice tone generator which emulates the Complex Sound Generator (SN76496N) device used in IBM PC-Junior and Tandy computers. The DIGI drivers provide BIOS-level compatibility for programs written for these machines via the INT 4D API.

Synthesized Music [DS301]

The PDIGI driver provides access to the DS301 music synthesizer via MIDI-level commands and also provides a Creative Labs .CMF file format compatible MIDI interpreter.

The DIGIAPI Library

The DSP Solutions Software Development Kit provides the application programmer with a library of function calls that interface with the DSP Solutions Audio Drivers. The library is compatible with the standard drivers for both the DS201/A and the PORT-ABLE Sound. See appendix D for a list of files on the SDK disk.

The DIGIAPI library does not use any compiler specific functions, and therefore can be linked with programs compiled for both Microsoft and Borland compilers. The library is supplied in SMALL and LARGE model versions. Both versions require FAR pointers for all functions requiring pointers. If you require a different programming model, please contact DSP Solutions for assistance.

The function descriptions in this manual each contain a function call summary, a **Description** section, a **Returns** section, and an **Implementation** section.

The function summary uses the 'C' language typedefs BYTE, WORD, and DWORD as follows:

```
typedef unsigned char    BYTE;
typedef unsigned int     WORD;
typedef unsigned long    DWORD;
```

The **Descriptions** section describes the basic operation of the function, and lists the values for each parameter, if applicable.

The **Returns** section lists the possible return values for that function. The return values are described in Appendix B and #defined in the DIGI.H header file.

The **Implementation** section lists the 80x86 register parameters for the function, including a mnemonic literal for the function code. The actual function codes can be found in Appendix B and in the DIGI.H header file.

The Sample Programs

The DSP Solutions Software Development Kit provides multiple examples of 'C' language applications that interface with the DIGIAPI library.

All sample programs included on the SDK disk were written using Microsoft C, v6.0a. Most of the 'C' programs have been tested with both the Microsoft and Borland compilers.

The following section provides a brief summary of each source example. The source code itself is contained in the corresponding files on the SDK disk.

PLAY.C

PLAY.C illustrates the use of the basic digitized audio functions for playback of audio files in various formats, both compressed and uncompressed. The program parses DSP Solution's file headers, but the code for playing digitized audio is modularized, and may be used with other file formats.

PLAY.C plays audio data in the background, allowing the user to control operation with the keyboard while sound is in progress. Background operation is accomplished via a double-buffering scheme, with the driver playing data from one buffer while the foreground process (the application), is filling the other buffer.

The *main* subroutine in PLAY.C supervises the operation. After signing on and parsing the command line, it checks to see if a DIGI driver is installed using the **DriverInstalled** function.

DSSetup

After opening the file passed to the PLAY.C program, *DSSetup* is called to initialize the driver for playback.

> **DSReset** places the driver in a known state.
>
> **DSSetBuffer** is called twice, once for each buffer. This application uses static buffers, but a typical application would dynamically allocate buffers to get the largest buffer for the available memory. The larger the buffer, the better, up to a maximum of 64K bytes. Large buffers allow more time for reading from disk and performing user I/O.
>
> **DSSetGain** is called to set a starting volume level. This call is not strictly necessary, as **DSReset** sets a default gain level, and may be left out if the default is acceptable.
>
> **DSQuery** determines the capabilities of the device and the driver for use by other routines.

ParseFileHeader
If the driver setup is successful, *ParseFileHeader* is called to determine the audio data format based on the file header. It also matches the format and sample rate with the current device and driver. If anything goes wrong, the program is aborted. If everything is OK, the data format and sample rate are displayed.

PlayManager
At this point, the *PlayManager* routine is called. After starting the background play operation, it sits in a loop and monitors the driver progress, filling each buffer from disk as it is used by the driver, and checking for user input.

> **StartPlay** is a higher-level DIGIAPI function used to start a background process. It is similar to the **DSBPlay** routine except that it works correctly with all data formats on all DIGI drivers. Typical applications will chose the low-level routine appropriate for the format or formats required for that application.
>
> **DSPause** disables the driver playback function temporarily. It returns E_IGNORED status if the driver was already in a paused state.
>
> **DSResume** restores the driver playback function after a **DSPause** call. It returns E_IGNORED status if the driver was not in a paused state.
>
> **DSGetStatus** returns the current state of the driver. While the playback operation is in progress, it will normally return only E_PAUSE, E_BUF0, or E_BUF1. E_BUF0 and E_BUF1 indicate the buffer currently being read by the driver. *PlayManager* uses this value in combination with it's bufferIndex variable to determine which buffer, if any, it should load from disk. E_PAUSE indicates, what else, pause state. Any other status is cause for ending the loop. E_OK indicates that we have played back for the length specified in the **StartPlay** invocation, all else is an error.

The *main* subroutine ends by resetting the driver, closing the

data file, and exiting.

RECORD.C

RECORD.C illustrates the use of the basic digitized audio functions for recording audio data in various formats, both compressed and uncompressed. The program writes DSP Solutions file headers, but the code for recording digitized audio is modularized, and may be used with other file formats.

RECORD.C records audio data in the background, allowing the user to control operation with the keyboard while recording is in progress. Background operation is accomplished via a double-buffering scheme, with the driver writing data to one buffer while the foreground process (the application), is writing the other buffer to a disk file.

The *main* subroutine in RECORD.C supervises the operation. After signing on and parsing the command line, it checks to see if a DIGI driver is installed using the **DriverInstalled** function.

Ctrl-Break actions are disabled so that the program cannot be exited without proper cleanup. If the application is interrupted while the DIGI driver is recording, it is possible for the next application executed to be overwritten by the driver. To avoid this problem, **DSReset** must be called before exiting the application.

DSSetup
After opening the file passed to the RECORD.C program, *DSSetup* is called to initialize the driver for recording.

> **DSReset** places the driver in a known state.
>
> **DSSetBuffer** is called twice, once for each buffer. This application uses static buffers, but a typical application would dynamically allocate buffers to get the largest buffer for the available memory. The larger the buffer, the better, up to a maximum of 64K bytes. Large buffers allow more time for writing to disk and performing user I/O.
>
> **DSSetGain** is called to set a starting volume level. This call is not strictly necessary, as DSReset sets a

default gain level, and may be left out if the default is acceptable.

DSQuery determines the capabilities of the device and the driver for use by other routines.

SetFormat
If the driver setup is successful, *SetFormat* is called to initialize the parameters for recording based on the user command line. It matches the selected format and sample rate with the current device and driver. If any incompatibilities are detected, the program is aborted. If everything is OK, the selected data format and sample rate are displayed.

WriteFileHeader
If the driver and recording parameter setups are successful, *WriteFileHeader* writes an initial file header, with place holders for the file length, if applicable. This routine is also called after the recording is complete, to update the header with the final file length. DSP Solutions file headers are defined in Appendix A, although we recommend the Windows .WAV file format for all new applications.

RecordManager
At this point, the *RecordManager* routine is called. This routine first prompts the user for a key to start recording. After starting the background record operation, it sits in a loop and monitors the driver progress, writing each buffer to disk as it is filled by the driver, and checking for user input.

StartRecord is a higher-level DIGIAPI function used to start a background process. It is similar to the **DSBRec** routine except that it works correctly with all recordable data formats on all DIGI drivers. Typical applications will chose the low-level routine appropriate for the format or formats required for that application.

DSPause disables the driver recording function temporarily. It returns E_IGNORED status if the driver was already in a paused state.

DSResume restores the driver recording function after a

DSPause call. It returns E_IGNORED status if the driver was not in a paused state.

DSGetStatus returns the current state of the driver. While the recording operation is in progress, it will normally return only E_PAUSE, E_BUF0, or E_BUF1. E_BUF0 and E_BUF1 indicate the buffer currently being written to by the driver. *RecordManager* uses this value in combination with it's bufferIndex variable to determine which buffer, if any, it should write to disk. E_PAUSE indicates, what else, pause state. Any other status is cause for ending the loop. E_OK indicates that we have recorded for the length specified in the **StartRecord** invocation, anything else is an error.

DSGetByteCount is called if the record loop is exited without an error status. It returns the number of bytes the driver wrote to the last buffer it was writing to. This buffer has not been written to disk, so we write the remainder here.

The *main* subroutine ends by resetting the driver, closing the data file, and exiting.

CALLBACK.C

CALLBACK.C illustrates the use of the **DSSetCallback** and related functions to implement a more sophisticated multiple buffering scheme. The concept is similar to the double-buffering scheme used in PLAY.C, but allows more than two buffers.

CALLBACK.C plays an audio data file in the background, allowing the user to control operation with the keyboard while sound is in progress. Background operation is accomplished via an eight buffer scheme, with the driver playing data from one buffer while the foreground process (the application), is filling another buffer. The driver and the foreground application rotate through all eight buffers in sequence.

This method is less efficient than a double-buffered scheme with two large buffers, but is more flexible.

CALLBACK.C is a much simpler program than PLAY.C. It accepts one file name on the command line. The file must be Linear 8 bit PCM at 8000 sps.

The *main* subroutine in CALLBACK.C performs all the file handling and background task management. First, it signs on and parses the command line.

> **DriverInstalled** is used to detect the presence of a DSP Solutions Audio driver.

It then opens the file specified on the command line.

> **DSQuery** is used to determine if the driver supports the callback function.

> **DSReset** places the driver in a known state.

The eight audio data buffers are filled from the contents of the first file, and the length of data in each buffer is saved. A flag is set for each buffer to indicate that it has been loaded. This is done even if no data was actually placed in the buffer.

DSSetBuffer is called twice, once each for the first two buffers in the list.

DSSetCallback is used to pass the address of our callback procedure to the DIGI driver. Note that we pass the address as a FAR pointer.

StartPlay is a higher-level DIGIAPI function used to start a background process.

DSPause disables the driver playback function temporarily.

DSResume restores the driver playback function after a **DSPause** call.

DSGetStatus returns the current state of the driver. The loop is exited based on the result of this call.

DSGetByteCount is called so that we can display the total number of bytes played to this point.

The *main* subroutine ends by resetting the driver, closing the data file, and exiting.

OurCallback

This function is the heart of the callback scheme. First, it establishes a local data segment, since it is being called from the driver, and the data segment is unknown. The address of the internal driver buffer info structure is taken from the ES:DI registers.

Next, it marks the previous data buffer as empty. It knows this because that's why it was called. The local index indicating the buffer that is currently being played is incremented and possibly wrapped around. The buffer that must be sent is calculated as the list buffer following the current buffer.

If the buffer to be sent has not been filled yet, an error flag is set, but life goes on. This routine is not too smart. There are many possible ways to handle this situation.

The routine now fills the internal driver structure using the address of the 'to be sent' buffer, and the accompanying format and sampling information. If the buffer is empty, we return a 0, if it contains data, we return a 1.

CALLIDLE.C

CALLIDLE.C illustrates the use of the **DSSetCallIdle** and related functions to implement a completion interrupt callback. The concept is similar to the double-buffering scheme used in PLAY.C, but performs a special callback to a application supplied routine when the operation is truly complete.

CALLIDLE.C plays an audio data file in the background, half at 8000 sps, and half at 11025 sps, allowing the user to control operation with the keyboard while sound is in progress. Background operation is accomplished via the standard double-buffered method, with the driver playing data from one buffer while the foreground process (the application), is filling the other buffer.

CALLIDLE.C is a much simpler program than PLAY.C. It accepts one file name on the command line. The file must be Linear 8 bit PCM at 8000 sps. The file should be at least 29000 bytes long.

