COVOX

Developer's Toolkit

Supporting: Sound Master II®, Voice Master®, Voice Master II®, MIDI Maestro and Speech Thing®

Release 4.01
February 25, 1992

Software

Ryan S. Hanlon
Nick Skrepetos
Donny W. Fowler

Manual

Ryan S. Hanlon
Lance A. Williams
Donny W. Fowler
Nick Skrepetos
Eric Gustafson

COVOX INC.
675 Conger St.
Eugene, OR 97402

Tel (503) 342-1271 • FAX (503) 342-1283 • BBS (503) 342-4135

Trademark Acknowledgements

Ad Lib is a trademark of Ad Lib Inc.
Borland and dBase are trademarks of Borland International, Inc.
FoxPro and FoxPro2 are trademarks of Fox Software.
GRASP is a trademark of Paul Mace Software, Inc.
IBM and IBM PC are trademarks of International Business Machines Corp.
Intel is a trademark of Intel Corporation.
Microsoft is a trademark of Microsoft Corporation.
Show Partner is a trademark of Brightbill Roberts and Company, Ltd.
Tandy is a trademark of Tandy Corporation.
Yamaha is a trademark of Yamaha LSI.

Covox Software License

1. **Grant of License.** Covox grants to you the right to use one copy of the enclosed software program (the "SOFTWARE") on a single terminal connected to a single computer (i.e. with a single CPU). You may not network the SOFTWARE or otherwise use it on more than one computer or computer terminal at the same time.

2. **Copyright, Patents, and Trademarks.** The SOFTWARE is owned by Covox, Inc. or its suppliers and is protected by United States copyright laws and international treaty provisions. Therefore, you must treat the SOFTWARE like any other copyrighted material (e.g. a book or musical recording) except that you may either (a) make a copy of the SOFTWARE solely for backup or archival purposes, or (b) transfer the SOFTWARE to a single hard disk provided you keep the original solely for backup or archival purposes. You may not copy the written material accompanying the SOFTWARE. The voice recognition, data compression routines, and other routines that might be present in this disk package are subject to existing patents and patents pending. The names COVOX, VOICE MASTER, SPEECH THING, SOUND MASTER, VOICE HARP, and the Covox logo are registered trademarks of Covox, Inc.

3. **Other Restrictions.** You may not reverse engineer, decompile, or disassemble the compiled SOFTWARE.

4. **Development Software.** The compiled library routines supplied by Covox may be used in application software on a royalty-free basis provided that you: (a) distribute the compiled and linked library routines only in conjunction with and as a part of your software product; (b) the sign-on message for your software product must display that the sound and/or speech software is the copyright of Covox, Inc.; (c) that the owners manual for your software product must state in a conspicuous location that the sound and/or speech software is licensed from Covox, Inc., that Covox hardware products are supported, and where Covox products might be obtained; (d) that if the SOFTWARE is used to support non-Covox sound hardware, Covox hardware must also be supported by your software product, and; (e) you agree to indemnify, hold harmless, and defend Covox from and against any claims or lawsuits, including attorney fees, that arise or result from the use or distribution of our software product.

5. **Exceptions.** No part of the "SmoothTalker" text-to-speech software (SPEECHV2.EXE, SPEECHV3.EXE, STALK.EXE, or STDRIVER.SYS) may be incorporated within your software product.

Limited Warranty Statement

Covox, Inc. guarantees the software disks to be free of manufacturing or duplication defects for a period of one year from the date of purchase. Covox, Inc. will replace the diskette free of charge, provided it is returned to the factory via prepaid transportation.

Liability Disclaimer

Software is provided on an "as is" basis. Covox, Inc. disclaims liability for direct, indirect or incidental damages arising from the use of this software, including but not limited to the interruption of service, loss of business or potential profits, legal actions or other consequential damages even if Covox has been advised of the possibility of such damages.

Control of environmental factors by means of voice could expose the user to some risk. Automatic voice recognition by machine remains an unreliable technology due to uncontrollable variations in the way that normal speech is produced in an uncertain acoustic environment. Covox, Inc. specifically disclaims liability as stated in the preceding paragraph when applied to voice recognition.

Software Update Policy

Covox, Inc. reserves the right to incorporate software improvements in products without prior notice and without obligation to replace previously issued software. Updated software is available from the company at nominal cost. Current product information will be sent to customers of record, namely, those that purchased directly from Covox, Inc. and/or returned warranty cards.

TABLE OF CONTENTS

INTRODUCTION

Welcome to the Covox Developer's Toolkit. This package allows you to incorporate music, voice and other sound effects into your own software applications for the IBM PC and its compatibles. All Covox products are supported, including the Speech Thing, Voice Master Key System, Voice Master Key System II, MIDI Maestro and Sound Master II.

The Developer's Toolkit manual is separated into thirteen chapters. The first six chapters document the input and output of digitized sound, and are divided as follows:

1. Non-DMA Record and Play Functions.
2. Buffer Pack and Unpack Functions.
3. DMA Record and Playback Functions.
4. TSRPLAY Functions.
5. BIOS Interrupt 1A Sound Support.
6. FoxPro and dBase Sound Support.

Programmers interested in the musical capabilities of the Covox MIDI Maestro and Sound Master II cards should refer to the following two chapters:

7. FM Synthesizer Functions.
8. MIDI Functions.

Chapter nine consists of general utility functions.

9. Covox Utility Functions.

The final four chapters of the manual describe, in low-level detail, the basics of programming the Covox hardware interfaces.

10. Programming With Polled (non-DMA) I/O.
11. Programming With Direct Memory Access.
12. Programming the FM Synthesizer.
13. Programming MIDI.

Toolkit Filename Conventions

The Developer's Toolkit includes small, medium, compact and large model libraries compiled with Microsoft C 6.0 and Borland C 2.0. These object libraries include all the necessary functions for programming digitized I/O, MIDI and the FM Synthesizer.

Library Filename Conventions

The libraries are named according to model size, type of tool, compiler, and language, as follows:

```
<model> <tool type> <compiler> <language>.lib
```

model: 'S', 'M', 'C' or 'L' for the library model size.

tool type: 'MIDI' for MIDI tools.
'DIGI' for Digitized I/O tools.
'FMSY' for FM Synthesizer tools.
'UTIL' for Utility tools.

compiler: 'M' or 'B' indicating Microsoft or Borland.

language: 'C' for the C programming language.

Examples

SMIDIBC.LIB

- Small model
- MIDI tools
- Borland compiler
- C language

LDIGIMC.LIB

- Large model
- Digitized I/O tools
- Microsoft compiler
- C language

Header Filename Conventions

The appropriate header file must be included when using the Covox libraries.

```
cvx <tool type>.h
```

"cvx" Stands for Covox.

tool type Corresponds to the value of *<tool type>* in the above libraries.
'MIDI' for MIDI.
'DIGI' for Digitized I/O.
'FMSY' for FM Synthesizer.
'UTIL' for utility functions.

Library Header Files

When using the Toolkit libraries, the appropriate header files must be included at the beginning of the program performing the function calls. The following header files are printed out in **Appendix A** for convenient reference.

CVXDIGI.H A header file to be included with all programs using the Digitized I/O functions in Chapters 1 through 4.

CVXFMSY.H A header file to be included with all programs using the FM functions in Chapter 7.

CVXMIDI.H A header file to be included with all programs using the MIDI functions in Chapter 8.

CVXUTIL.H A header file to be included with all programs using the utility functions in Chapter 9.

CVXBRLND.H A header file automatically included with programs written in Borland C (version 2.0).

Note: the above header files all include **CVXDEFS.H**.

CVXDEFS.H The following type definition statements are taken from the **CVXDEFS.H** file. These definitions are used in all library header files and in the example code throughout the Toolkit. They have been included for ease of use, code compaction and code portability.

```
typedef  void                VOID;
typedef  unsigned            BOOL;
typedef  unsigned            WORD;
typedef  unsigned char       BYTE;
typedef  unsigned long       LONG;
typedef  unsigned long       DWORD;

typedef  BYTE  *             PSTR;
typedef  BYTE  *             PBYTE;
typedef  WORD  *             PWORD;
typedef  LONG  *             PLONG;
typedef  VOID  *             PVOID;

typedef  BYTE  far   *       LPSTR;
typedef  BYTE  far   *       LPBYTE;
typedef  WORD  far   *       LPWORD;
typedef  LONG  far   *       LPLONG;
typedef  VOID  far   *       LPVOID;

typedef  BYTE  huge  *       HPSTR;
typedef  BYTE  huge  *       HPBYTE;
typedef  WORD  huge  *       HPWORD;
typedef  LONG  huge  *       HPLONG;
typedef  VOID  huge  *       HPVOID;

typedef  unsigned            HANDLE;
```

CHAPTER 1

Non-DMA Record and Play Functions

High-Level Functions

play2s	record2s
play3s	record3s
play4s	record4s
play8	record8
play8s	record8s
playAny	

Low-Level Functions

playByte	recordByte
playByteInit	recordByteInit
playByteUninit	recordByteUninit

This chapter describes both high and low level functions for recording and playback.

The high level functions, documented first in this chapter, use a method of I/O called polling and require no initialization.

The **record8** and **play8** functions record and play back industry standard 8 bit PCM (Pulse Code Modulation) sound data. The **record8s** and **play8s** functions record 8 bit PCM sound data with optional silence encoding.

The **record2s, record3s** and **record4s** functions record Covox's own brand of ADPCM (Adaptive Differential PCM) sound data with optional silence encoding. The **play2s, play3s** and **play4s** functions play back the ADPCM sound data.

Silence encoding may be adjusted or completely disabled in all of the record functions that utilize silence encoding.

The low level play and record functions must be initialized before use and uninitialized when finished.

To record sound data using the low level functions, first initialize recording with **recordByteInit**. Use the **recordByte** function to input the data, then use **recordByteUninit** to deactivate recording.