The *main* subroutine in CALLIDLE.C performs all the file handling and background task management. First, it signs on and parses the command line.

> **DriverInstalled** is used to detect the presence of a DSP Solutions Audio driver.

It then opens the file specified on the command line.

> **DSQuery** is used to determine if the driver supports the Idle callback function.

> **DSReset** places the driver in a known state.

The four audio data buffers are filled from the contents of the data file.

DSSetBuffer is called twice, once each for the first two buffers in the list.

DSSetCallback is used to pass the address of our callback procedure to the DIGI driver. Note that we pass the address as a FAR pointer.

StartPlay is a higher-level DIGIAPI function used to start a background process.

DSPause disables the driver playback function temporarily.

DSResume restores the driver playback function after a **DSPause** call.

DSGetStatus returns the current state of the driver. The loop is exited based on the result of this call.

DSGetByteCount is called so that we can display the total number of bytes played to this point.

The *main* subroutine ends by resetting the driver, closing the data file, and exiting.

OurCallIdle

This function is the heart of the callback scheme. First, it establishes a local data segment, since it is being called from the driver, and the data segment is unknown.

If this is the first time the routine has been called, it resets the audio adapter, otherwise, no action is taken.

Since **DSReset** clears the callback state, **DSSetCallIdle** must be re-issued. **DSSetBuffer** is called to select the second set of audio data buffers loaded by the main application. **DSStartPlay** begins playback of the second set of buffers at the 11025 sample rate.

MUSIC.C

MUSIC.C illustrates the use of the Synthesized Music functions combined with the basic digitized audio functions.

MUSIC.C first plays a set of scales using a default synthesizer instrument. It then plays audio data in the background, allowing the user to control operation with the keyboard while sound is in progress. During digitized audio playback, semi-random synthesized notes are played to illustrate the combination of synthesized and digitized audio.

This program is derived from the PLAY.C program, therefore, the digitized audio functionality will not be described here.

The *main* subroutine in MUSIC.C supervises the operation. After signing on and parsing the command line, it checks to see if a DIGI driver is installed using the **DriverInstalled** function.

DSSetup
After opening the digitized audio file passed to the MUSIC.C program, *DSSetup* is called to initialize the driver for playback of digitized and synthesized audio.

> **DSInitSynthesizer** initializes the driver synthesizer in preparation for the **DSMIDI<xxx>** calls.
>
> **DSMIDIPgmChange** is called to load channel 0 with default instrument 3.

PlayScales

PlayScales contains a loop that plays 8 octaves of 12 notes each; the entire range of the synthesizer. The scales may be interrupted by pressing any key.

> **DSMIDINoteOn** is used to play the selected note by converting the note and octave values to a 1 of 128 pitch value. An intermediate velocity of 65 is used for every note. Channel 0 is used since the DSSetup routine loaded an instrument patch on that channel.
>
> After a delay of approximately a quarter-second, **DSMIDINoteOff** is used to turn the note off. The same channel and pitch values are sent to correctly identify the note to be turned off.

ParseFileHeader

ParseFileHeader is called to determine the digitized audio data format based on the file header. Since the driver only allows mixing synthesized audio with 8-bit linear and muLaw PCM, these are the only file types validated. If everything is OK, the data format and sample rate are displayed.

PlayManager

At this point, the *PlayManager* routine is called to manage digitized audio playback. The only differences in this routine from it's original in PLAY.C is in the pause and resume handling, and the addition of code to play random synthesized music notes. The functions added are described below.

> *Pause*
>
> This function replaces the simple call to **DSPause** found in PLAY.C. Before calling **DSPause**, a command is sent to turn off the current note. Then *Pause* waits a short period of time to ensure that the note off command is received by the driver before calling the **DSPause** function. If the routine did not wait, the **DSPause** might take effect before the note off command was processed, in some cases causing a continuing tone from the synthesizer during the pause.

Resume
This function contains exactly the same logic as the PLAY.C routine, but was moved to a subroutine for symmetry with the *Pause* function.

PlayRandom
Turns off the previous note played, then sets a new, semi-random, note and octave and sends the command to play the note.

The *main* subroutine ends by resetting the driver, closing the data file, and exiting.

EMULATE.C

EMULATE.C illustrates the use of the Synthesizer Emulator functions combined with the basic digitized audio functions.

EMULATE.C first plays a set of scales using an internally generated synthesizer instrument. It then plays audio data in the background, allowing the user to control operation with the keyboard while sound is in progress. During digitized audio playback, semi-random synthesized notes are played to illustrate the combination of synthesized and digitized audio.

This program is derived from the PLAY.C program, therefore, the digitized audio functionality will not be described here.

The *main* subroutine in EMULATE.C supervises the operation. After signing on and parsing the command line, it checks to see if a DIGI driver is installed using the **DriverInstalled** function.

DSSetup
After opening the digitized audio file passed to the EMULATE.C program, *DSSetup* is called to initialize the driver for playback of digitized and synthesized audio.

> **DSGetEntry** is used to obtain the entry pointer to the synthesizer emulator. The result is stored in the local **Emulate** pointer.
>
> **Emulate** is called to initialize the emulator.
>
> *LoadInstrument*
> Calls **Emulate** multiple times to load channel 0 of the emulator with the instrument table values from the testInstrument structure.

PlayScales

PlayScales contains a loop that plays 8 octaves of 12 notes each; the entire range of the synthesizer. The scales may be interrupted by pressing any key.

> *PlayNote*
>
> Performs a table lookup for the appropriate note frequency values, then calls **Emulate** to load the synthesizer frequency registers. The second call to **Emulate** also adds the key-on bit to sound the note.

After a delay of approximately a quarter-second, *PlayNote* is called for the next note, without actually turning the note off, which causes a smoother sequence through the notes.

ParseFileHeader

ParseFileHeader is called to determine the digitized audio data format based on the file header. Since the driver only allows mixing synthesized audio with 8-bit linear and muLaw PCM, these are the only file types validated. If everything is OK, the data format and sample rate are displayed.

PlayManager

At this point, the *PlayManager* routine is called to manage digitized audio playback. The only differences in this routine from it's original in PLAY.C is in the pause and resume handling, and the addition of code to play random synthesized music notes. The functions added are described below.

> *Pause*
>
> This function replaces the simple call to **DSPause** found in PLAY.C. Before calling **DSPause**, a command is sent to turn off the current note. Then *Pause* waits a short period of time to ensure that the note off command is received by the driver before calling the **DSPause** function. If the routine did not wait, the **DSPause** might take effect before the note off command was processed, in some cases causing a continuing tone from the synthesizer during the pause.

Resume
This function contains exactly the same logic as the PLAY.C routine, but was moved to a subroutine for symmetry with the *Pause* function.

PlayRandom
Sets a new, semi-random, note and octave and calls *PlayNote* to play the note.

The *main* subroutine ends by resetting the driver, closing the data file, and exiting.

Emulate is called a final time, with the SYNTH_RESET function code, to stop the synthesizer. This is necessary because the emulator functions are independent from the digitized audio driver functions.

Programming Considerations

- The DSP Solutions drivers are Terminate-and-Stay-Resident (TSR) DOS programs. The application program should verify that the TSR driver is resident in memory before any TSR function is called.

  A call to *ANY* TSR that is not resident in memory may result in a system crash.

- When a DIGI driver is loaded into memory, the audio unit is initialized and tested. The driver is installed as a TSR program only after the hardware tests have been completed successfully. If the TSR driver is successfully loaded, it returns to DOS with an exit code of 0. Otherwise it is not loaded and the exit code is 255.

- When the DIGI loader is executed, it checks if a DSP Solutions driver for either device family is already installed in the system. If such a driver exists, the name and version of the driver is displayed, installation is canceled, and the loader exits to DOS with an exit code of 0 (the same as a successful installation).

- During all LWDIGI **foreground** playback and recording routines, keyboard interrupts are disabled to prevent sound distortion. The keyboard interrupt is not disabled for background operations.

- Background recording and playback are based on a double buffer technique. Before starting any background playback or record process, two buffers must be defined using the **DSSetBuffer** function.

- Buffers for both foreground and background operations must not cross 64K segment boundaries.

- If a playback or recording routine is called when the driver is busy (playing or recording a prior request), the driver will ignore the command and return an error code.

- Speech files compressed in Digispeech Standard Format are limited to a total playback time of 1048 seconds (17.5 minutes).

- During background playback or recording, the application is restricted in using system calls which have a potential of frequently disabling system interrupts for relatively long periods of time (more than 50-200 microseconds). The Audio Adapters have an internal input buffer, used to store data sent by computer. If this buffer becomes empty prematurely during playback, sound can be distorted. Operations that can distort sound during background playback or recording are listed below:

    - Routines that disable interrupts for long periods of time.

    - Changing graphics or text mode during background playback or recording (most graphics functions are still allowed).

    - Using modified system timer (INT 08H) or timer tick interrupt (INT 1CH) handlers, which can disable interrupts for long periods of time.

    - Using mouse driver calls during CVSD recording in background.

    - Excessive disk or CD-ROM accesses. Double buffers should be set up that are sufficient to support the sound unit during periods of disk access.

Failure to follow the above restrictions can result in degradation of the sound quality reproduced by the DSP Solutions Audio Adapter.

- If the DIGIAPI interface library is not used, the DIGI drivers may be called directly by the application program. Most of the library calls in this manual have direct equivalents in driver functions. The **implementation** section of each library description indicates the registers that are used to pass parameters to and from the driver functions. The calling program must load the appropriate registers and call interrupt 0x4D. Unless otherwise indicated, the routines return an integer status code in the AX register. See appendices for error code descriptions and function codes.

  Ex.
  ```
  mov     ax, 0x0403        ; Set Audio Adapter Gain
                            ; function code
  mov     bx, gain          ; 0 - 31 gain value
  int     04Dh              ; Call driver via
                            ; software interrupt
  mov     status, ax        ; Save return status
  ```

  - Note that whenever a pointer is required, the user must pass a full FAR pointer (segment and offset) to the driver function.

Utility Functions

This section describes the low-level control and status functions of the Audio Adapter Unit that are common to all modes of operation.

DSReset

```
#include "digi.h"

int DSReset( void );
```

Description:

Terminates any current operation, including background operations, sets default audio gain, and resets the Audio Adapter hardware. Does not affect buffer definitions (see **DSSetBuffer**). Also clears pause state (see **DSPause**) and error state.

Returns: E_OK

Implementation:

```
AX -            AX_RESET
```

DSGetStatus

#include "digi.h"

WORD DSGetStatus(void);

Description:

Returns status of the DSP Solutions driver. The routine returns E_OK if the driver is idle and no errors have occurred. **DSGetStatus** can be called at any time and does not affect background functions.

DSGetStatus is an integral part of background operation. When called while a background operation is in progress the return value will indicate which buffer is currently in use or if the background operation is paused.

Returns: E_OK, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_MUSIC

Implementation:

```
AX -            AX_STATUS
```

DSUninstall

#include "digi.h"

int DSUninstall(void);

Description:

Resets the Audio Adapter Unit and removes the DSP Solutions driver from memory. The DSP Solutions driver should be the last driver (TSR) loaded for this function to work properly. No calls to the driver should be made after this routine returns.

The **DSUninstall** routine is called by the DSP Solutions loader program when the /U parameter is specified. Application programs may call this directly if TSR unloading rules are followed properly.

Returns: E_OK

Implementation:

```
AX -          AX_UNINSTAL
```

DSQuery

#include "digi.h"

int DSQuery (WORD _far *majVer, WORD _far *minVer, DWORD _far *caps)

majVer-	FAR pointer to major ID value.
minVer-	FAR pointer to minor ID value.
caps -	FAR pointer to capabilities flag word

Description:

After the **DSQuery** call, *majVer* contains the driver major version number, *minVer* contains the driver minor ID number. The *caps* variable contains the driver capabilities flag word in the lower 16 bits and the driver ID number in the upper sixteen bits.

Specifying a NULL pointer for any of the three parameters causes **DSQuery** to skip loading that value.

The *caps* word is a 32-bit flag word that may be tested with the following values.

Flag value	**Supports**
CAPS_EPCM	Extended PCM.
CAPS_LPC	LPC speech synthesis
CAPS_TONE	3-voice tone generator
CAPS_201A	DS201A
CAPS_ADPCM	OKI ADPCM
CAPS_CALLB	Callback function
CAPS_LPCVOC	LPC vocabulary
CAPS_PCMREC	PCM recording
CAPS_DRIVERID	Driver ID in upper 16 bits
CAPS_301	If this bit is set, the following bits are also valid:
CAPS_MIDI	MIDI Music translation.
CAPS_DIGI	Digispeech formats
CAPS_CALLIDLE	IDLE callback function

The upper 16 bits of the flag word contain the driver ID code. This code is available only on newer DSP Solutions drivers (CAPS_DRIVERID == 1). All other drivers return 0 for driver ID. Current ID codes are:

DID_UNK	Unknown driver (early version)
DID_201	LWDIGI DS201 driver (new version)
DID_201A	LWDIGI DS201A driver (new version)
DID_301	PDIGI Digital Audio only
DID_301M	PDIGI Digital Audio / MIDI Synthesizer
DID_301V	PDIGI Digital Audio / LPC Vocabulary
DID_301MV	PDIGI All features
DID_301TD	PDIGI Digispeech format only.

Returns: E_OK, E_UNKNOWN

Implementation:

```
AX -            AX_QUERY
```

Digitized Audio Control Functions

This section describes functions that control the digitized audio capabilities of the Audio Adapter.

DSGetByteCount

#include "digi.h"

```
int DSGetByteCount( DWORD _far *total );
```

total - FAR pointer to total byte count of operation.

Description:

Returns the number of bytes which were played back or recorded from/into the buffer currently in use by the background operation. If the *total* parameter is not a NULL pointer, the long integer it points to is loaded with the total number of bytes played/recorded since the background operation was started.

The return value is the current byte index into the active buffer, and can be used at the end of a record operation to determine the number of bytes to flush from the last active buffer.

The *total* value can be used to synchronize external events with the audio playback. The value starts at zero and increases with time, until the final count specified with the original play command is reached or the operation is aborted via a call to **DSReset**.

Returns: See description

Implementation:

```
AX -        AX_GETCNT
```

DSSetBuffer

#include "digi.h"

int DSSetBuffer(int bufferIndex, int len, char _far *buf);

bufferIndex -	Index of buffer being assigned (0 or 1).
len -	Length of buffer being assigned.
buf -	FAR pointer to buffer being assigned.

Description:

The **DSSetBuffer** routine is used to assign externally allocated buffers for background operation. **DSSetBuffer** must be called twice before calling any other driver background function. Buffer length can be as small as 2 bytes (16 bytes for LWDIGI), but is recommended to be at least 256 bytes. Typical buffer length is 8,192.

To redefine a buffer, call this function again with new buffer parameters. A buffer cannot be redefined when a driver operation is in progress (even if paused).

During a background operation, the two buffers are used alternately - one by the background process (driver) and the other by the foreground (the application program). **DSStatus** can be called by the foreground process to determine which buffer is currently used by background process, so that the foreground process can load or save the other buffer.

See **DSSetCallback** for alternative background methods.

Returns: E_OK, E_INDEX, E_LENGTH, E_PAUSE, E_BUF0,
E_BUF1, E_COM, E_MUSIC

Implementation:

```
AX -        AX_SETBUF
BX -        buffer index
CX -        buffer length
```

SI - buffer offset

DS - buffer segment

DSSetGain

#include "digi.h"

int DSSetGain(int gain);

gain - Audio volume.

Description:

Set new gain value for playback (DS201,DS301) operation or recording (DS301 only) operation. Scale value 0 is silence, 31 is maximum volume. Default value is 21. The **DSSetGain** function can be called before or after starting a playback operation.

When **DSSetGain** is called during background playback of CVSD files using the LWDIGI (DS201) driver, the sound volume does not change until playback of the next message.

DSSetGain has no effect on the tone generator. The output sound volume of the tone generator can be changed using the **DSToneCmd** function.

Returns: E_OK, E_GAIN, E_PAUSE

Implementation:

```
AX -        AX_SETGAIN
BX -        gain
```

DSPause

```
#include "digi.h"

int DSPause( void );
```

Description:

If a background operation is in progress, it will be paused. Otherwise there is no effect. Use the **DSResume** function to resume the operation. This function also works for the synthesized music interpreter (DS301).

Returns: E_OK, E_IGNORED, E_COM, E_MUSIC

Implementation:

```
AX -          AX_PAUSE
```

DSResume

#include "digi.h"

int DSResume(void);

Description:

If a background operation is currently paused, it will be resumed. Otherwise there is no effect. Use the **DSPause** function to pause a background operation. This function also works for the synthesized music interpreter (DS301).

Returns: E_OK, E_IGNORED, E_COM, E_MUSIC

Implementation:

```
AX -            AX_RESUME
```

DSSetCallback

```
#include "digi.h"

int DSSetCallback( void _far *routine );
```

routine - FAR pointer to user callback routine.

Description:

Establish a user callback function for background playback operation. The callback routine is used to change a buffer. During background operations, the DIGI driver calls the specified routine after switching buffers. The pointer must be a FAR pointer to a LARGE model routine (i.e. the routine must execute a FAR return when finished).

Normally, background operation is carried out using the double-buffering technique supported by the general functions of the driver. However, the callback mechanism can be used to implement more sophisticated multiple buffering schemes.

The **DSReset** function disables the callback mechanism and cancels the effect of the previous **DSSetCallback** call. The callback mechanism is disabled by default. Since **DSReset** disables the callback mechanism, the **DSSetCallback** function must be called just before calling any of the background play or record functions. It is recommended to call **DSReset** after the operation ends.

The application must establish two audio buffers in the standard way, using **DSSetBuffer** calls. These two buffers will be the first two buffers used during the operation. Subsequent buffers must be defined by the callback function.

The application's callback function is passed a FAR pointer to the following internal driver structure in the processor ES:DI registers.

```
struct
  {
  BYTE far *ptr;     // Sampled data buffer address.
  int  bufLength;    // Length of buffer (bytes).
  } bufferCtrl;
```

The callback function is expected to load this structure with the address and length of an audio data buffer. This buffer will be switched to at the completion of the current buffer.

The callback function must return 1 in AX if more data is to be played back or recorded. It should return 0 in AX and set the length member of `bufferCtrl` to 0 when no more data is available or required.

The driver plays the audio buffers in the following order after any background start function is called:

1) Buffer #0, (**DSSetBuffer** call). Callback call #1.
2) Buffer #1, (**DSSetBuffer** call). Callback call #2.
3) Buffer set by callback call #1. Callback call #3.
4) Buffer set by callback call #2. Callback call #4.
5) Buffer set by callback call #3. Callback call #5.
6) etc.

The callback function is called for the first time when the driver switches from buffer (1) to (2), as defined above. When driver switches from (2) to (3), the callback function is called again, etc.

If the callback function is unable to setup a new buffer address for some reason, the driver will alternate between the two most recently defined buffers.

Note that when the callback function modifies the `bufferCtrl` structure, it also alters the driver's internal definition of one of the two buffers defined during the **DSSetBuffer** calls. Therefore, it is recommended to redefine the buffers (by using **DSSetBuffer**) before the next background operation is started.

The callback function must load the DS register before accessing its private data segment. It can use any CPU registers without saving them. Since the callback function is called from inside a hardware interrupt handler, it should not call any DOS or BIOS functions. The callback function is allowed to do direct memory manipulations. If the callback function uses more than few bytes of stack, it should switch to its private stack. Interrupts are enabled when the callback function is called, but the run time should be kept to a minimum to prevent stack overflow, reentrant calls, etc.

The C program source in file CALLBACK.C, supplied with the Developer's Kit can be used as an example of an application using the callback mechanism.

Returns: E_OK, E_UNKNOWN

Implementation:

```
AX -        AX_CALLBACK
DX -        Callback routine offset
DS -        Callback routine segment
```

DSSetCallIdle

#include "digi.h"

int DSSetCallIdle(void _far *routine);

routine - FAR pointer to user Idle callback routine.

Description:

Establish a user callback function for background playback operation. The callback routine is used to call an application supplied routine when the playback or recording operation is complete. During background operations, the DIGI driver calls the specified routine after the last buffer has been played. The pointer must be a FAR pointer to a LARGE model routine (i.e. the routine must execute a FAR return when finished).

The **DSReset** function disables the callback mechanism and cancels the effect of the previous **DSSetCallIdle** call. The callback mechanism is disabled by default. Since **DSReset** disables the callback mechanism, the **DSSetCallIdle** function must be called just before calling any of the background play or record functions. It is recommended to call **DSReset** after the operation ends.

The driver ignores any return value.

The callback function must load the DS register before accessing its private data segment. It can use any CPU registers without saving them. Since the callback function is called from inside a hardware interrupt handler, it should not call any DOS or BIOS functions. The callback function is allowed to do direct memory manipulations. If the callback function uses more than few bytes of stack, it should switch to its private stack. Interrupts are enabled when the callback function is called, but the run time should be kept to a minimum to prevent stack overflow, reentrant calls, etc.

The C program source in file CALLIDLE.C, supplied with the Developer's Kit can be used as an example of an application using the callback mechanism.

Returns: E_OK, E_UNKNOWN

Implementation:

```
AX -        AX_CALLIDLE
DX -        Callback
DS -        Callback routine segment
```

Audio Playback/Record Functions

The routines in this section provide access to the background recording and playback functions of the DIGI drivers. They should be used for all new application designs. The routines do not support the older DS201 audio adapter.

DSBPlay

#include "digi.h"
[DS201A][DS301]

```
int DSBPlay( int format, WORD rate, DWORD len );
```

format -	audio data format.
rate -	Sample rate in samples per second.
len -	Number of bytes to be played.

Description:

Starts background playback of audio data. Background playback is accomplished via a double-buffering scheme. Sampled data is read from one buffer while the other buffer is loaded by the application program.

Background buffers are set up using the **DSSetBuffer** function. The **DSGetStatus** routine is used to determine the buffer from which **DSBPlay** is currently drawing data.

Total playback length is determined by the *len* parameter. The playback operation is terminated by the driver after *len* bytes are sent to the audio device. Background operation may be aborted using the **DSReset** function. If a callback routine is defined (see **DSSetCallback**), it is called after the driver switches to each new buffer.

The *rate* parameter specifies the playback sampling rate:

DS201A - 8000 to 11025 samples per second.
22050 sps (DF_PCM8, DF_PCMU, DF_PCMA)
44100 sps (DF_PCM8, DF_PCMU, DF_PCMA)
DS301 - 4000 to 44100 samples per second.

DS201A playback rates of 22050 and 44100 for the DF_PCM8, DF_PCMU, and DF_PCMA formats is accomplished via sample-skipping by the LWDIGI

driver. Audio data is actually played at a rate of 11025 sps.