To play back sound data using the low level functions, first initialize playback with **playByteInit**. Use the **playByte** function to output the data, then use **playByteUninit** to deactivate playback.

Description Play a 2, 3 or 4 bit ADPCM sound buffer recorded with or without silence encoding.

Prototype

WORD play2s(LPSTR *bufferStart*, **LONG** *bufferSize*, **BYTE** *playbackRate*, **WORD** *portOut*, **BOOL** *trebleFlag*, **BOOL** *noiseFlag* **);**

WORD play3s(LPSTR *bufferStart*, **LONG** *bufferSize*, **BYTE** *playbackRate*, **WORD** *portOut*, **BOOL** *trebleFlag*, **BOOL** *noiseFlag* **);**

WORD play4s(LPSTR *bufferStart*, **LONG** *bufferSize*, **BYTE** *playbackRate*, **WORD** *portOut*, **BOOL** *trebleFlag*, **BOOL** *noiseFlag* **);**

Parameters

bufferStart A far pointer (**LPSTR**) to the sound buffer.

bufferSize A **LONG** value indicating the number of bytes to be played from the buffer pointed to by *bufferStart*.

If bufferSize is not evenly divisible by 16, the remainder is ignored. If bufferSize is smaller than 16, the playback functions return an error (_ERROR_INVALID_BUFFER_SIZE).

playbackRate A **BYTE** value from 0 to 229, indicating the sampling rate at which to play back the sound buffer.

If *playbackRate* is 0, these functions look for the Covox 16-byte header at the beginning of the buffer pointed to by *bufferStart*. If no header exists or the rate in the header is 0, the default value of 132 (9622 Hz) is used.

If *playbackRate* is larger than 229, it is set to 229 (44,191 Hz).

See **Appendix B** for rate value descriptions.

portOut This **WORD** value designates which DAC port to use for playback. If the value of *portOut* is not one of the Covox logical port values shown below, it represents the physical port address of the DAC.

The Covox logical port values are:

_CVX_VM0	**_CVX_LPT1**
_CVX_VM1	**_CVX_LPT2**
_CVX_VM2	**_CVX_LPT3**
_CVX_VM3	**_CVX_ISP**

See **Appendix C** for port value descriptions.

trebleFlag A flag which, if **_TRUE**, indicates that the output is to be differentiated. If **_FALSE**, there is no differentiation.

Differentiation is used to achieve the effects of a high-pass filter, which enhances the upper frequencies of the digitized sound.

noiseFlag A flag which, if **_TRUE**, indicates that computer generated noise is to be output during silence periods. If **_FALSE**, silence periods are played back as dead silence.

Remarks These functions automatically detect and convert silence encoding if it exists in the sound file. If **_ioStopFlag** is not set to **_TRUE** during playback, the function will play back the entire buffer, up to the limit specified by *bufferSize*.

See **Appendix D** for information on usage of **_ioStopFlag**.

Return Value If successful, these functions return a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_PORT_INIT_FAILED
_ERROR_INVALID_BUFFER_SIZE

See the last page of the chapter for error code descriptions.

Example

```
// PLAY3S.C
// This program uses the non-DMA play3s function. Memory is
// allocated for the sound data. A file name TEST1.V3S is opened
// and its contents read into a buffer. Finally, the allocated
// memory is freed.

#include <fcntl.h>
#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE      0x8000
#define _SOUND_FILE       "TEST1.V3S"
#define _TREBLE_FLAG      _FALSE
#define _NOISE_FLAG       _FALSE

VOID main( VOID )
{
   HANDLE    fileHandle;
   WORD      bytesRead;
   WORD      bufferSegment;
   LPSTR     playBuffer;
```

```
    // Allocate memory for playback buffer.
    if( ( _dos_allocmem( ( _BUFFER_SIZE / 16  + 1 ),
                         &bufferSegment ) ) )
       printf( "ERROR : Cannot Allocate Memory!!\n" );
    else
    {
       // play3s requires a far pointer (LPSTR).
       FP_SEG( playBuffer ) = bufferSegment;
       FP_OFF( playBuffer ) = 0x0000;

       // Open the file containing sound data.
       if( _dos_open( _SOUND_FILE, O_RDONLY, &fileHandle ) )
          printf( "ERROR : %s not found.\n", _SOUND_FILE );
       else
          // Read sound data from file.
          _dos_read( fileHandle, playBuffer, _BUFFER_SIZE,
                     &bytesRead );

       // Prompt user to hit a key to begin playback.
       printf( "Hit any key to begin playback.\n" );

       // Wait for keystroke.
       while( !_bios_keybrd( _KEYBRD_READY ) );

       // Clear keystroke(s) from keyboard buffer.
       while( _bios_keybrd( _KEYBRD_READY ) )
          getch();

       // Notify user that playback has begun.
       printf( "Playing sound file ..." );

       // Playback a file of length bytesRead.
       play3s( playBuffer, ( LONG )bytesRead, _CVX_RATE_DEFAULT,
               _CVX_VM0, _TREBLE_FLAG, _NOISE_FLAG );

       // Notify user that playback has completed.
       printf( " complete.\n" );

       // Close file containing sound data.
       _dos_close( fileHandle );

       // Free memory used by buffer.
       if( _dos_freemem( bufferSegment ) )
          printf( "ERROR : Cannot Free Memory!!\n" );
    }
}
```

Description Plays back a standard 8 bit PCM (Pulse Coded Modulation) sound buffer.

Prototype **WORD play8(LPSTR** *bufferStart*, **LONG** *bufferSize*, **BYTE** *playbackRate*, **WORD** *portOut*, **BOOL** *trebleFlag* **);**

Parameters

bufferStart A far pointer (**LPSTR**) to the sound buffer.

bufferSize A **LONG** value indicating the number of bytes to be played from the buffer pointed to by *bufferStart*.

If bufferSize is not evenly divisible by 16, the remainder is ignored. If bufferSize is smaller than 16, the record functions return an error (_ERROR_INVALID_BUFFER_SIZE).

playbackRate A **BYTE** value from 0 to 229, indicating the sampling rate at which to play back the sound buffer.

If *playbackRate* is 0, **play8** looks for the Covox 16-byte header at the beginning of the buffer pointed to by *bufferStart*. If no header exists or the rate in the header is 0, the default value of 132 (9622 Hz) is used.

If *playbackRate* is larger than 229, it is set to 229 (44,191 Hz).

See **Appendix B** for rate value descriptions.

portOut This **WORD** value designates which DAC port to use for playback. If the value of *portOut* is not one of the Covox logical port values shown below, it represents the physical port address of the DAC.

The Covox logical port values are:

_CVX_VM0	**_CVX_LPT1**
_CVX_VM1	**_CVX_LPT2**
_CVX_VM2	**_CVX_LPT3**
_CVX_VM3	**_CVX_ISP**

See **Appendix C** for port value descriptions.

trebleFlag A flag which, if **_TRUE**, indicates that the output is to be differentiated. If **_FALSE**, there is no differentiation.

Differentiation is used to acheive the effects of a high-pass filter, which enhances the upper frequencies of the digitized sound.

Remarks This function plays back an industry standard 8 bit PCM sound buffer (i.e. a buffer recorded with **record8**). If **_ioStopFlag** is not set to **_TRUE** during playback, the function will play back the entire buffer, up to the limit specified by *bufferSize*.

See **Appendix D** for information on usage of **_ioStopFlag**.

Return Value If successful, **play8** returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_PORT_INIT_FAILED
_ERROR_INVALID_BUFFER_SIZE
_ERROR_PLAY8_BUSY

See the last page of the chapter for error code descriptions.

Example

```
// PLAY8.C
// This program uses the non-DMA play8 function. Memory is
// allocated for the sound data. A file name TEST1.V8 is opened
// and its contents read into a buffer. Finally, the allocated
// memory is freed.

#include <fcntl.h>
#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE      0x8000
#define _SOUND_FILE       "TEST1.V8"
#define _TREBLE_FLAG      _FALSE

VOID main( VOID )
{
   HANDLE    fileHandle;
   WORD      bytesRead;
   WORD      bufferSegment;
   LPSTR     playBuffer;

   // Allocate memory for playback buffer.
   if( ( _dos_allocmem( ( _BUFFER_SIZE / 16 + 1 ),
                        &bufferSegment ) ) )
      printf( "ERROR : Cannot Allocate Memory!!\n" );
   else
   {
      // play8 requires a far pointer (LPSTR).
      FP_SEG( playBuffer ) = bufferSegment;
      FP_OFF( playBuffer ) = 0x0000;

      // Open the file containing sound data.
      if( _dos_open( _SOUND_FILE, O_RDONLY, &fileHandle ) )
         printf( "ERROR : %s not found.\n", _SOUND_FILE );
```

```
        else
            // Read sound data from file.
            _dos_read( fileHandle, playBuffer, _BUFFER_SIZE,
                       &bytesRead );

        // Prompt user to hit a key to begin playback.
        printf( "Hit any key to begin playback.\n" );

        // Wait for keystroke.
        while( !_bios_keybrd( _KEYBRD_READY ) );

        // Clear keystroke(s) from keyboard buffer.
        while( _bios_keybrd( _KEYBRD_READY ) )
            getch();

        // Notify user that playback has begun.
        printf( "Playing sound file ..." );

        // Playback a file of length bytesRead.
        play8( playBuffer, ( LONG )bytesRead, _CVX_RATE_DEFAULT,
               _CVX_VM0, _TREBLE_FLAG );

        // Notify user that playback has completed.
        printf( " complete.\n" );

        // Close file containing sound data.
        _dos_close( fileHandle );

        // Free memory used by buffer.
        if( _dos_freemem( bufferSegment ) )
            printf( "ERROR : Cannot Free Memory!!\n" );
    }
}
```

Description Plays back an 8 bit PCM sound buffer recorded with or without silence encoding.