DF_DIGIREAL, DF_DIGISTD, and DF_DIGIFIX formats are played at a sampling rate of 8000 sps, and ignore the *sampleRate* parameter.

The *format* parameter specifies the data format as follows.

	DF_PCM8	0
	DF_PCMMU	1
	DF_PCMA	2
	DF_PCM16	3
[DS301]	DF_SB2	4
[DS301]	DF_SB3	5
[DS301]	DF_SB4	6
[DS301]	DF_OKI4	7
	DF_DVI4	8
[DS301]	DF_DIGIREAL	9
[DS301]	DF_DIGISTD	10
	DF_DIGIFIX	11
	DF_LPC	12

Returns: E_OK, E_UNDEFINED, E_ARGUMENT, E_LENGTH, E_NOBUFFER, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -        AX_BLAY
BX -        data format code
CX -        Data length low word
DX -        Data length high word
SI -        Playback sampling rate
```

DSBPlayStereo

#include "digi.h"
[DS301]

int DSBPlayStereo(int format, WORD rate, DWORD len);

format -	audio data format.
rate -	Sample rate in samples per second.
len -	Number of bytes to be played.

Description:

Starts background playback of audio data in stereo mode. This routine operates the same as the **DSBPlay** function except that the number of data formats and sampling rates supported is reduced.

The stereo data must be interleaved single samples, in left channel - right channel order.

The *rate* parameter specifies a playback sampling rate of 11025, 22050, or 44100 sps.

The *format* parameter specifies the data format as follows.

DF_PCM8	0
DF_PCMMU	1
DF_PCM16	3

Returns: E_OK, E_UNDEFINED, E_ARGUMENT, E_LENGTH, E_NOBUFFER, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -    AX_BPSTEREO
BX -    data format code
CX -    Data length low word
DX -    Data length high word
SI -    Playback sampling rate
```

DSBRec

```
#include "digi.h"
        [DS201A][DS301]

int DSBRec( int format, WORD rate, DWORD len );
```

format -	audio data format.
rate -	Sample rate in samples per second.
len -	Number of bytes to be recorded.

Description:

Starts background recording in specified format. Background recording is accomplished via a double-buffering scheme. Sampled data is written to one buffer while the second buffer is being saved by the application program.

Background buffers are set up using the **DSSetBuffer** function. The **DSGetStatus** routine is used to determine the buffer to which **DSBRecord** is currently writing data.

Total recording length is determined by the *len* parameter. The recording operation is terminated by the driver after *len* bytes are received from the audio device. Background operation may be aborted using the **DSReset** function. If a callback routine is defined (see **DSSetCallback**), it is called after the driver switches to a new buffer.

The *rate* parameter specifies a recording sampling rate of 8000 or 11025 sps (DS301) or 8000 sps (DS201A).

DF_DIGIREAL format is recorded at a sampling rate of 8000 sps, and ignores the *rate* parameter.

The *format* parameter specifies the data format as follows.

DF_PCM8 0
DF_PCMMU 1
DF_PCMA 2
[DS301]
DF_PCM16 3
[DS301]
DF_SB2 4
[DS301]
DF_SB3 5
[DS301]
DF_SB4 6
[DS301]
DF_OKI4 7
[DS301]
DF_DVI4 8
[DS301]
DF_DIGIREAL 9
[DS301]

Returns: E_OK, E_UNDEFINED, E_ARGUMENT, E_LENGTH, E_NOBUFFER, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -        AX_BREC
BX -        data format code
CX -        Data length low word
DX -        Data length high word
```

Compatibility Functions

The routines in this section are provided for backwards compatibility with older DSP Solutions LWDIGI drivers (v1.21A and under). It is recommended that the **DSBPlay** function be used for new designs.

Digispeech Format Functions

The Digispeech Standard Format is based on a novel algorithm for sound compression. The algorithm compresses audio to an average rate of 1100 bytes per second, while preserving high audio quality and natural sounding voices.

Audio compressed in this format can be played back as a background process, since only a small part of the processing power of the CPU is required (e.g. less than 10% for an 8MHz 8088 CPU).

The compression of audio to Digispeech Standard Format is accomplished by the DSP Solutions Encoding Work Station.

DSFPlayDigi

#include "digi.h"

int DSFPlayDigi(char _far *buf, WORD len);

buf -	FAR pointer to sampled data buffer.
len -	Number of bytes in buffer to be played.

Description:

Foreground playback of Digispeech STD or FIX (movie) data formats. Also plays 8-bit PCM formats. Sampled data is read from *buf*, and is limited to a maximum of 65535 bytes, as specified by the *len* parameter.

The data format is determined by the data file header. The entire header must be passed to the driver as part of the sampled data. See appendix A for data file headers.

The routine returns control to the calling program only when playback is completed. Keyboard interrupts are disabled during foreground playback operations to prevent sound distortion.

Returns: E_OK, E_LENGTH, E_PAUSE, E_BUF0, E_BUF1,
E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -        AX_FPDIGI
CX -        buffer length
SI -        buffer offset
DS -        buffer segment
```

DSBPlayDigi

#include "digi.h"

int DSBPlayDigi(DWORD len);

len - Total length in bytes of sampled data to be played

Description:

Background playback of Digispeech format data. Background playback is accomplished via a double-buffering scheme. Sampled data is read from one buffer while the second buffer is being loaded by the application program. The buffers are set up via the **DSSetBuffer** function. The **DSGetStatus** routine is used to determine the buffer from which the **DSBPlayDigi** routine is currently drawing data. Background operation may be aborted using the **DSReset** function. If a callback routine is defined (see **DSSetCallback**), it is called when the driver switches from the first buffer to the second buffer.

Playback is limited to a maximum of 1152800 bytes (1048 seconds, or approx. 17.5 minutes), as specified by the *len* parameter. Note that this parameter specifies the total length to be played, not the individual buffer length.

The data format is determined by the data file header. The entire header must be passed to the driver as part of the first buffer of sampled data. See appendix A for data file headers.

Returns: E_OK, E_LENGTH, E_NOBUFFER, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -        AX_BPDIGI
CX -        Data length low word
```

DX - Data length high word

CVSD Format Functions

Digispeech CVSD functions utilize compression rates from 1800 bytes per second (bps) to 4800 BPS. Speech compressed in CVSD format at 4800 bps has the same quality as speech compressed in Digispeech format. CVSD compression at rates above 3000 bps of audio containing music or a combination of music and speech sometimes results in better sound quality than Digispeech format compression.

Routines in this section are compatible with the equivalent IBM Speech Adapter BIOS calls. Programs written using these functions should operate unchanged using the IBM PS/2 Speech Adapter, providing that the DSP Solutions **DriverInstalled** function is not used to detect a driver. **DriverInstalled** does not detect the presence of an IBM PS/2 Speech Adapter.

DSFRecCVSD

#include "digi.h"

int DSFRecCVSD(char _far *buf,int rate, WORD len);

buf -	FAR pointer to sampled data buffer.
rate -	Recording rate code.
len -	Maximum number of bytes to record.

Description:

Foreground recording in CVSD format. Sampled data is stored in *buf*, and is limited to a maximum of 65535 bytes, as specified by the *len* parameter. The duration of the recording is determined by the *len* parameter divided by the *rate* parameter value, as specified below.

RATE_1800	1800 bytes per second
RATE_2400	2400 bytes per second
RATE_3000	3000 bytes per second
RATE_3600	3600 bytes per second
RATE_4200	4200 bytes per second
RATE_4800	4800 bytes per second

DSFRecCVSD returns control to the calling program only when recording is complete. Keyboard interrupts are disabled during foreground playback operations to prevent sound distortion.

Returns: E_OK, E_CVSDSPEED, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -        AX_FRCVSD
BL -        rate code
CX -        Buffer length
SI -        Buffer offset
DS -        Buffer segment
```

DSFPlayCVSD

#include "digi.h"

int DSFPlayCVSD(char _far *buf, WORD rate, WORD len);

buf -	FAR pointer to sampled data buffer.
rate -	Playback rate code.
len -	Number of bytes in buffer to be played.

Description:

Foreground playback of CVSD format data. Sampled data is read from *buf*, and is limited to a maximum of 65535 bytes, as specified by the *len* parameter. The duration of the playback is determined by the *len* parameter divided by the *rate* parameter value.

See the **DSFRecCVSD** routine description for *rate* parameter values.

DSFPlayCVSD returns control to the calling program only when playback is completed. Keyboard interrupts are disabled during foreground playback operations to prevent sound distortion.

Returns: E_OK, E_CVSDSPEED, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -        AX_FPCVSD
BL -        rate code
CX -        Buffer length
SI -        Buffer offset
DS -        Buffer segment
```

DSFRecUCVSD

#include "digi.h"

int DSFRecUCVSD(char _far *buf,int speed,WORD len);

buf -	FAR pointer to sampled data buffer.
speed -	Recording rate divisor.
len -	Maximum number of bytes to record.

Description:

Same as **DSFRecCVSD** except that the *speed* parameter specifies the sampling rate as follows:

sampling rate = 596656 / *speed*.

This implementation quantizes the user specified speed into six levels according to the following rules.

speed parameter	sampling rate
speed < 132	1800 bytes per second
132 <= speed < 152	2400 bytes per second
152 <= speed < 180	3000 bytes per second
180 <= speed < 220	3600 bytes per second
220 <= speed < 284	4200 bytes per second
284 <= speed	4800 bytes per second

Returns: E_OK, E_LENGTH, E_PAUSE, E_BUF0, E_BUF1,
E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -        AX_FRUCVSD
BX -        speed parameter
CX -        Buffer length
SI -        Buffer offset
DS -        Buffer segment
```

DSFPlayUCVSD

#include "digi.h"

```
int DSFPlayUCVSD(char _far *buf,int speed,WORD len);
```

buf -	FAR pointer to sampled data buffer.
speed -	Playback rate divisor.
len -	Number of bytes in buffer to be played.

Description:

Same as **DSFPlayCVSD** except that the *speed* parameter specifies the sampling rate as follows:

sampling rate = 596656 / *speed*.

The current implementation quantizes the user specified speed into six levels according to the following rules.

speed parameter	sampling rate
speed < 132	1800 bytes per second
132 <= speed < 152	2400 bytes per second
152 <= speed < 180	3000 bytes per second
180 <= speed < 220	3600 bytes per second
220 <= speed < 284	4200 bytes per second
284 <= speed	4800 bytes per second

Returns: E_OK, E_LENGTH, E_PAUSE, E_BUF0, E_BUF1,
E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -        AX_FPUCVSD
BX -        speed parameter
CX -        Buffer length
SI -        Buffer offset
DS -        Buffer segment
```

DSBPlayCVSD

#include "digi.h"

int DSBPlayCVSD(int rate,DWORD len);

rate -	Playback rate code (see below).
len -	Number of bytes in buffer to be played.

Description:

Starts background playback of CVSD format data. Background playback is accomplished via a double-buffering scheme. Sampled data is read from one buffer while the second buffer is loaded by the application program.

Background buffers are set up using the **DSSetBuffer** function. The **DSGetStatus** routine is used to determine the buffer from which the **DSBPlayCVSD** routine is currently drawing data.

The total duration of the playback is determined by the *len* parameter divided by the *rate* parameter value. The playback operation is terminated by the driver after *len* bytes are sent to the Audio Adapter unit. Background operation may be aborted using the **DSReset** function. If a callback routine is defined (see **DSSetCallback**), it is called when the driver switches from one buffer to the other buffer.

See the **DSFRecCVSD** routine description for *rate* parameter values.

Returns: E_OK, E_CVSDSPEED, E_LENGTH, E_NOBUFFER, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -        AX_BPCVSD
BL -        rate code
CX -        Data length low word
```

DX - Data length high word

DSBRecCVSD

#include "digi.h"

int DSBRecCVSD(int rate,DWORD len);

rate -	Recording rate code.
len -	Number of bytes in buffer to be played.

Description:

Starts background recording in CVSD format. Sampled data is stored in *buf*, and is limited to a maximum of xxxxx bytes, as specified by the *len* parameter. Background recording is accomplished via a double-buffering scheme. Sampled data is written to one buffer while the second buffer is being saved by the application program.

Background buffers are set up using the **DSSetBuffer** function. The **DSGetStatus** routine is used to determine the buffer to which the **DSBRecCVSD** routine is currently writing data.

The duration of the recording is determined by the *len* parameter divided by the *rate* parameter value. Background operation may be aborted using the **DSReset** function. If a callback routine is defined (see **DSSetCallback**), it is called when the driver switches from the first buffer to the second buffer.

See the **DSFRecCVSD** routine description for *rate* parameter values.

Returns: E_OK, E_CVSDSPEED, E_LENGTH, E_NOBUFFER, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -        AX_BRCVSD
BL -        rate code
CX -        Data length low word
```

DX - Data length high word

PCM Format Functions

DSFPlayPCM

#include "digi.h"

int DSFPlayPCM(char _far *buf,WORD len);

buf -	FAR pointer to sampled data buffer.
len -	Number of bytes in buffer to be played.

Description:

DSFPlayPCM is a synonym for **DSFPlayDigi**. See that function description for further information.

Returns: E_OK, E_LENGTH, E_PAUSE, E_BUF0, E_BUF1,
E_COM, E_TIMEOUT, E_MUSIC

Implementation: See **DSFPlayDigi**

DSBPlayOKI

#include "digi.h"
[DS201A][DS301]

int DSBPlayOKI(int pole, WORD rate, DWORD len);

pole -	Set to 0
rate -	Playback sample rate.
len -	Number of bytes in buffer to be played.

Description:

Starts background playback of OKI ADPCM format data.

Except for *rate* and *pole* values, **DSBPlayOKI** is identical in operation to **DSBPlayPCM**.

The *pole* parameter is unused by most of the DIGI drivers and should be set to 0 for nominal operation.

The *rate* parameter specifies the playback sampling rate, 8000 to 11025 samples per second (DS201A), or 4000 to 44100 samples per second (DS301).

Returns: E_OK, E_UNDEFINED, E_ARGUMENT, E_LENGTH, E_NOBUFFER, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT, E_MUSIC

Implementation:

```
AX -        AX_BPOKIADPCM
BX -        pole
CX -        Data length low word
DX -        Data length high word
SI -        Playback sampling rate
```

LPC Playback Functions

The DS201A and DS301 audio adapters support LPC speech synthesis compatible with the IBM PS/2 Speech Adapter LPC implementation. The DIGI drivers implement the following LPC related functions of PS/2 Speech Adapter API BIOS (interrupt 4DH):

AX=0200H	- LPC Status
AX=0201H	- *LPC Speak vocabulary word., background.
AX=0202H	- LPC Speak data in buffer, background.
AX=0300H	- LPC Status
AX=0301H	- *LPC Speak vocabulary word., foreground.
AX=0302H	- LPC Speak data in buffer, foreground.

* - Requires /V command line parameter when driver is loaded.

More information about LPC functions can be found in the *PS/2 Speech Adapter Technical Reference* by IBM, 68X2207 S68X-2207-00, January 1987.

DSLPCStatus

#include "digi.h"
[DS201A][DS301]

int DSLPCStatus(void);

Description:

Returns the status of the current LPC operation.

Returns: E_OK, E_LPC, E_LPCINDEX, E_COM

Implementation:

```
AX -            AX_LPCSTAT
```

DSFPlayLPC

#include "digi.h"
[DS201A][DS301]

int DSFPlayLPC(int index);

index - LPC vocabulary word index.

Description:

Foreground playback of an LPC vocabulary word. The word played is determined by the *index* parameter and the loaded vocabulary.

This routine requires the DSP Solutions driver to be loaded with either the /V (use standard IBM LPC vocabulary) parameter, or with a custom vocabulary file.

The routine returns control to the calling program only when playback is completed.

Returns: E_OK, E_LPCINDEX, E_COM, E_MUSIC

Implementation:

```
AX -        AX_FLPCIX
BX -        Vocabulary word index
```

DSFPlayULPC

#include "digi.h"
[DS201A][DS301]

int DSFPlayULPC(char _far *buf , int len);

buf -	FAR pointer to LPC data buffer.
len -	buffer length (max 4095).

Description:

Foreground playback of an LPC data buffer. The word or phrase played is determined by the data contained in the buffer.

The routine returns control to the calling program only when playback is completed.

Returns: E_OK, E_LPCINDEX, E_COM, E_MUSIC

Implementation:

```
AX -        AX_FLPCBUF
CX -        Buffer data length
SI -        Buffer offset
DS -        Buffer segment
```

DSBPlayLPC

#include "digi.h"
[DS201A][DS301]

int DSBPlayLPC(int index);

index - LPC vocabulary word index.

Description:

Starts background playback of an LPC vocabulary word. The word played is determined by the *index* parameter and the loaded vocabulary.

This routine requires the DSP Solutions driver to be loaded with either the /V (use standard IBM LPC vocabulary) parameter, or with a custom vocabulary file.

The routine returns control to the calling program immediately. Use the **DSLPCStatus** routine to determine when playback is completed.

Returns: E_OK, E_LPCINDEX, E_COM, E_MUSIC

Implementation:

```
AX -        AX_BLPCIX
BX -        Vocabulary word index
```

DSBPlayULPC

#include "digi.h"
[DS201A][DS301]

int DSBPlayULPC(char _far *buf , int len);

buf -	FAR pointer to LPC data buffer.
len -	buffer length (max 4095).

Description:

Starts background playback of an LPC data buffer. The word or phrase played is determined by the data contained in the buffer.

The routine returns control to the calling program immediately. Use the **DSLPCStatus** routine to determine when playback is completed.

Returns: E_OK, E_LPCINDEX, E_COM, E_MUSIC

Implementation:

```
AX -        AX_BLPCBUF
CX -        Buffer data length
SI -        Buffer offset
DS -        Buffer segment
```

Tone Generator Functions

Note: The DS301 does not contain a 3-tone generator. The PDIGI driver emulates the DS201A 3-tone generator functions using the DS301 music synthesizer.

DSToneStart

```
#include "digi.h"

int DSToneStart( void );
```

Description:

Enable the DIGI driver 3-tone generator functions. Places the DIGI driver in tone generator mode, precluding other operations until **DSToneStop** is called to exit tone generator mode.

Returns: E_OK, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT

Implementation:

```
AX -          AX_STONE
```

DSToneCmd

#include "digi.h"

int DSToneCmd(BYTE command);

command - BYTE containing command parameter(s).

Description:

Send a command to the 3-tone generator. The **DSToneCmd** routine can be called only after **DSToneStart** has been successfully called. The command format is the same as that of the SN76496N Complex Sound Generator I.C. The DSP Solutions Audio Adapter does not support a noise generator, as the SN76496N does, but it supports 3 tone generators, which can be programmed independently.

There are 3 different sub-commands:

- Set Attenuation.
- Set 4 Least Significant Bits of Frequency Counter.
- Set 6 Most Significant Bits of Frequency Counter.

The tone frequency is physically changed only when "Set Frequency MSBs" command is sent to the driver. The "Set Frequency LSBs" command is latched by the driver but does not affect the tone frequency until the upper part of the frequency counter is updated.

The following pages describe the **DSToneCmd** sub-command codes.

Returns: E_OK, E_IGNORED, E_COM, E_TIMEOUT

Implementation:

```
AH -        AH_CTONE
AL -        tone command parameter
```

DSToneCmd - Set Attenuation

Command parameter format:

```
-------------------------------------------------
|Bit7 |Bit6 |Bit5 |Bit4 |Bit3 |Bit2 |Bit1 |Bit0 |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | t1  | t0  | 1   | g3  | g2  | g1  | g0  |
-------------------------------------------------
```

Where: `<t1,t0>` field is tone selector:

00 : Tone #1 is selected
01 : Tone #2 is selected
10 : Tone #3 is selected

`<g3,g2,g1,g0>` field is attenuation factor of specified tone:

0000 - No attenuation
0001 : (-02dB) attenuation
0010 : (-04dB) attenuation
0011 : (-06dB) attenuation
0100 : (-08dB) attenuation
0101 : (-10dB) attenuation
0110 : (-12dB) attenuation
0111 : (-14dB) attenuation
1000 : (-16dB) attenuation
1001 : (-18dB) attenuation
1010 : (-20dB) attenuation
1011 : (-22dB) attenuation
1100 : (-24dB) attenuation
1101 : (-26dB) attenuation
1110 : (-28dB) attenuation
1111 - silence

DSToneCmd - Set Frequency LSBs

Command parameter format:

```
-----------------------------------------------
|Bit7 |Bit6 |Bit5 |Bit4 |Bit3 |Bit2 |Bit1 |Bit0 |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | t1  | t0  | 0   | n3  | n2  | n1  | n0  |
-----------------------------------------------
```

Where: `<t1,t0>` field is tone selector:

00 : Tone #1 is selected
01 : Tone #2 is selected
10 : Tone #3 is selected

`<n3,n2,n1,n0>` field is 4 least significant bits of 10-bit number, N, used to set the tone frequency.

The frequency of selected tone can be calculated as follows:

$$F = 3{,}579{,}000 \text{ Hz} / 32 / N$$

The tone frequency is physically changed only when "Set Frequency MSBs" command is sent to the driver. The "Set Frequency LSBs" command is latched by the driver but does not affect the tone frequency until the upper part of the frequency counter is updated.

DSToneCmd - Set Frequency MSBs

Command parameter format:

```
------------------------------------------------
|Bit7 |Bit6 |Bit5 |Bit4 |Bit3 |Bit2 |Bit1 |Bit0 |
|-----|-----|-----|-----|-----|-----|-----|-----|
|  0  |  x  | n9  | n8  | n7  | n6  | n5  | n4  |
------------------------------------------------
```

Tone number is set to the tone number of the previous command.

`<n9,n8,...,n4>` field is 6 most significant bits of 10-bit number, N, used to set the tone frequency.

The frequency of selected tone can be calculated as follows:

$$F = 3{,}579{,}000 \text{ Hz} / 32 / N$$

The tone frequency is physically changed only when "Set Frequency MSBs" command is sent to the driver. The "Set Frequency LSBs" command is latched by the driver but does not affect the tone frequency until the upper part of the frequency counter is updated.

DSToneStop

#include "digi.h"

int DSToneStop(void);

Description:

Disable the DIGI driver 3-tone generator functions, allowing other operations. Use **DSToneStart** to enter tone generator mode.

Returns: E_OK

Implementation:

```
AX -          AX_ETONE
```

Synthesizer Emulator Functions

The Synthesizer Emulator Functions of the PDIGI driver are used to control the DS301 music synthesizer via function calls that emulate I/O port writes to the Yamaha YM3812 synthesizer chip. Applications that are written for this chip can be easily adapted to the DS301 by adding some initialization code and changing the appropriate input and output instructions to calls into these library functions.

The Synthesizer Emulator functions are not available on the DS201 or DS201A.

The Synthesizer Emulator functions are contained in a library module separate from the other low-level routines. If none of the functions are used, the module is not linked into the application, saving space.

During operation of the Synthesizer Emulator, recording is not permitted, and playback of digitized audio is limited to eight bit linear (DF_PCM8) or mu-Law (DF_PCMMU) PCM.

DSGetEntry

#include "digi.h"
[DS301]

```
int (_far *DSGetEntry( ))( int func, BYTE addr, BYTE value);
```

Description:

Places a far pointer to the synthesizer control function **DSSynthEntry** in *funcPtr*. **DSGetEntry** requires the PDRVXM driver loaded with the DOS command line `PDIGI /XM`. The upper sixteen bits of the *caps* value returned by the **DSQuery** function contains the constant DID_301XM if this driver is loaded. NULL is placed in *funcPtr* if the appropriate driver is not present.

Returns: E_OK

Implementation:

```
AX -          AX_DSMGETENTRY
```

DSEmulate

#include "digi.h"
[DS301]

int DSEmulate(int function, BYTE addr, BYTE value);

function -	Synthesizer function
addr -	YM3812 register address.
value -	YM3812 register value.

Description:

This function is not contained in the DIGIAPI library. Rather, it is called via a function pointer obtained in a call to the **DSGetEntry** function.

DSEmulate provides access to the synthesizer emulator functions without the overhead of the INT4D interface used by most of the other library functions.

The emulator subfunctions are used by passing the subfunction code in the function parameter and the YM3812 parameters as required. The following pages describe the **DSEmulate** sub-functions.

Returns: E_OK

Implementation:

```
push    word ptr ymValue
push    word ptr ymAddr
push    word ptr function
call    far ptr funcPtr         ;Obtained from
                                ;DSGetEntry.
```

DSEmulate - SYNTH_INIT

```
#include "digi.h"
        [DS301]

int  DSEmulate( SYNTH_INIT, 0, 0 );
```

Description:

Initialize the Synthesizer Emulator. Places the DS301 in synthesizer mode and sets the emulator registers to their default state. Ignored if already in synthesizer mode.

Returns: E_OK, E_IGNORED

DSEmulate - SYNTH_READ

```
#include "digi.h"
        [DS301]

int DSEmulate( SYNTH_READ, addr, 0 );
```

addr - YM3812 register address.

Description:

Returns the last value written to the emulator at the passed YM3812 register address. If no value has been written, the default value is returned. No error values are returned.

Returns: See description.

DSEmulate - SYNTH_WRITE

#include "digi.h"
[DS301]

int DSEmulate(SYNTH_WRITE, addr, value);

addr -	YM3812 register address.
value -	YM3812 register value.

Description:

Write an eight bit value to an emulated YM3812 register. The emulator generates the appropriate data to be sent to the DS301.

Returns: E_OK

DSEmulate - SYNTH_RESET

```
#include "digi.h"
        [DS301]

int DSEmulate( SYNTH_RESET, 0, 0 );
```

Description:

Reset the synthesizer emulator. Stops synthesizer operation, but does not stop digitized audio if in progress.

The standard **DSReset** function will not stop emulator operation. To completely reset the driver, this function must be called in addtion to **DSReset**.

Returns: E_OK

Synthesized Music Functions

The Synthesized Music Functions of the PDIGI driver are used to control the DS301 music synthesizer. Applications that provide note to note timing may use these functions to implement MIDI playback. Synthesized Music is not available on the DS201 or DS201A.

The Synthesized Music functions are contained in a library module separate from the other low-level routines. If none of the functions are used, the module is not linked into the application, saving space.

The driver provides two levels of support for synthesized music. The following functions provide no note to note timing control. They are for use by applications that provide note timing independently.

The High Level functions provide support for Creative Labs .CMF file data. Using these functions, combined with some of the following functions, an application can directly play Creative Labs .CMF song files.

DSInitSynthesizer

#include "digi.h"
[DS301]

void DSInitSynthesizer(void);

Description:

Initializes the synthesizer decode portion of the DIGI driver. Must be called before any other Synthesizer Music function. This function is called by the **DSMusicOpen** high level function.

The DS301 music driver must be loaded (PDIGI /M) for this function to operate. All other drivers will return E_INVALID.

Returns: E_OK, E_INVALID

Implementation:

```
AX -          AX_SYNTHINIT
```

DSSetPatchTable

#include "digi.h"
[DS301]

int DSSetPatchTable(char _far *table, int count);

table -	FAR pointer to user instrument table.
count -	Number of instruments in user table.

Description:

This function is used to specify the set of instrument definitions for use by the Synthesizer Music note functions and the High Level music functions.

If this function is not called, the music functions reference a set of 16 default instruments contained within the driver.

The application passes a FAR pointer to it's instrument table. The table format is the same as used by Creative Labs .CMF file format instrument block. It is an array of 16-byte structures with the following format.

```
typedef unsigned char BYTE;
typedef struct _instrumentDef
  {
  BYTE mSound;    // Modulator Sound Characteristics
                  // Bit 7     : Pitch Vibrato
                  // Bit 6     : Amplitude Vibrato
                  // Bit 5     : Sustaining snd type
                  // Bit 4     : Envelope Scaling
                  // Bit 3-0   : Frequency Multiplier
  BYTE cSound;   // Carrier Sound Characteristics
                  // Same bits as mSound.
  BYTE mScaling;// Modulator Scaling / Output
                  // Bit 7-6   : Level Scaling
                  // Bit 5-0   : Output level.
  BYTE cScaling;// Carrier Scaling / Output
                 // Same bits as mScaling.
  BYTE mAttDec; // Modulator Attack/Decay rates
```

```
                    // Bit 7-4    : Attack Rate.
                    // Bit 3-0    : Decay Rate.
BYTE cAttDec; // Carrier Attack/Decay rates
                   // Same bits as mAttDec
BYTE mSusRel; // Modulator Sustain/Release rates
                    // Bit 7-4    : Sustain Level.
                    // Bit 3-0    : Release Rate.
BYTE cSusRel; // Carrier Sustain/Release rates
                   // Same bits as mSusRel
BYTE   mWaveSel;// Modulator Wave Select
                    // Bit 7-2    : All 0
                    // Bit 1-0    : Wave Select.
BYTE   cWaveSel;// Carrier Wave
                   // Same bits as mWaveSel
BYTE   feedback;// Feedback / connection
                    // Bit 7-4    : All 0
                    // Bit 3-1    : Modulator Feedback.
                    // Bit 0      : Connection
BYTE   reserved [5];
} INSTRUMENT;
```

Returns: E_OK

Implementation:

```
AX -        AX_SETINSTTAB
CX -        Instrument count
SI -        Instrument array offset
DS -        Instrument array segment
```

DSSetMusicMode

#include "digi.h"
[DS301]

int DSSetMusicMode(int mode);

mode - 0 - Melody mode (9 melody voices).
1 - Rhythm mode (7 melody voices, 4 rhythm).

Description:

Select the music synthesizer operating mode. Melody mode (9 voices) is the default and is set whenever the **DSReset** function is called.

Returns: E_OK

Implementation:

```
AX -          AX_MUSICMODE
CX -          Music mode
```

DSTranspose

#include "digi.h"
[DS301]

int DSTranspose(int semiToneOffset);

semiToneOffset - pitch offset value.

Description:

This function causes the Synthesizer Music functions to play all notes at a higher or lower pitch than specified by the actual note parameters. The *semitoneOffset* value may be positive or negative and is added to each note's pitch value before the note is played. If the offset would cause the pitch value to exceed the driver limit, it is truncated to the appropriate limit.

Returns: E_OK

Implementation:

```
AX -        AX_TRANSPOSE
CX -        pitch offset
```

DSMIDINoteOn

#include "digi.h"
[DS301]

int DSMIDINoteOn(char channel, char note, char velocity);

channel - MIDI channel to play note (range 0 - 15).
note - MIDI note number (range 0 - 127).
velocity - MIDI note velocity (range 0 - 127).

Description:

Plays a note on the requested channel. Notes are specified using the same parameters as MIDI format music data.

The routine returns control to the calling program immediately. The calling program is responsible for timing between notes. Use the **DSMIDINoteOff** function to turn off the note.

Calling the **DSMIDINoteOn** function with a *velocity* of 0 is the same as calling the **DSMIDINoteOff** function with the same *channel* and *note* parameters.

Returns: E_OK

Implementation:

```
AX -        AX_NOTEON
BX -        Channel number
CX -        Note
DX -        Velocity
```

DSMIDINoteOff

#include "digi.h"
[DS301]

int DSMIDINoteOff(char channel, char note);

channel - MIDI channel to play note (range 0 - 15).
note - MIDI note number (range 0 - 127).

Description:

Turns the specified note off on the requested channel. Notes are specified using the same parameters as MIDI format music data.

The calling program is responsible for timing between notes. When turning notes on and off at the same time interval, perform the **DSMIDINoteOff** functions first. This allows the **DSMIDINoteOn** function to allocate voices for the notes to be turned on.

Since multiple notes can be played on the same channel, you must specify the note value in addition to the channel.

Calling the **DSMIDINoteOn** function with a *velocity* of 0 is the same as calling the **DSMIDINoteOff** function with the same *channel* and *note* parameters.

Returns: E_OK

Implementation:

```
AX -        AX_NOTEOFF
BX -        Channel number
CX -        Note
```

DSMIDIPgmChange

#include "digi.h"
[DS301]

int DSMIDIPgmChange(char channel, char program);

channel -MIDI channel to change program (range 0 - 15).
program -MIDI program number (range 0 - 127).

Description:

Changes the program patch (instrument) for the specified *channel*. The *program* parameter specifies the index of a instrument in the driver patch table.

The driver provides a default set of 16 patches, or the **DSSetPatchTable** function can be called to load a custom patch table. The program parameter should not exceed the maximum number of patches currently available.

The application should turn all notes off for the specified channel before calling the **DSMIDIPgmChange** function to prevent unwanted sound effects from changing the instrument on an active note.

Returns: E_OK

Implementation:

```
AX -        AX_PROGRAMCHG
BX -        Channel number
CX -        Program number
```

High Level Functions

The High level functions have been added to the PDIGI library to allow the programmer to talk to the Audio Adapter driver using a minimum of functions. For example, the **Tone** function is the only function needed to access the tone generator, and the parameters are simplified from the low level requirements.

With the exception of the **DriverInstalled** function, the high level functions are contained in a library module separate from the low-level routines. If none of the high level functions are used, the module is not linked into the application, saving space. The **DriverInstalled** function is contained in the Utility Functions module.

General

These high level functions are contained in the same library module as the low-level functions, since they are commonly used even if none of the other high-level functions are used.

DriverInstalled