Prototype **WORD play8s(LPSTR** *bufferStart*, **LONG** *bufferSize*, **BYTE** *playbackRate*, **WORD** *portOut*, **BOOL** *trebleFlag*, **BOOL** *noiseFlag* **);**

Parameters

bufferStart A far pointer (**LPSTR**) to the sound buffer.

bufferSize A **LONG** value indicating the number of bytes to be played from the buffer pointed to by *bufferStart*.

If bufferSize is not evenly divisible by 16, the remainder is ignored. If bufferSize is smaller than 16, the record functions will return an error (_ERROR_INVALID_BUFFER_SIZE).

playbackRate A **BYTE** value from 0 to 229, indicating the sampling rate at which to play back the sound buffer.

If *playbackRate* is 0, **play8s** looks for the Covox 16-byte header at the beginning of the buffer pointed to by *bufferStart*. If no header exists or the rate in the header is 0, the default value of 132 (9622 Hz) is used.

If *playbackRate* is larger than 229, it is set to 229 (44,191 Hz).

See **Appendix B** for rate value descriptions.

portOut This **WORD** value designates which DAC port to use for playback. If the value of *portOut* is not one of the Covox logical port values shown below, it represents the physical port address of the DAC.

The Covox logical port values are:

_CVX_VM0	**_CVX_LPT1**
_CVX_VM1	**_CVX_LPT2**
_CVX_VM2	**_CVX_LPT3**
_CVX_VM3	**_CVX_ISP**

See **Appendix C** for port value descriptions.

trebleFlag A flag which, if **_TRUE**, indicates that the output is to be differentiated. If **_FALSE**, there is no differentiation.

Differentiation is used to acheive the effects of a high-pass filter, which enhances the upper frequencies of the digitized sound.

noiseFlag A flag which, if **_TRUE**, indicates that computer generated noise is to be output during silence periods. If **_FALSE**, silence periods are played back as dead silence.

Remarks This function automatically detects and converts silence encoding if it exists in the sound file. If **_ioStopFlag** is not set to **_TRUE** during playback, the function will play back the entire buffer, up to the limit specified by *bufferSize*.

See **Appendix D** for information on usage of **_ioStopFlag**.

Return Value If successful, **play8s** returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_PORT_INIT_FAILED
_ERROR_INVALID_BUFFER_SIZE

See the last page of the chapter for error code descriptions.

Example

```
// PLAY8S.C
// This program uses the non-DMA play8s function. Memory is
// allocated for the sound data. A file name TEST1.V8S is opened
// and its contents read into a buffer. Finally, the allocated
// memory is freed.

#include <fcntl.h>
#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE      0x8000
#define _SOUND_FILE       "TEST1.V8S"
#define _TREBLE_FLAG      _FALSE
#define _NOISE_FLAG       _FALSE

VOID main( VOID )
{
   HANDLE    fileHandle;
   WORD      bytesRead;
   WORD      bufferSegment;
   LPSTR     playBuffer;

   // Allocate memory for playback buffer.
   if( ( _dos_allocmem( ( _BUFFER_SIZE / 16 + 1 ),
                        &bufferSegment ) ) )
```

```
        printf( "ERROR : Cannot Allocate Memory!!\n" );
    else
    {
        // play8s requires a far pointer (LPSTR).
        FP_SEG( playBuffer ) = bufferSegment;
        FP_OFF( playBuffer ) = 0x0000;

        // Open the file containing sound data.
        if( _dos_open( _SOUND_FILE, O_RDONLY, &fileHandle ) )
            printf( "ERROR : %s not found.\n", _SOUND_FILE );
        else
            // Read sound data from file.
            _dos_read( fileHandle, playBuffer, _BUFFER_SIZE,
                       &bytesRead );

        // Prompt user to hit a key to begin playback.
        printf( "Hit any key to begin playback.\n" );

        // Wait for keystroke.
        while( !_bios_keybrd( _KEYBRD_READY ) );

        // Clear keystroke(s) from keyboard buffer.
        while( _bios_keybrd( _KEYBRD_READY ) )
            getch();

        // Notify user that playback has begun.
        printf( "Playing sound file ..." );

        // Playback a file of length bytesRead.
        play8s( playBuffer, ( LONG )bytesRead, _CVX_RATE_DEFAULT,
                _CVX_VM0, _TREBLE_FLAG, _NOISE_FLAG );

        // Notify user that playback has completed.
        printf( " complete.\n" );

        // Close file containing sound data.
        _dos_close( fileHandle );

        // Free memory used by buffer.
        if( _dos_freemem( bufferSegment ) )
            printf( "ERROR : Cannot Free Memory!!\n" );
    }
}
```

Description Plays back a sound buffer of any Covox format.

Prototype **WORD playAny(LPSTR** *bufferStart*, **LONG** *bufferSize*, **BYTE** *playbackRate*, **WORD** *portOut*, **BOOL** *trebleFlag*, **BOOL** *noiseFlag* **);**

Parameters

bufferStart A far pointer (**LPSTR**) to the sound buffer.

bufferSize A **LONG** value indicating the number of bytes to be played from the buffer pointed to by *bufferStart*.

If bufferSize is not evenly divisible by 16, the remainder is ignored. If bufferSize is smaller than 16, the record functions will return an error (_ERROR_INVALID_BUFFER_SIZE).

playbackRate A **BYTE** value from 0 to 229, indicating the sampling rate at which to play back the sound buffer.

If *playbackRate* is 0, **playAny** reads the file header for the rate at which it was recorded. If no header exists or the rate in the header is 0, the default value of 132 (9622 Hz) is used.

If *playbackRate* is larger than 229, it is set to 229 (44,191 Hz).

See **Appendix B** for rate value descriptions.

portOut This **WORD** value designates which DAC port to use for playback. If the value of *portOut* is not one of the Covox logical port values shown below, it represents the physical port address of the DAC.

The Covox logical port values are:

_CVX_VM0	**_CVX_LPT1**
_CVX_VM1	**_CVX_LPT2**
_CVX_VM2	**_CVX_LPT3**
_CVX_VM3	**_CVX_ISP**

See **Appendix C** for port value descriptions.

trebleFlag A flag which, if **_TRUE**, indicates that the output is to be differentiated. If **_FALSE**, there is no differentiation.

Differentiation is used to acheive the effects of a high-pass filter, which enhances the upper frequencies of the digitized sound.

noiseFlag A flag which, if **_TRUE**, indicates that computer generated noise is to be output during silence periods instead of dead silence.

Remarks This function plays back a Covox sound buffer of any format previously mentioned in this chapter. The packing format is retrieved from the sound file header. If no header is present then the data is assumed to be 8 bit PCM.

If **_ioStopFlag** is not set to **_TRUE** during playback, the function will play back the entire buffer, up to the limit specified by *bufferSize*.

See **Appendix D** for information on usage of **_ioStopFlag**.

Return Value If successful, **playAny** returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_PORT_INIT_FAILED
_ERROR_INVALID_BUFFER_SIZE

See the last page of the chapter for error code descriptions.

Example

```
// PLAYANY.C
// This program uses the non-DMA playAny() function. Memory is
// allocated for the sound data. A file name TEST1.V2S is opened
// and its contents read into the buffer used with the call to
// playAny(). Finally, the allocated memory is freed.

#include <fcntl.h>
#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE  0x8000
#define _SOUND_FILE   "TEST1.V2S"
#define _TREBLE_FLAG  _FALSE
#define _NOISE_FLAG   _TRUE

VOID main( VOID )
{
   HANDLE    fileHandle;
   WORD      bytesRead;
   WORD      bufferSegment;
   LPSTR     playBuffer;

   // Allocate memory for playback buffer.
   if( ( _dos_allocmem( ( _BUFFER_SIZE / 16 + 1 ),
                        &bufferSegment ) ) )
      printf( "ERROR : Cannot Allocate Memory!!\n" );
   else
```

```
    {
        // playAny() requires a far pointer (LPSTR).
        FP_SEG( playBuffer ) = bufferSegment;
        FP_OFF( playBuffer ) = 0x0000;

        // Open the file containing sound data.
        if( _dos_open( _SOUND_FILE, O_RDONLY, &fileHandle ) )
            printf( "ERROR : %s not found.\n", _SOUND_FILE );
        else
        {
            // Read sound data from file.
            _dos_read( fileHandle, playBuffer, _BUFFER_SIZE, &bytesRead );

            // Play a file of length bytesRead.
            playAny( playBuffer, ( LONG )bytesRead, _CVX_RATE_DEFAULT,
                    _CVX_VM0, _TREBLE_FLAG, _NOISE_FLAG );

            // Close file containing sound data.
            _dos_close( fileHandle );

            // Free memory used by buffer.
            if( _dos_freemem( bufferSegment ) )
                printf( "ERROR : Cannot Free Memory!!\n" );
        }
    }
}
```

Description Record a 2, 3 or 4 bit ADPCM sound buffer with optional silence encoding.

Prototype

WORD record2s(LPSTR *bufferStart*, **LONG** *bufferSize*, **BYTE** *recordRate*, **WORD** *portIn*, **WORD** *silenceRange*, **LONG *** *bytesRecorded*);

WORD record3s(LPSTR *bufferStart*, **LONG** *bufferSize*, **BYTE** *recordRate*, **WORD** *portIn*, **WORD** *silenceRange* **LONG *** *bytesRecorded*);

WORD record4s(LPSTR *bufferStart*, **LONG** *bufferSize*, **BYTE** *recordRate*, **WORD** *portIn*, **WORD** *silenceRange* **LONG *** *bytesRecorded*);

Parameters

bufferStart A far pointer (**LPSTR**) to the buffer that is to contain the recorded data.

bufferSize A **LONG** value indicating the number of bytes in the buffer pointed to by *bufferStart*.

If bufferSize is not evenly divisible by 16, the remainder is ignored. If bufferSize is less than 32, the record functions will return an error (_ERROR_INVALID_BUFFER_SIZE).

recordRate A **BYTE** value from 0 to 209, indicating the sampling rate at which to record the sound buffer.

If *recordRate* is 0, the default value of 132 (9622 Hz) is used. If *recordRate* is larger than 209, it is set to 209 (25,386 Hz).

See **Appendix B** for rate value descriptions.

portIn This **WORD** value designates which ADC port to use for recording. If the value of *portIn* is not one of the Covox logical port values shown below, it represents the physical port address of the ADC.

The Covox logical port values are:

_CVX_VM0	**_CVX_LPT1**
_CVX_VM1	**_CVX_LPT2**
_CVX_VM2	**_CVX_LPT3**
_CVX_VM3	

See **Appendix C** for port value descriptions.

silenceRange A **WORD** value that determines the range of sampled values interpreted as silence. Note that all ranges center on 0x80, as this value represents actual silence. The value passed represents the following hexadecimal ranges:

_SILENCE_0	**_SILENCE_3**
_SILENCE_1	**_SILENCE_4**
_SILENCE_2	**_SILENCE_5**

See **CVXDIGI.H** in **Appendix A** for silence range descriptions.

bytes-Recorded A pointer to a variable of type **LONG**, representing the number of bytes actually recorded by the function.

Remarks Using **record2s**, a typical recording will be about 1/4 the size of an 8 bit PCM file. Using **record3s**, a typical recording will be about 1/3 the size of an 8 bit PCM file. Using **record4s**, a typical recording will be about 1/2 the size of an 8 bit PCM file. In all three cases, the resulting sound file may be much shorter if silence encoding is used and there are long periods of silence during recording.

A 16-byte header, described in **Appendix B**, precedes the resulting file.

If **_ioStopFlag** is not set to **_TRUE** during recording, the entire buffer will be filled with ADPCM data before recording is terminated. See **Appendix D** for information on usage of **_ioStopFlag**.

Return Value If successful, these functions return a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_PORT_INIT_FAILED
_ERROR_INVALID_BUFFER_SIZE

See the last page of this chapter for error code descriptions.

Example

```
// RECORD3S.C
// This program uses the non-DMA record3s function. Memory is
// allocated for the record, the recording is performed, and the
// recorded buffer is written to a file. The memory allocated for
// the record buffer is freed.

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE      0x8000
#define _SOUND_FILE       "TEST1.V3S"
```

```
VOID main( VOID )
{
   LONG     bytesRecorded;
   HANDLE   fileHandle;
   WORD     bytesWritten;
   WORD     bufferSegment;
   LPSTR    recordBuffer;

   // Allocate memory for recording buffer. Segment of
   // allocated memory is returned in bufferSegment.
   if( ( _dos_allocmem( ( _BUFFER_SIZE / 16 + 1 ),
                        &bufferSegment ) ) )
      printf( "ERROR : Cannot Allocate Memory!!\n" );
   else
   {
      // record3s requires a far pointer (LPSTR).
      FP_SEG( recordBuffer ) = bufferSegment;
      FP_OFF( recordBuffer ) = 0x0000;

      // Prompt user to hit a key to begin recording.
      printf( "Hit any key to begin recording\n" );

      // Wait for keystroke.
      while( !_bios_keybrd( _KEYBRD_READY ) );

      // Clear keystroke(s) from keyboard buffer.
      while( _bios_keybrd( _KEYBRD_READY ) )
         getch();

      // Notify user that recording has begun.
      printf( "Recording ..." );

      // Record a file of length _BUFFER_SIZE.
      if( record3s( recordBuffer, ( LONG )_BUFFER_SIZE,
                    _CVX_RATE_DEFAULT, _CVX_VM0,
                    _SILENCE_3, &bytesRecorded ) )
      {
         // Notify user that error occured during record.
         printf( " aborted. Error during record3s().\n" );
      }
      else
      {
         // Notify user that recording has completed.
         printf( " complete.\n" );
      }

      // If bytesRecorded is not zero, no error was encountered.
      if( bytesRecorded )
      {
         // Open a new file to write sound data into.
         if( _dos_creat( _SOUND_FILE, _A_NORMAL, &fileHandle ) )
         {
            printf( "ERROR : Cannot create file.\n" );
            printf( "        Sound data lost.\n" );
         }
         else
         {
            // Write sound data to file.
            _dos_write( fileHandle, recordBuffer,
                        ( WORD )bytesRecorded, &bytesWritten );
```

```
            // Close file containing sound data.
            _dos_close( fileHandle );
         }
      }

      // Free memory used by buffer.
      if( _dos_freemem( bufferSegment ) )
         printf( "ERROR : Cannot Free Memory!!\n" );
   }
}
```

Description Records a standard 8 bit PCM (Pulse Coded Modulation) file.

Prototype **WORD record8(LPSTR** *bufferStart*, **LONG** *bufferSize*, **BYTE** *recordRate*, **WORD** *portIn*, **LONG *** *bytesRecorded* **);**

Parameters

bufferStart A far pointer (**LPSTR**) to the buffer that is to contain the recorded data.

bufferSize A **LONG** value indicating the number of bytes in the buffer pointed to by *bufferStart*.

If bufferSize is not evenly divisible by 16, the remainder is ignored. If bufferSize is less than 32 then the record functions will return an error (_ERROR_INVALID_BUFFER_SIZE).

recordRate A **BYTE** value from 0 to 209, indicating the sampling rate at which to record the sound buffer.

If *recordRate* is 0, the default value of 132 (9622 Hz) is used. If *recordRate* is larger then 209, it is set to 209 (25,386 Hz).

See **Appendix B** for rate value descriptions.

portIn This **WORD** value designates which ADC port to use for recording. If the value of *portIn* is not one of the Covox logical port values shown below, it represents the physical port address of the ADC.

The Covox logical port numbers are:

_CVX_VM0	**_CVX_LPT1**
_CVX_VM1	**_CVX_LPT2**
_CVX_VM2	**_CVX_LPT3**
_CVX_VM3	

See **Appendix C** for port value descriptions.

bytes-Recorded A pointer to a variable of type **LONG**, representing the number of bytes actually recorded by the function.

Remarks This function records an industry standard 8 bit PCM (Pulse Coded Modulation) file. A 16-byte header, described in **Appendix B**, precedes the resulting file.

If **_ioStopFlag** is not set to **_TRUE** during recording, the entire buffer will be filled with PCM data before recording is terminated. See **Appendix D** for information on usage of **_ioStopFlag**.

Return Value If successful, these functions return a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_PORT_INIT_FAILED
_ERROR_INVALID_BUFFER_SIZE

See the last page of this chapter for error code descriptions.

Example

```
// RECORD8.C
// This program uses the non-DMA record8 function. Memory is
// allocated for the record. The recording is performed and the
// recorded buffer is written to a file. The memory allocated for
// the record buffer is freed.

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE      0x8000
#define _SOUND_FILE       "TEST1.V8"

VOID main( VOID )
{
   LONG      bytesRecorded;
   HANDLE    fileHandle;
   WORD      bytesWritten;
   WORD      bufferSegment;
   LPSTR     recordBuffer;

   // Allocate memory for recording buffer. Segment of the
   // allocated memory is returned in bufferSegment.
   if( ( _dos_allocmem( ( _BUFFER_SIZE / 16 + 1 ),
                        &bufferSegment ) ) )
      printf( "ERROR : Cannot Allocate Memory!!\n" );
   else
   {
      // record8 requires a far pointer (LPSTR).
      FP_SEG( recordBuffer ) = bufferSegment;
      FP_OFF( recordBuffer ) = 0x0000;

      // Prompt user to hit a key to begin recording.
      printf( "Hit any key to begin recording\n" );

      // Wait for keystroke.
      while( !_bios_keybrd( _KEYBRD_READY ) );

      // Clear keystroke(s) from keyboard buffer.
      while( _bios_keybrd( _KEYBRD_READY ) )
```

```
            getch();

         // Notify user that recording has begun.
         printf( "Recording ..." );

         // Record a file of length _BUFFER_SIZE.
         if( record8( recordBuffer, ( LONG )_BUFFER_SIZE,
                      _CVX_RATE_DEFAULT, _CVX_VM0,
                      &bytesRecorded ) )
         {
            // Notify user that error occured during record.
            printf( " aborted. Error during record8().\n" );
         }
         else
         {
            // Notify user that recording has completed.
            printf( " complete.\n" );
         }

         // If bytesRecorded is not zero, no error was encountered.
         if( bytesRecorded )
         {
            // Open a new file to write sound data into.
            if( _dos_creat( _SOUND_FILE, _A_NORMAL, &fileHandle ) )
            {
               printf( "ERROR : Cannot create file.\n" );
               printf( "        Sound data lost.\n" );
            }
            else
            {
               // Write sound data to file.
               _dos_write( fileHandle, recordBuffer,
                           ( WORD )bytesRecorded, &bytesWritten );

               // Close file containing sound data.
               _dos_close( fileHandle );
            }
         }


         // Free memory used by buffer.
         if( _dos_freemem( bufferSegment ) )
            printf( "ERROR : Cannot Free Memory!!\n" );
      }
   }
```

Description Records 8 bit PCM speech with optional silence encoding.

Prototype **WORD record8s(LPSTR** ***bufferStart*****, LONG** ***bufferSize*****, BYTE** ***recordRate*****, WORD** ***portIn*****, WORD** ***silenceRange*****, LONG *** ***bytesRecorded*** **);**

Parameters

bufferStart A far pointer (**LPSTR**) to the buffer that is to contain the recorded data.

bufferSize A **LONG** value indicating the number of bytes in the buffer pointed to by ***bufferStart***.

If bufferSize is not evenly divisible by 16, the remainder is ignored. If bufferSize is less than 32 then the record functions will return an error (_ERROR_INVALID_BUFFER_SIZE).

recordRate A **BYTE** value from 0 to 209, indicating the sampling rate at which to record the sound buffer.