```
#include "digi.h"

int  DriverInstalled(void);
```

Description:

Determines if a DSP Solutions Audio Adapter driver is present in memory.

Returns:

TRUE (1) -	DSP Solutions driver present.
FALSE (0) -	DSP Solutions driver not found.

Implementation: Implemented by the DIGIAPI library.

Digitized Audio

These high level functions control the digitized audio functions of the driver. They are contained in their own DIGIHI library module within the DIGIAPI library.

StartPlay

#include "digi.h"

int StartPlay(int format, WORD rate, int channels, DWORD len);

format -	audio data format.
rate -	Sample rate appropriate to format.
channels -	1- mono, 2 - stereo
len -	Number of bytes to be played.

Description:

One-stop shopping for the lower level background play-back routines. Chooses the appropriate low-level routine for all drivers and audio devices.

The *channels* parameter is ignored for formats that do not support stereo. See **DSBStereo** for applicable formats.

See the **DSBPlay**, **DSBStereo**, **DSBPlayCVSD**, **DSBPlayOKI**, and **DSBPlayDigi** descriptions for parameter values and description of operation for each data format.

Returns: E_OK, E_UNDEFINED, E_ARGUMENT, E_LENGTH, E_NOBUFFER, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT, E_MUSIC

Implementation: Implemented by the DIGIAPI library.

StartRecord

#include "digi.h"

int StartRecord(int format, WORD rate,int channels, DWORD len);

format -	audio data format.
rate -	Sample rate appropriate to format.
channels -	1- mono, 2 - stereo
len -	Number of bytes to be played.

Description:

One-stop shopping for the lower level background play-back routines. Chooses the appropriate low-level routine for all drivers and audio devices.

The *channels* parameter is ignored in the current version.

See the **DSBRec**, and **DSBRecCVSD** descriptions for parameter values and description of operation for each data format.

Returns: E_OK, E_UNDEFINED, E_ARGUMENT, E_LENGTH, E_NOBUFFER, E_PAUSE, E_BUF0, E_BUF1, E_COM, E_TIMEOUT, E_MUSIC

Implementation: Implemented by the DIGIAPI library.

Tone Generator

These high level functions control the three-voice tone generator. They are contained in their own TONE library module within the DIGIAPI library.

Tone

#include "digi.h"

```
int Tone(int command,int tone,WORD freq);
```

command	- Tone generator function code.
tone	- tone index, range 0 - 2.
freq	- frequency, range 160 - 3000 (TONE_FREQ) attenuation level, range 0 - 15 (TONE_LEVEL)

Description:

One-stop shopping for the lower level Tone Generator routines. Not all parameters are used by all functions. The following table lists supported functions and their required parameters.

command **parameter**	**required parameters**
TONE_ENABLE	none
TONE_DISABLE	none
TONE_LEVEL	*tone,freq(level)*
TONE_FREQ	*tone,freq*

Parameters that are not required for a particular function are ignored, but should be specified as 0.

Returns: E_OK

Implementation: Implemented by the DIGIAPI library.

PlayOneTone

```
#include "digi.h"

int PlayOneTone( int  tone, int  freq, int level, WORD duration );
```

tone - tone index, range 0 - 2.
freq - frequency, range 160 - 3000
level - attenuation level, range 0 - 15
duration - length of tone, in milliseconds.

Description:

Play a tone using a single channel of the 3-tone generator. The tone generator is enabled and disabled automatically.

The routine returns only after *duration* milliseconds.

Returns: E_OK

Implementation: Implemented by the DIGIAPI library.

PlayThreeTone

```
#include "digi.h"

int PlayThreeTone( int f1, int f2, int f3, int level, WORD
duration );
```

f1	- frequency of tone channel 1, range 160 - 3000
f2	- frequency of tone channel 2, range 160 - 3000
f3	- frequency of tone channel 3, range 160 - 3000
level	- attenuation level, range 0 - 15
duration	- length of tone, in milliseconds.

Description:

Play a tone using all three channels of the 3-tone generator. The tone generator is enabled and disabled automatically.

The routine returns only after *duration* milliseconds.

Returns: E_OK

Implementation: Implemented by the DIGIAPI library.

Synthesized Music

These high level functions control the music synthesizer. They are contained in their own HFM library module within the PDIGI library.

DSMusicInit

#include "digi.h"

int DSMusicInit(void);

Description:

Checks for a driver that supports FM Music and initializes the driver music functions. This function must be called prior to using the **DSReadStatus** function. If this function is used, the **DSSetStatusAddress** function must *not* be called.

If this function is successful, the calling program *must* call the **DSTerminateMusic** function prior to exiting.

Returns: E_OK

Implementation: Implemented by the DIGIAPI library.

DSSetStatusAddress

#include "digi.h"
[DS301]

```
void DSSetStatusAddress( BYTE _far *statusByte );
```

statusByte - FAR pointer to an unsigned char.

Description:

The calling program passes a pointer to it's local music status byte. The PDIGI driver saves the pointer and sets the music status byte to 0x00. During music play operations, the calling program may poll this byte to obtain the current status of the music. The status byte is set to 0xFF by the **DSPlayMusic** routine at the start of play. The status byte is set to 0x00 when play is halted via **DSStopMusic**, **DSReset** or the MIDI STOP status function code.

The status byte can also be set by the MIDI Control Change function 0x67. This is a non-standard MIDI extension that can be used to synchronize an external program action with FM music events.

Returns: E_OK

Implementation:

```
AX -        AX_SETSTATADDR
SI -        statusByte offset
DS -        statusByte segment
```

DSMusicStatus

```
#include "digi.h"
        [DS301]

int  DSMusicStatus( void );
```

Description:

Return the current contents of the music status byte set by the **DSSetStatusAddress** routine. If the value is non-zero, it is set to 0xFF to avoid multiple reads of the same status value.

The status byte is set to 0xFF by the **DSPlayMusic** routine at the start of play. The status byte is set to 0x00 when play is halted via **DSStopMusic**, **DSReset** or the MIDI STOP status function code.

The status byte can also be set by the MIDI Control Change function 0x67. This is a non-standard MIDI extension that can be used to synchronize an external program action with music events.

Returns: E_OK

Implementation: Implemented by the DIGIAPI library.

DSSetMusicRate

```
#include "digi.h"
        [DS301]

int DSSetMusicRate( WORD ticksPerSecond );
```

ticksPerSecond -Music rate (range 20 - 160), default 96.

Description:

Set the rate, in ticks per second, at which the music interpreter will process MIDI data.

Returns: E_OK

Implementation: Implemented by the DIGIAPI library.

DSStartMusic

#include "digi.h"
[DS301]

int DSPlayMusic(char _far *buf);

buf - FAR pointer to buffer containing MIDI format music data.

Description:

Background play of MIDI format synthesized music.

The routine returns control to the calling program immediately. The calling program can monitor the music status byte or use **DSMusicStatus** to determine completion of play.

Returns: E_OK

Implementation:

```
AX -        AX_PLAYMUSIC
SI -        Music data buffer offset
DS -        Music data buffer segment
```

DSStopMusic

#include "digi.h"
[DS301]

int DSStopMusic(void);

Description:

Stop current synthesized music play operation.

Returns: E_OK

Implementation:

```
AX -            0x0806
```

DSTerminateMusic

#include "digi.h"

int DSSetMusicCallback(char _far *routine);

Description:

Undo **DSMusicInit.**

Returns: E_OK

Implementation: Implemented by the DIGIAPI library.

Appendix A - Data File Formats

This appendix defines the DSP Solutions Audio data file formats. For all new designs, it is recommended that the Microsoft .WAV format be used. All new DSP Solutions utilities utilize the .WAV format.

All values are displayed as hexadecimal. File extensions shown in parenthesis are suggested, but not enforced.

Digispeech Standard Format file (.PAC)

```
--------------------------------------------------
| 77 | N1 | N2 |  DIGISPEECH STANDARD FORMAT DATA
--------------------------------------------------
```

$N_1:N_2$ - number of frames (N_1 is the MSB)

Digispeech Fixed (Movie) Format file (.FIX)

```
--------------------------------------------------
| 78 | N1 | N2 |  DIGISPEECH FIXED FORMAT DATA
--------------------------------------------------
```

$N_1:N_2$ - number of frames (N_1 is the MSB)

Digispeech Realtime Format file (.RPE)

```
--------------------------------------------------
| 79 |  DIGISPEECH REALTIME FORMAT DATA
--------------------------------------------------
```

PCM 8 bit linear file (.PCM)

```
--------------------------------------------------
| 80 |  PCM LINEAR DATA
--------------------------------------------------
```

PCM 16 bit linear file (.SAM)

```
--------------------------------------------------
| FE |  16 BIT PCM LINEAR DATA
--------------------------------------------------
```

Note: Early .SAM files have no header byte.

PCM A-Law file (.A)

```
-----------------------------------------------------------
| 55 |  PCM A-LAW DATA
-----------------------------------------------------------
```

PCM Mu-Law file (.U)

```
-----------------------------------------------------------
| FF |  PCM MU-LAW DATA
-----------------------------------------------------------
```

ADPCM file (OKIDATA ADPCM) (.OKI)

```
-----------------------------------------------------------
| 11 | N1 | OKIDATA ADPCM DATA
-----------------------------------------------------------
```

N_1 - sampling rate index (0-3).
0 - 8 KHz (4.0 Kbyte/sec)
1 - 9 KHz (4.5 Kbyte/sec)
2 - 10 KHz (5.0 Kbyte/sec)
3 - 11 KHz (5.5 Kbyte/sec)

DVI ADPCM (.DVI)

```
-----------------------------------------------------------
| 12 | N1 | N2 | DVI ADPCM DATA
-----------------------------------------------------------
```

$N_2:N_1$ - samples per second (N_1 is the LSB)

Digispeech CVSD file (.CVx) (x = 0-5)

```
-----------------------------------------------------
| 44 | N3 | N4 | N5 |  CVSD DATA
-----------------------------------------------------
```

N_3 - CVSD speed index (between 00H and 05H)
0 - 1800 bytes/sec.
1 - 2400 bytes/sec.
2 - 3000 bytes/sec.
3 - 3600 bytes/sec.
4 - 4200 bytes/sec.
5 - 4800 bytes/sec.

N_4:N_5 - number of words (N_4 is the MSB)

Note: CVSD DATA for both Digispeech CVSD and IBM Linkway CVSD is the same after the file header.

IBM Linkway CVSD file (.CVS)

```
-----------------------------------------------------
|N1 |N2 |N3 |N4 |N5 |R1 |X1 |X2 |X3 |X4 |  CVSD DATA
-----------------------------------------------------
```

N_1:N_5 - CVSD DATA Length in ASCII format.

R_1 - CVSD speed index (between 0 and 5) in ASCII format.
0 - 1800 bytes/sec.
1 - 2400 bytes/sec.
2 - 3000 bytes/sec.
3 - 3600 bytes/sec.
4 - 4200 bytes/sec.
5 - 4800 bytes/sec.

X_4:X_4 - ASCII space characters (0x20)

Note: CVSD DATA for both Digispeech CVSD and IBM Linkway CVSD is the same after the file header.

Microsoft Windows WAVE (.WAV)

The format of Windows .WAV files is documented in the Microsoft Multimedia development kit. Please consult the Microsoft documentation for further information.

The following WAVE file format identifiers are supported by the DSP Solutions DOS drivers.

```
#define WAVE_FORMAT_PCM              0x0001
#define WAVE_FORMAT_IBM_CVSD         0x0005
#define WAVE_FORMAT_ALAW             0x0006
#define WAVE_FORMAT_MULAW            0x0007
#define WAVE_FORMAT_OKI_ADPCM        0x0010
#define WAVE_FORMAT_DVI_ADPCM        0x0011
#define WAVE_FORMAT_DIGISTD          0x0015
#define WAVE_FORMAT_DIGIFIX          0x0016
#define WAVE_FORMAT_UNKNOWN          0x0000
```

Appendix B - DIGI.H & DIGIDRV.H

This appendix provides the applicable parts of the DIGI.H and DIGIDRV.H 'C" language header files.