If ***recordRate*** is 0, the default value of 132 (9622 Hz) is used. If ***recordRate*** is larger than 209, it is set to 209 (25,386 Hz).

See **Appendix B** for rate value descriptions.

portIn This **WORD** value designates which ADC port to use for recording. If the value of ***portIn*** is not one of the Covox logical port values shown below, it represents the physical port address of the ADC.

The Covox logical port values are:

_CVX_VM0	**_CVX_LPT1**
_CVX_VM1	**_CVX_LPT2**
_CVX_VM2	**_CVX_LPT3**
_CVX_VM3	

silenceRange A **WORD** value that determines the range of sampled values that are to be encoded as silence. Note that all ranges center around 0x80, as this value represents actual silence. The value passed represents the following hexadecimal ranges:

_SILENCE_0	**_SILENCE_3**
_SILENCE_1	**_SILENCE_4**
_SILENCE_2	**_SILENCE_5**

See **CVXDIGI.H** in **Appendix A** for silence range descriptions.

bytes-Recorded	A pointer to a variable of type **LONG**, representing the number of bytes actually recorded by the function.

Remarks This function records industry standard 8 bit PCM speech with optional silence encoding. A typical recording will be about the same size as an 8 bit PCM file, but may be much shorter if silence encoding is used and there are long periods of silence during the recording.

A 16-byte header, described in **Appendix B**, precedes the resulting file.

If **_ioStopFlag** is not set to **_TRUE** during recording, the entire buffer will be filled with PCM data before recording is terminated. See **Appendix D** for information on usage of **_ioStopFlag**.

Return Value If successful, these functions return a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_PORT_INIT_FAILED
_ERROR_INVALID_BUFFER_SIZE

See the last page of this chapter for error code descriptions.

Example

```
// RECORD8S.C
// This program uses the non-DMA record8s function. Memory is
// allocated for the record. The recording is performed and the
// recorded buffer is written to a file. The memory allocated for
// the record buffer is freed.

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE      0x8000
#define _SOUND_FILE       "TEST1.V8"

VOID main( VOID )
{
   LONG      bytesRecorded;
   HANDLE    fileHandle;
   WORD      bytesWritten;
   WORD      bufferSegment;
   LPSTR     recordBuffer;


   // Allocate memory for recording buffer. Segment of the
   // allocated memory is returned in bufferSegment.
   if( ( _dos_allocmem( ( _BUFFER_SIZE / 16 + 1 ),
                        &bufferSegment ) ) )
      printf( "ERROR : Cannot Allocate Memory!!\n" );
```

```
    else
    {
       // record8s requires a far pointer (LPSTR).
       FP_SEG( recordBuffer ) = bufferSegment;
       FP_OFF( recordBuffer ) = 0x0000;

       // Prompt user to hit a key to begin recording.
       printf( "Hit any key to begin recording\n" );

       // Wait for keystroke.
       while( !_bios_keybrd( _KEYBRD_READY ) );

       // Clear keystroke(s) from keyboard buffer.
       while( _bios_keybrd( _KEYBRD_READY ) )
          getch();

       // Notify user that recording has begun.
       printf( "Recording ..." );

       // Record a file of length _BUFFER_SIZE.
       if( record8s( recordBuffer, ( LONG )_BUFFER_SIZE,
                     _CVX_RATE_DEFAULT, _CVX_VM0,
                     _SILENCE_3, &bytesRecorded ) )
       {
          // Notify user that error occured during record.
          printf( " aborted. Error during record8s().\n" );
       }
       else
       {
          // Notify user that recording has completed.
          printf( " complete.\n" );
       }

       // If bytesRecorded is not zero, no error was encountered.
       if( bytesRecorded )
       {
          // Open a new file to write sound data into.
          if( _dos_creat( _SOUND_FILE, _A_NORMAL, &fileHandle ) )
          {
             printf( "ERROR : Cannot create file.\n" );
             printf( "        Sound data lost.\n" );
          }
          else
          {
             // Write sound data to file.
             _dos_write( fileHandle, recordBuffer,
                         ( WORD )bytesRecorded, &bytesWritten );

             // Close file containing sound data.
             _dos_close( fileHandle );
          }
       }

       // Free memory used by buffer.
       if( _dos_freemem( bufferSegment ) )
          printf( "ERROR : Cannot Free Memory!!\n" );
    }
}
```

Description Outputs one byte to the device specified in **playByteInit**.

Prototype **VOID playByte(BYTE** *outByte* **);**

Parameters *outByte* An 8 bit value to be output to a Digital to Analog converter.

Remarks This function outputs one byte to the device specified by the **playByteInit** function. If *disableTimerFlag* was **_FALSE** when **playByteInit** was called this procedure polls timer 2 before outputting the byte. If *trebleFlag* was **_TRUE** when passed to **playByteInit**, this function outputs the differentiated result of the stream of bytes it receives.

Return Value None.

Example

```
// BYTE_IO.C
// This program initializes the non-DMA recording and playback
// system. A byte is input and immediately output through the
// Voice Master default port.

#include <bios.h>
#include <stdio.h>
#include "cvxdigi.h"

#define _TREBLE_FLAG      _FALSE

VOID main( VOID )
{
   BYTE aByte;
   BOOL disableTimerFlag = _FALSE;
   WORD physPort;

   // Initialize recording system.
   physPort = recordByteInit( _CVX_VM0, _CVX_RATE_DEFAULT,
                              disableTimerFlag );

   // Set timing flag so that no timing is performed during playback.
   disableTimerFlag = _TRUE;

   // Initialize playback system.
   physPort = playByteInit( _CVX_VM0, _CVX_RATE_DEFAULT,
                            disableTimerFlag, _TREBLE_FLAG );

   // If a physical port value was returned then record and
   // playback a byte at a time.
   if( physPort )
   {
      // Wait for key hit to exit.
      while( !_bios_keybrd( _KEYBRD_READY ) )
      {
         // Input one byte from physPort.
         aByte = recordByte();

         // Output one byte to physPort.
```

```
            playByte( aByte );
        }

        // Clear keystroke(s) from keyboard buffer.
        while( kbhit() )
            getch();

        // Uninitialize the playback system.
        playByteUninit();

        // Uninitialize the recording system.
        recordByteUninit();
    }
    else
    {
        printf( "ERROR : I/O port initialization failed.\n" );
    }
}
```

Description Initializes hardware for output to any Covox device using the **playByte** function.

Prototype **WORD playByteInit(WORD** *portOut*, **BYTE** *playbackRate*, **BOOL** *disableTimerFlag*, **BOOL** *trebleFlag* **);**

Parameters

portOut This **WORD** value designates which DAC port to use for playback. If the value of *portOut* is not one of the Covox logical port values shown below, it represents the physical port address of the DAC.

The Covox logical port numbers are:

_CVX_VM0	**_CVX_LPT1**
_CVX_VM1	**_CVX_LPT2**
_CVX_VM2	**_CVX_LPT3**
_CVX_VM3	**_CVX_ISP**

See **Appendix C** for port value descriptions.

playbackRate A **BYTE** value between 0 and 229, indicating the rate at which **playByte** will synchronize output. This value is ignored if *disableTimerFlag* is **_TRUE**.

If *playbackRate* is larger then 229, it is set to 229.

See **Appendix B** for rate value descriptions.

disableTimer-Flag A flag which, if **_TRUE**, indicates that the timer is to be used to synchronize output for the **playByte** function. If **_FALSE**, each byte will be output to the hardware device as soon as **playByte** is called.

trebleFlag A flag which, if **_TRUE**, indicates that the output is to be differentiated. If **_FALSE**, there is no differentiation.

Differentiation is used to achieve the effects of a high-pass filter, which enhances the upper frequencies of the digitized sound.

Remarks This function is used to initialize the hardware for output. Output is performed one byte at a time using the **playByte** function. The **playByteUninit** function must be called when all sound output is complete.

Return Value This function returns the physical port address. If the port is not found, (eg., LPT not supported or Sound Master II not installed), the function returns a 0.

Example

See **playByte.**

Description Resets the hardware and software after calling **playByteInit.**

Prototype **WORD playByteUninit(VOID);**

Parameters None.

Remarks This function is used to reset the hardware and software after an **playByteInit** has been called. This function is to be called after you are done using the **playByte** function.

Return Value The **WORD** value returned is either 0 if successful or 1 if unsuccessful.

Example

See **playByte.**

Description Inputs one byte from the device specified in **recordByteInit.**

Prototype **BYTE recordByte(VOID);**

Parameters None.

Remarks This function inputs one byte from the device specified using the **recordByteInit** function. If the timer was enabled when **recordByteInit** was called then this procedure polls timer 2 before reading the byte.

Return Value The byte input from the selected ADC.

Example

See **playByte.**

Description Initializes hardware for recording with the **recordByte** function.

Prototype **WORD recordByteInit(WORD** *portIn*, **WORD** *recordRate*, **BOOL** *disableTimerFlag* **);**

Parameters

portIn This **WORD** value designates which ADC port to use for recording. If the value of *portIn* is not one of the Covox logical port values shown below, it represents the physical port address of the ADC.

The Covox logical port numbers are:

_CVX_VM0	**_CVX_LPT1**
_CVX_VM1	**_CVX_LPT2**
_CVX_VM2	**_CVX_LPT3**
_CVX_VM3	

recordRate A **BYTE** value from 0 to 209, indicating the rate at which **recordByte** will synchronize input. This value is ignored if *disableTimerFlag* is **_TRUE.**

If *recordRate* is larger then 209, it is set to 209.

See **Appendix B** for rate value descriptions.

disableTimer-Flag A flag which, if **_TRUE**, indicates that the timer is to be used to synchronize input for the **recordByte** function. If **_FALSE**, each byte will be input from the hardware device as soon as **recordByte** is called.

Remarks This function is used to initialize the hardware for recording. The input will be performed a byte at a time using the function **recordByte**. The function **recordByteUninit** must be called when sound input is complete.