```
    /* DSQuery flag word definitions.   */

#define CAPS_EPCM        0x0001  // DS201A Extended PCM
                                 // features
#define CAPS_LPC         0x0002  // Driver plays LPC data
#define CAPS_TONE        0x0004  // Supports PCJr 3-voice
                                 // tone generator
#define CAPS_201A        0x0008  // Unit is DS201A (or
                                 // compatible)
#define CAPS_ADPCM       0x0010  // Driver plays OKI ADPCM
#define CAPS_CALLB       0x0020  // Driver supports bank
                                 // switch callback
#define CAPS_LPCVOC      0x0040  // Driver contains LPC
                                 // vocabulary.
#define CAPS_PCMREC      0x0080  // Driver supports PCM
                                 // recording
#define CAPS_DRIVERID    0x0100  // DSQuery returns Driver
                                 // ID in upper 16 bits.
#define CAPS_301         0x0200  // If this bit is set, the
                                 // following are valid:
#define CAPS_DIGI        0x0400  // Driver supports
                                 // Digispeech\CVSD\LPC
                                 // formats
#define CAPS_MIDI        0x0800  // Driver contains MIDI
                                 // interpreter.
#define CAPS_CALLIDLE    0x1000  // Driver supports Idle
                                    // callback

    /* The upper 16 bits of the flag word  */
    /* contain the driver ID code.         */

#define DID_UNK          0   // Unknown driver (early version)
#define DID_201          1   // LWDIGI DS201 driver (new)
#define DID_201A         2   // LWDIGI DS201A driver (new)
#define DID_301          3   // PDIGI Digital Audio only
#define DID_301M         4   // PDIGI Digital Audio / MIDI
                             // Synthesizer
#define DID_301V         5   // PDIGI Digital Audio / LPC
                             // Vocabulary
#define DID_301MV        6   // PDIGI All features
#define DID_301TD        7   // PDIGI Digispeech format only.
#define DID_301E         8   // PDIGI Educational LPC Emulator
#define DID_301XM        9   // PDIGI Synthesizer Emulator
```

```
#define E_OK            0x0000 // Everything is OK.
#define E_UNDEFINED     0x0001 // Undefined command.
#define E_BUF0          0x0002 // Current buffer is 0.
#define E_BUF1          0x0102 // Current buffer is 1.
#define E_MUSIC         0x0202 // Driver is in 3-tone
                               // gen mode.
#define E_COM           0x0003 // Communication error.
#define E_LPC           0x0004 // LPC index out of  range.
#define E_CVSDSPEED     0x0005 // CVSD speed is invalid.
#define E_TIMEOUT       0x0006 // Audio Unit not responding.
#define E_ERR           0x0007 // Audio Unit reported an
                               // error.
#define E_PAUSE         0x1002 // DS201 is paused.
#define E_GAIN          0x2001 // Invalid gain index.
#define E_INDEX         0x3001 // Buffer index is invalid.
#define E_LENGTH        0x4001 // Buffer length is invalid.
#define E_NOBUFFER      0x5001 // No buffers were defined.
#define E_IGNORED       0x6001 // Command ignored.
#define E_INVALID       0x6002 // Bad tone index specified.
#define E_BUSY          0x6003 // Driver or device busy
#define E_SYNTH         0x6004 // Driver is playing synth
#define E_ARGUMENT      0x7001 // Invalid argument(s).
#define E_RATE          0x7001 // Invalid RATE argument.
#define E_FORMAT        0x7002 // Invalid FORMAT argument.
#define E_MODE          0x7003 // Invalid MODE argument
#define E_VXD           0x7004 // VxD Request error.
#define E_CHANNELS      0x7005 // Invalid channel count.

    /* Digispeech file header byte codes.    */

#define H_ADPCM         0x11
#define H_PCMDVI        0x12
#define H_MSADPCM       0x13
#define H_SB4           0x14
#define H_SB3           0x15
#define H_SB2           0x16
#define H_DIGICVSD      0x44
#define H_PCMA          0x55
#define H_DIGISTD       0x77
#define H_DIGIFIX       0x78
#define H_DIGIREAL      0x79
#define H_PCMRAW        0x80
#define H_PCMRAW16      0xFE
#define H_PCMMU         0xFF
```

```
    /* Codes for data format parameter. */

#define DF_PCM8         0
#define DF_PCMMU        1
#define DF_PCMA         2
#define DF_PCM16        3
#define DF_SB2          4
#define DF_SB3          5
#define DF_SB4          6
#define DF_OKI4         7
#define DF_DVI4         8
#define DF_DIGIREAL     9
#define DF_DIGISTD      10
#define DF_DIGIFIX      11
#define DF_LPC          12
#define DF_MSADPCM      13
#define DF_CVSD         14

    /* PlayTone() command parameters.  */

#define TONE_ENABLE     0
#define TONE_DISABLE    1
#define TONE_LEVEL      2
#define TONE_FREQ       3

#define TONE_FREQ_MIN   175
#define TONE_FREQ_MAX   3000

   /* Synthesizer Emulator function parameter.   */

#define SYNTH_INIT      0
#define SYNTH_READ      1
#define SYNTH_WRITE     2
#define SYNTH_RESET     3
```

```
/*****************************************************************/
/*                                                               */
/*  DIGIDRV.H - Interrupt 4D Low Level function codes.           */
/*                                                               */
/*****************************************************************/

#define SOUND_INT        0x4D

#define AH_RESET         0x00
#define AX_RESET         0x0000

#define AH_FCVSD         0x01
#define AX_FRCVSD        0x0100
#define AX_FPCVSD        0x0101
#define AX_FRUCVSD       0x0102
#define AX_FPUCVSD       0x0103

#define AH_BLPC          0x02
#define AX_LPCSTAT       0x0200
#define AX_BLPCIX        0x0201
#define AX_BLPCBUF       0x0202

#define AH_FLPC          0x03
#define AX_FLPCIX        0x0301
#define AX_FLPCBUF       0x0302

#define AH_GENERAL       0x04
#define AX_STATUS        0x0400
#define AX_GETCNT        0x0401
#define AX_SETBUF        0x0402
#define AX_SETGAIN       0x0403
#define AX_PAUSE         0x0404
#define AX_RESUME        0x0405
#define AX_CALLBACK      0x0406
#define AX_AUDIOMIX      0x0407
#define AX_CALLIDLE      0x0408

#define AH_BCVSD         0x05
#define AX_BPCVSD        0x0500
#define AX_BRCVSD        0x0501

#define AH_BDIGI         0x06
#define AX_BPDIGI        0x0600
#define AX_FPDIGI        0x0601

#define AH_BPCM          0x07
#define AX_BPLAY         0x0700
#define AX_BPOKIADPCM    0x0701
#define AX_BREC          0x0702
#define AX_BPSTEREO      0x0703
```

```
#define AH_FMMUSIC        0x08
#define AX_MUSICINIT      0x0800
#define AX_SETSTATADDR    0x0801
#define AX_SETINSTTAB     0x0802
#define AX_SETMUSICRATE   0x0803
#define AX_TRANSPOSE      0x0804
#define AX_PLAYMUSIC      0x0805
#define AX_STOPMUSIC      0x0806
#define AX_NOTEON         0x0809
#define AX_NOTEOFF        0x080A
#define AX_PROGRAMCHG     0x080B

#define AX_GETCHANNELS    0x0880
#define AX_GETMUSICPTR    0x0881

#define AX_GETDSMENTRY     0x0885

#define AH_STONE          0x0A
#define AX_STONE          0x0A00
#define AH_CTONE          0x0B
#define AX_CTONE          0x0B00
#define AH_ETONE          0x0C
#define AX_ETONE          0x0C00

#define AH_UNINSTAL       0x64
#define AX_UNINSTAL       0x6400
#define AH_QUERY          0x65
#define AX_QUERY          0x6500
```

Appendix C - Detecting the Driver

The application program should test for existence of a DIGI TSR driver (LWDIGI or PDIGI) in memory before using the driver services. The test for both drivers is the same, including the identification string. This test can be done as follows:

> Get address of current INT 4DH interrupt handler from interrupt vector location.
>
> Add 5 (as byte offset) to this address.
>
> The contents of memory at this new address should contain the following text (not zero terminated):
>
> "Digispeech DS201 LinkWay Driver"

If this text is found, then the TSR driver is resident in memory and can be used. Otherwise the TSR should not be used until it is successfully loaded.

Example of driver detection code:

```
#define SOUND_INT   0x4D     // TSR interrupt.
#define KEY_LENGTH  31       // Length of keyID[]

static char keyID [] =
    "Digispeech DS201 LinkWay Driver";

int DriverInstalled ( void )
    {
    int     yvette;
    char    far *str1, far *str2;

        // Get pointer to DIGISPEECH TSR
        // interrupt routine.
        // Translate Interrupt vector to
        // memory address.

    _asm
        {
        mov     bx, 0         ; ES:BX == 0:(vector * 4 )
        mov     es, bx
        mov     bx, SOUND_INT
        shl     bx, 1
        shl     bx, 1

        les     bx, ES:[bx]     ; ES:BX == pointer to start
```

```
                                        ; of driver

    mov     word ptr str1, bx
    mov     word ptr str1 + 2, es
    }

    /* Set SRC to point to TSR STAMP area */

FP_OFF( str1 ) += 5;
str2 = (char far *)keyID;

    /* Check for Digispeech signature.   */

for ( yvette = 0; yvette < KEY_LENGTH; yvette++ )
    {
    if ( *(str1++) != *(str2++) )
        return( 0 );                    // Driver NOT found.
    }

return( 1 );                            // Driver found.
}
```

Appendix D - SDK Files

The following files are found on the SDK disk. Consult the README.TXT file on the disk for any updates to this list or changes to the manual.

ROOT Directory

README.TXT -	SDK Updates and changes document.
PDIGI.EXE -	PORT-ABLE Sound Driver Loader
PDRVA.DAT -	PORT-ABLE Sound standard driver
PDRVB.DAT -	PORT-ABLE Sound standard driver w/music
PDRVC.DAT -	PORT-ABLE Sound standard driver w/LPC
PDRVD.DAT -	PORT-ABLE Sound standard driver w/both
LWDIGI.EXE -	DS201/A Driver Loader
LWDRV.DAT -	DS201 standard driver
LWDRVA.DAT -	DS201A standard driver
LWDRVC.DAT -	DS201/A standard driver w/LPC
DGSETUP.EXE -	PORT-ABLE Sound Setup program
PLAY.EXE -	Executable version of play program.
RECORD.EXE -	Executable version of record program.
CALLBACK.EXE -	Executable version of callback program.
CALLIDLE.EXE -	Executable version of callidle program.
MUSIC.EXE -	Executable version of Synthesizer Interface program.
EMULATE.EXE -	Executable version of Emulator Interface program.
DIGI.CV5 -	DIGISPEECH Format CVSD File.
DEMO.OKI -	OKI ADPCM Sample file.
P1.PCM -	Linear 8 bit PCM Sample file.

LIB Directory

DIGIAPIL.LIB -	LARGE Model API library
DIGIAPIS.LIB -	SMALL Model API library

SOURCE Directory

GO.BAT -	Generates all sample programs.
PLAY.C -	Sample background play program.
RECORD.C -	Sample background record program.
CALLBACK.C -	Sample background play program.
CALLIDLE.C -	Sample background play program.
MUSIC.C -	Sample Synthesizer Interface program.
EMULATE.C -	Sample Emulator Interface program.
DIGI.H -	'C' Header file.
DIGIDRV.H -	Low-level 'C' Header file.