Return Value This function returns the physical port number; if the port was not found, (e.g. LPT port not installed), the function returns 0.

Example

See **playByte.**

recordByteUninit

Description Resets the hardware and software after calling **recordByteInit.**

Prototype **WORD recordByteUninit(VOID);**

Parameters None.

Remarks This function is used to reset the hardware and software after an **recordByteInit** has been called. This function is to be called after you are done using the **recordByte** function.

Return Value The **WORD** value returned is either 0 if successful or 1 if unsuccessful.

Example

See **playByte.**

_ERROR_NONE

No errors.

_ERROR_PORT_INIT_FAILED

Unable to find the specified port.

_ERROR_INVALID_BUFFER_SIZE

The size of the specified buffer was found to be less than the minimum acceptable value.

_ERROR_INVALID_FORMAT

The buffer format does not correspond to the function called.

_ERROR_PLAY8_BUSY

The **play8** function is already active.

CHAPTER 2

Buffer Pack And Unpack Functions

Structure

_CONVERT_DATA

High-Level Functions

packFirst
packNext
unpackFirst
unpackNext

The functions in this chapter are used to pack and unpack sound data. The structure **_CONVERT_DATA** needs to be appropriately initialized for use of the pack/unpack functions.

The functions **packFirst** and **packNext** can be used to compress 8 bit PCM (Pulse Coded Modulation) sound data to one of the available Covox ADPCM formats.

The functions **unpackFirst** and **unpackNext** can be used to unpack any of the Covox ADPCM formats to 8 bit PCM.

_CONVERT_DATA

Description A typedef structure used by the packing and unpacking routines. This structure must be properly initialized before calling any of the pack/unpack functions.

Structure

```
typedef struct _tagConvertDataStruct
{
        LPSTR     sourcePointer;
        LONG      sourceLength;
        LONG      sourceUsed;
        LPSTR     destinationPointer;
        LONG      destinationLength;
        LONG      destinationFilled;
        WORD      bufferFormat;
        BYTE      sampleRate;
        WORD      silenceRange;
        BOOL      trebleFlag;
        BOOL      noiseFlag;
} _CONVERT_DATA;
```

Structure Elements

sourcePointer A far pointer (**LPSTR**) to the location of the 8 bit PCM sound buffer when packing, or of the packed sound buffer when unpacking.

sourceLength A **LONG** value specifying the number of bytes in the buffer pointed to by *sourcePointer*.

sourceUsed Returned in this **LONG** variable is the number of bytes in the buffer pointed to by *sourcePointer* that have been converted.

When unpacking sound data, if the value of *sourceUsed* is less than the value of *sourceLength*, there are still bytes remaining to be unpacked in the buffer pointed to by *sourcePointer*.

destinationPointer A far pointer (**LPSTR**) to the location of the buffer to be packed or unpacked into.

destinationLength A **LONG** value indicating the number of bytes in the buffer pointed to by *destinationPointer*.

destinationFilled Returned in this **LONG** variable is the number of bytes in the buffer pointed to by *destinationPointer* that have been filled with converted sound data.

bufferFormat A **WORD** value that indicates the format of the buffer that is being packed to or unpacked from.

The possible values for *bufferFormat* are the following:

_FORMAT_V8
_FORMAT_V2S
_FORMAT_V3S
_FORMAT_V4S
_FORMAT_V8S

See **Appendix B** for buffer format descriptions.

sampleRate This **BYTE** value corresponds to the *recordRate* and *playbackRate* parameters of the **record** and **play** functions, respectively. It can be a value from 0 to 229.

If *sampleRate* is 0, the value in the source header is used to determine the sampling rate of the destination buffer. If the source header value is 0 or not present, the default value of 132 (9622 Hz) is used.

If *sampleRate* (or the rate specified in the header) is non-zero, that value is placed in the destination header; if it is larger than 229, a value of 229 (44,191 Hz) is used.

See **Appendix B** for rate value descriptions.

silenceRange A **WORD** value that determines the range of sampled values that are to be encoded as silence. Note that all ranges center around 0x80, as this value represents actual silence. The value passed represents the following hexadecimal ranges:

_SILENCE_0	**_SILENCE_3**
_SILENCE_1	**_SILENCE_4**
_SILENCE_2	**_SILENCE_5**

See **CVXDIGI.H** in **Appendix A** for silence range descriptions.

trebleFlag A flag which, if **_TRUE**, indicates that the output is to be differentiated upon playback. If **_FALSE**, there is no differentiation performed.

Differentiation is used to achieve the effects of a high-pass filter, which enhances the upper frequencies of the digitized sound.

noiseFlag A flag which, if **_TRUE**, indicates that computer generated noise is to be output during silence periods. If **_FALSE**, silence periods are played back as dead silence.

Description Convert an 8 bit PCM sound buffer to a packed ADPCM format.

Prototype **WORD packFirst((_CONVERT_DATA far *) *convert*);**

WORD packNext((_CONVERT_DATA far *) *convert*);

Parameters *convert* See the **_CONVERT_DATA** description for information on this structure.

Remarks The **packFirst** and **packNext** functions are used to convert an 8 bit PCM sound buffer to a packed format.

The **packFirst** function is called before **packNext** to create the destination header, set the necessary parameters and start the packing process. The **packNext** function is called after **packFirst** or another **packNext** to continue the packing process. Packing is finished when there is no more source data to be packed.

The **packFirst** and **packNext** functions are useful in a circular buffer scheme, like DMA recording and long file conversion.

When using **packFirst** the following **_CONVERT_DATA** parameters must be initialized:

sourcePointer
sourceLength
destinationPointer
destinationLength
sampleRate
bufferFormat
silenceRange

When using **packNext** the following **_CONVERT_DATA** parameters must be initialized:

sourcePointer
sourceLength
destinationPointer
destinationLength

Note: *destinationPointer* and *sourcePointer* are not changed by calls to **packFirst** or **packNext** and only need to be initialized once.

Return Value Both the **packFirst** and the **packNext** functions return 0 if packing is successful; a non-zero value if unsuccessful. If either function returns 0, *sourceUsed* contains the number of source bytes already packed by that function and *destinationFilled* contains the number of bytes placed in the destination buffer.

Example

```
// PACK8.C
// This program reads 8 bit PCM sound data from a file.
// The 8 bit sound data is converted to 3 bit with silence
// encoding using the packFirst and packNext routines.
// The 3 bit data is then written to a new file.
//
// The 2 buffers that are used in this program are allocated
// using _dos_allocmem. The size of both buffers is set to 32K.

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include <fcntl.h>
#include "cvxdigi.h"

#define  _BUFFER_SIZE     0x8000
#define  _PCM_FILE        "TEST1.V8"
#define  _3_BIT_FILE      "TEST1.V3S"

VOID main( VOID )
{
   LONG              recordedLength;
   HANDLE            pcmFileHandle;
   HANDLE            packedFileHandle;
   WORD              bytesWritten;
   WORD              bytesRead;

   _CONVERT_DATA     cnvrtData;
   WORD              pcmSegment;
   LPSTR             pcmBuffer;
   WORD              conversionSegment;
   LPSTR             conversionBuffer;


   // Allocate memory for buffers used to read 8 bit PCM
   // data from file.
   if( ( _dos_allocmem( ( WORD )( ( _BUFFER_SIZE / 16 ) + 1 ),
                        &pcmSegment ) ) )
   {
      printf( "ERROR : Cannot allocate memory for PCM buffer!\n" );
      exit( 0 );
   }
   else
   {
      // Pack requires a far pointer (LPSTR)
      FP_SEG( pcmBuffer ) = pcmSegment;
      FP_OFF( pcmBuffer ) = 0x0000;
   }

   // Allocate memory for buffer used for packing 8 bit PCM to
   // 3 bit ADPCM with silence encoding.
   //
   if( _dos_allocmem( ( WORD )( ( _BUFFER_SIZE / 16 ) + 1 ),
                      &conversionSegment ) )
   {
      printf( "ERROR : Cannot allocate memory for PCM buffer!\n" );
      exit( 0 );
   }
   else
```

```
{
   // pack requires a far pointer (LPSTR)
   FP_SEG( conversionBuffer ) = conversionSegment;
   FP_OFF( conversionBuffer ) = 0x0000;
}

// Open a file containg 8 bit PCM data.
if( _dos_open( _PCM_FILE, O_RDONLY, &pcmFileHandle ) )
   printf( "ERROR : %s not found.\n", _PCM_FILE );
else
   // Read header information from file.
   _dos_read( pcmFileHandle, pcmBuffer, ( WORD )_HEADER_LENGTH,
              &bytesRead );

// Point _CONVERT_DATA element to same location that will contain
// 8 bit PCM data read from _PCM_FILE
cnvrtData.sourcePointer = ( LPSTR )pcmBuffer;

// Point _CONVERT_DATA structure element to same location as
// conversionBuffer. This buffer will contain the data
// that has been packed.
cnvrtData.destinationPointer = ( LPVOID )conversionBuffer;

// Open a new file to write 3 bit sound data.
if( _dos_creat( _3_BIT_FILE, _A_NORMAL, &packedFileHandle ) )
   printf( "ERROR : Cannot create file %s.\n", _3_BIT_FILE );
else
{
   // Put header information into structure for call
   // to packFirst.
   cnvrtData.sampleRate      = ( PBYTE )( pcmBuffer +
                                  _HEADER_RATE_OFFSET );
   cnvrtData.bufferFormat    = _FORMAT_V3S;
   cnvrtData.silenceRange    = _TRUE;
   cnvrtData.sourceLength          = ( LONG )_HEADER_LENGTH;
   cnvrtData.destinationLength     = ( LONG )_HEADER_LENGTH;

   // Features not needed during pack.
   cnvrtData.trebleFlag          = 0;
   cnvrtData.noiseFlag           = 0;

   // Call the function to set up the sound file header.
   if( packFirst( ( _CONVERT_DATA far * )&cnvrtData ) )
      printf( "ERROR : Function packFirst failed.\n" );
   else
   {
      // Write the 16 bit header returned by packFirst()
      // to file.
      _dos_write( packedFileHandle, cnvrtData.destinationPointer,
                  ( WORD )_HEADER_LENGTH, &bytesWritten );

      // Increment pcmBuffer past the 16 byte header
      pcmBuffer = pcmBuffer + ( WORD )_HEADER_LENGTH;

      // Setup loop to process contents of pcmBuffer
      while( !eof( pcmFileHandle ) )
      {
         // Read 8 bit PCM data from file.
         _dos_read( pcmFileHandle, pcmBuffer, _BUFFER_SIZE,
                    &bytesRead );

         // Put information into _CONVERT_DATA structure for
```

```
            // packNext.
            cnvrtData.sourceLength      = bytesRead;
            cnvrtData.destinationLength = _BUFFER_SIZE;

            // Call the function to set up the next buffer.
            if( packNext( ( _CONVERT_DATA far * )&cnvrtData ) )
               printf( "ERROR : Data packing failure.\n" );
            else
               _dos_write( packedFileHandle,
                           cnvrtData.destinationPointer,
                           ( WORD )cnvrtData.destinationFilled,
                           &bytesWritten );
         }
      }

      // Close files.
      _dos_close( pcmFileHandle );
      _dos_close( packedFileHandle );
   }

   // Free memory used for 3 bit sound data.
   if( _dos_freemem( conversionSegment ) )
      printf( "ERROR : Cannot free memory for ADPCM buffer!\n" );

   // Free memory used by 8 bit PCM buffer.
   if( _dos_freemem( pcmSegment ) )
      printf( "ERROR : Cannot free memory for PCM buffer!\n" );
}
```

Description Convert a packed ADPCM sound buffer to an 8 bit PCM format.

Prototype **WORD unpackFirst((_CONVERT_DATA far *) *convert*)**

WORD unpackNext((_CONVERT_DATA far *) *convert*);

Parameters *convert* See the **_CONVERT_DATA** description for information on this structure.

Remarks The **unpackFirst** and **unpackNext** functions are used to convert a packed sound buffer to a standard 8 bit PCM format.

The **unpackFirst** function is called before **unpackNext** to create the Covox 16-byte header, set the parameters and start the unpacking process. The **unpackNext** function is called after **unpackFirst** or another **unpackNext** to continue the unpacking process. Continue using the function **unpackNext** on source data left over from previous unpack calls to unpack the rest of the data.

These functions are useful in a circular buffer scheme, like DMA playback and long file conversion.

When using **unpackFirst** the following **_CONVERT_DATA** parameters must be initialized:

sourcePointer	*sampleRate*
sourceLength	*bufferFormat*
destinationPointer	*trebleFlag*
destinationLength	*noiseFlag*

When using **unpackNext** the following **_CONVERT_DATA** parameters must be initialized:

sourcePointer
sourceLength
destinationPointer
destinationLength

Note: *destinationPointer* and *sourcePointer* are not changed by calls to **unpackFirst** or **unpackNext** and only need to be initialized once.

Return Value Both the **unpackFirst** and **unpackNext** functions return 0 if unpacking is successful; a non-zero value indicates that an error occurred in the values passed to the function. If the function returns 0, the number of source bytes already unpacked by the function is stored in *sourceUsed* and the number of bytes placed in the destination buffer is stored in *destinationFilled*.

Example

```
// UNPACK3S.C
// This program reads 3 bit sound data from a file.
// The 3 bit sound data is converted to 8 bit PCM encoding
// using the unpackFirst and unpackNext routines.
// The 8 bit PCM data is then written to a new file.
//
// The 2 buffers that are used in this program are allocated
// using _dos_allocmem. The size of both buffers is set to 32K.

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include <fcntl.h>
#include "cvxdigi.h"

#define  _BUFFER_SIZE     0x8000
#define  _PCM_FILE        "TEST1.V8"
#define  _3_BIT_FILE      "TEST1.V3S"

VOID main( VOID )
{
   LONG              recordedLength;
   HANDLE            v3sFileHandle;
   HANDLE            pcmFileHandle;
   WORD              bytesWritten;
   WORD              bytesRead;

   _CONVERT_DATA     cnvrtData;
   WORD              v3sSegment;
   LPSTR             v3sBuffer;
   WORD              conversionSegment;
   LPSTR             conversionBuffer;
   WORD              i;
   WORD              sourceSize          = 0;
   BOOL              exitFlag            = _FALSE;

   // Allocate memory for buffers used to read 3 bit
   // data from file.
   if( ( _dos_allocmem( ( WORD )( ( _BUFFER_SIZE / 16 ) + 1 ),
                        &v3sSegment ) ) )
   {
      printf( "ERROR : Cannot allocate memory for v3s buffer!\n" );
      exitFlag = _TRUE;
   }
   else
   {
      // unpack requires a far pointer (LPSTR)
      FP_SEG( v3sBuffer ) = v3sSegment;
      FP_OFF( v3sBuffer ) = 0x0000;
   }

   // Allocate memory for buffer used for unpacking 3 bit data with
   // silence encoding to 8 bit PCM.
   //
   if( _dos_allocmem( ( WORD )( ( _BUFFER_SIZE / 16 ) + 1 ),
                      &conversionSegment ) )
   {
      printf( "ERROR : Cannot allocate memory for ADPCM buffer!\n" );
      exitFlag = _TRUE;
```

```
}
else
{
   // unpack requires a far pointer (LPSTR)
   FP_SEG( conversionBuffer ) = conversionSegment;
   FP_OFF( conversionBuffer ) = 0x0000;
}

// Open file containg 3 bit data.
if( _dos_open( _3_BIT_FILE, O_RDONLY, &v3sFileHandle ) )
{
   printf( "ERROR : %s not found.\n", _3_BIT_FILE );
   exitFlag = _TRUE;
}
else
   // Read header information from file.
   _dos_read( v3sFileHandle, v3sBuffer, ( WORD )_HEADER_LENGTH,
              &bytesRead );

// Point _CONVERT_DATA structure element to same location as
// conversionBuffer. This buffer will contain the unpacked data
cnvrtData.destinationPointer = ( LPVOID )conversionBuffer;

// Point _CONVERT_DATA structure element to same location as
// buffer containing 3 bit data.
cnvrtData.sourcePointer = ( LPVOID )v3sBuffer;

// Open a new file to write 8 bit sound data.
if( _dos_creat( _PCM_FILE, _A_NORMAL, &pcmFileHandle ) )
{
   printf( "ERROR : Cannot create file %s.\n", _PCM_FILE );
   exitFlag = _TRUE;
}

if( !exitFlag )
{
   // Put header information into structure for call
   // to unpackFirst.
   cnvrtData.sampleRate      = ( PBYTE )( v3sBuffer +
                                          _HEADER_RATE_OFFSET );
   cnvrtData.bufferFormat    = _FORMAT_V3S;
   cnvrtData.trebleFlag      = _FALSE;
   cnvrtData.noiseFlag       = _TRUE;

   cnvrtData.destinationLength     = ( LONG )_HEADER_LENGTH;
   cnvrtData.sourceLength          = ( LONG )_HEADER_LENGTH;

   // Features not needed during unpack.
   cnvrtData.silenceRange          = 0;

   // Make call to unpack first 16 bytes of 3 bit buffer.
   if( unpackFirst( ( _CONVERT_DATA far * )&cnvrtData ) )
   {
      printf( "ERROR : Function unpackFirst failed.\n" );
      exitFlag = _TRUE;
   }
   else
   {
      // Write the 16 bit header returned by unpackFirst()
      // to file.
      _dos_write( pcmFileHandle, cnvrtData.destinationPointer,
                  ( WORD )_HEADER_LENGTH, &bytesWritten );
```

```
            // Increment v3sBuffer past the 16 byte header
            v3sBuffer = v3sBuffer + ( WORD )_HEADER_LENGTH;

            // Setup loop to process contents of v3sBuffer.
            do
            {
               // Read 3 bit data from file.
               _dos_read( v3sFileHandle, ( LPVOID )( ( ( LONG )cnvrt-
                          Data.sourcePointer ) + sourceSize ),
                          ( _BUFFER_SIZE - sourceSize ), &bytesRead );

               // Calculate the amount of 3 bit data we have left
               // to unpack in the current buffer.
               sourceSize = sourceSize + bytesRead;

               // Put information into _CONVERT_DATA structure for
               // unpackNext.
               cnvrtData.sourceLength          = ( LONG )sourceSize;
               cnvrtData.destinationLength     = ( LONG )_BUFFER_SIZE;

               // Call function to unpack a buffer of 3 bit data.
               if( unpackNext( ( _CONVERT_DATA far * )&cnvrtData ) )
               {
                  printf( "ERROR : Data unpacking failure.\n" );
                  exitFlag = _TRUE;
               }
               else
                  _dos_write( pcmFileHandle,
                              cnvrtData.destinationPointer,
                              ( WORD )cnvrtData.destinationFilled,
                              &bytesWritten );

               // Move left over source data to the beginning of the
               // conversion buffer.
               sourceSize = sourceSize - ( WORD )cnvrtData.sourceUsed;

               for( i = 0; i < sourceSize; i++ )
                  *( cnvrtData.sourcePointer + i ) =
                     *( cnvrtData.sourcePointer +
                        cnvrtData.sourceUsed + i );

               // Check to see if we are done unpacking.
               if( ( sourceSize < 16 ) && eof( v3sFileHandle ) )
                  exitFlag = _TRUE;

            } while( !exitFlag );
         }

         // Close files.
         _dos_close( v3sFileHandle );
         _dos_close( pcmFileHandle );
      }

      // Free memory used for 3 bit sound data.
      if( _dos_freemem( conversionSegment ) )
         printf( "ERROR : Cannot free memory!\n" );

      // Free memory used by 8 bit PCM buffer.
      if( _dos_freemem( v3sSegment ) )
         printf( "ERROR : Cannot free memory used for PCM buffer!\n" );
   }
```

CHAPTER 3

DMA Record And Play Functions

High Level Routines

dmaFlushQueue	dmaPlay
dmaInit	dmaRecord
dmaNumberInQueue	dmaUninit

Low Level Routines

dmaBytesRemaining	dmaPause
dmaGetChannel	dmaPortDetect
dmaGetPort	dmaSetRate
dmaGetIrqNumber	dmaUnpause

The library functions in this chapter utilize a FIFO (First In, First Out) queue for recording and playing buffers of sound data under DMA. This queue can hold a maximum of 4 requests.

The function **dmaInit** installs the interrupt handler to manage the interrupts fired when DMA reaches a page boundary or the end of a buffer. **dmaInit** also initializes the FIFO queue data structure and all related pointers. The interrupt handler and the queue manipulation routines are referred to as the DMA sub-system.

A request is placed into the queue by the functions **dmaPlay** and **dmaRecord**.

The function **dmaNumberInQueue** is used to determine the number of requests in the DMA queue. The queue can hold a maximum of four requests. The function **dmaFlushQueue** removes all pending DMA I/O requests from the queue.

The function **dmaBytesRemaining** can be utilized when DMA is stopped before a buffer is completely played or recorded.

The flag **_dmaInProgressFlag** is **_TRUE** when DMA is performing I/O.

The functions **dmaPause** and **dmaUnpause** disable and enable the DMA controller on the motherboard, effectively starting and stopping DMA without completely removing the DMA sub-system.

A call to **dmaUninit** will completely remove the DMA sub-system.

Library Variables

The following variables are available for use by those applications using the functions in this chapter.

BYTE _dmaInProgressFlag: This flag, if **_TRUE**, indicates that a DMA operation is currently running. **_dmaInProgressFlag** can be monitored to tell when DMA recording or playback is finished.

BYTE _dmaDevice: This variable, if **_TRUE**, indicates that DMA has been initialized to use the Voice Master or Sound Master II. If the value of **_dmaDevice** is **_FALSE**, DMA has not been initialized.

Description Clears the Play and Record Queue and stops the present DMA operation.

Prototype **WORD dmaFlushQueue(VOID);**

Parameters None.

Remarks The **dmaRecord** and **dmaPlay** routines use a FIFO queue to store pending record and play requests. This function clears the queue and stops the present DMA operation.

Return Value If successful, **dmaFlushQueue** returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_DMA_NOT_INITIALIZED

See the last page of the chapter for error code descriptions.

Example

See **dmaRecord** and **dmaPlay.**

Description Initializes Covox hardware and software for DMA.

Prototype **WORD dmaInit(WORD** *portJumper*, **WORD** *dmaChannel*,
WORD *irqNumber*, **WORD *** *portAddress*);

Parameters *portJumper* Port address jumper setting. Search for DMA-capable devices (i.e VM or SM II) set to this port address.

_AUTODETECT	**_CVX_VM2**
_CVX_VM0	**_CVX_VM3**
_CVX_VM1	

See **Appendix C** for port value descriptions.

dmaChannel DMA channels searched to find active channel.

_AUTODETECT
_DMA_CHANNEL_1
_DMA_CHANNEL_3

See **Appendix C** for DMA channel value descriptions.

irqNumber IRQ numbers searched to find active IRQ line.

_AUTODETECT	**_IRQ_4**
_IRQ_2	**_IRQ_5**
_IRQ_3	**_IRQ_7**

See **Appendix C** for IRQ value descriptions.

portAddress A pointer to a **WORD**, representing the physical port address (0x220, 0x240, 0x280 or 0x2C0) detected by **dmaInit**.

Remarks This function finds which, if any, Covox DMA-capable hardware is installed. It detects the jumper settings for the port address, DMA channel and IRQ number, then initializes the hardware and software for DMA I/O.

Return Value If successful, **dmaInit** returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_DMA_ALREADY_INITIALIZED
_ERROR_DMA_DETECT_FAILED

See the end of the chapter for error code descriptions.

Example

See **dmaRecord** and **dmaPlay**.

dmaNumberInQueue

Description Returns the number of DMA I/O requests in the Play and Record Queue.

Prototype **WORD dmaNumberInQueue(VOID);**

Parameters None.

Remarks The **dmaRecord** and **dmaPlay** routines use a FIFO queue to store DMA I/O requests. A request is not removed from the queue until it is fully processed.

Return Value The number of DMA I/O requests in the Play and Record Queue.

Example

See **dmaRecord** and **dmaPlay**.

Description Plays 8 bit PCM data in the background using DMA.

Prototype **WORD dmaPlay(LPSTR** *bufferStart*, **LONG** *bufferSize*,
BYTE *playbackRate*, **WORD** *repeatCount*);

Parameters

bufferStart A far pointer (**LPSTR**) to a buffer within the first 1 MB of system memory. This buffer will temporarily hold the sound data while recording under DMA.

bufferSize Length (in bytes) of the buffer pointed to by *bufferStart*.

playbackRate A **BYTE** value from 0 to 229, indicating the sampling rate at which to play back the sound buffer.

If *playbackRate* is 0, the default value of 132 (9622 Hz) is used. If *playbackRate* is larger then 229, it is set to 229 (44,191 Hz).

See **Appendix B** for rate value descriptions.

repeatCount Number of times to play back the data buffer. If *repeatCount* is set to -1 (0xFFFF), the buffer is repeated indefinitely.

Remarks This function places a DMA playback request in the DMA queue and, if no other requests are in the queue, starts playing 8 bit PCM data in the background using DMA. If other requests are in the queue, **dmaPlay** first waits until they are complete, then sets the **_dmaInProgressFlag** flag (a **BYTE**) to 0 and activates interrupt 0x15 with register AX = 0x91F0 (A device post command with device set to 0xF0).

Return Value If successful, **dmaPlay** returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_DMA_NOT_INITIALIZED
_ERROR_QUEUE_FULL

See the last page of the chapter for a description of error codes.

Example

```
// DMAPLAY.C
// This program reads 8 bit PCM sound data from a file.
// The 8 bit sound data is then played back using DMA.
//
// The buffers that are used in this program are allocated
// using _dos_allocmem. Each buffer is set to 16K.

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include <fcntl.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE  0x4000       // Size of each DMA buffer.
#define _BUFFER_COUNT 4            // Number of DMA buffers.
#define _PCM_FILE     "TEST1.V8"   // File containing sound data.

extern _dmaInProgressFlag;
extern _dmaDevice;

VOID main( VOID )
{
   HANDLE  pcmFileHandle;
   WORD    pcmSegment[ _BUFFER_COUNT ];
   LPSTR   pcmBuffer[ _BUFFER_COUNT ];
   WORD    bytesRead;
   WORD    portAddress;
   BYTE    dmaRate;
   WORD    phase       = 0;
   WORD    repeatCount = 1;
   WORD    initError;
   WORD    i;

   // Allocate memory for buffers used to read 8 bit PCM
   // data from file.
   for( i = 0; i < _BUFFER_COUNT; i++ )
   {
      if( ( _dos_allocmem( ( WORD )( ( _BUFFER_SIZE / 16 ) + 1 ),
                           &pcmSegment[ i ] ) ) )
      {
         printf( "ERROR : Cannot allocate memory!\n" );
         exit( 0 );
      }
      else
      {
         // dmaPlay() requires a far pointer (LPSTR)
         FP_SEG( pcmBuffer[ i ] ) = pcmSegment[ i ];
         FP_OFF( pcmBuffer[ i ] ) = 0x0000;
      }
   }

   // Initialize DMA. Setting each parameter to _AUTODETECT
   // causes dmaInit to perform a search for the Port,
   // DMA channel, and IRQ setting respectively.
   initError = dmaInit( _AUTODETECT, _AUTODETECT, _AUTODETECT,
                        &portAddress );

   // If the variable _dmaDevice equals 0 then the DMA
   // sub-system was not initialized correctly.
```

```
    if( _dmaDevice == 0 )
       printf( "ERROR = %d : dmaInit failed\n", initError );
    else
    {
       // Open a file containing 8 bit PCM data.
       if( _dos_open( _PCM_FILE, O_RDONLY, &pcmFileHandle ) )
          printf( "ERROR : %s not found.\n", _PCM_FILE );
       else
       {
          // Get header information, if it exists, from file.
          _dos_read( pcmFileHandle, ( LPVOID )pcmBuffer[ phase ],
                     ( WORD )_HEADER_LENGTH, &bytesRead );

          // Get rate from header.
          dmaRate = pcmBuffer[ phase ][ _HEADER_RATE_OFFSET ];

          //  Loop here until we have queued the entire file.
          do
          {
             // Fill buffer with sound data.
             _dos_read( pcmFileHandle, ( LPVOID )pcmBuffer[ phase ],
                        ( WORD )_BUFFER_SIZE, &bytesRead );

             // Insert buffer into DMA queue.
             dmaPlay( ( LPSTR )pcmBuffer[ phase ], ( LONG )bytesRead,
                      dmaRate, repeatCount );

             // Toggle phase so next buffer is accessed.
             if( phase == ( _BUFFER_COUNT - 1 ) )
                phase = 0;
             else
                phase++;

             // Loop until a spot in the queue opens up.
             while( dmaNumberInQueue() == _BUFFER_COUNT );

          } while( !eof( pcmFileHandle ) );

          // Loop until DMA has completed.
          while( _dmaInProgressFlag );
       }
    }

    // Clear all requests from the Play and Record Queue.
    if( dmaFlushQueue() )
       printf( "DMA uninit failure : could not flush queue.\n" );

    // Uninitialize the DMA system.
    if( dmaUninit() )
       printf( "DMA uninit failure.\n" );

    // Close the sound file.
    _dos_close( pcmFileHandle );

    // Free memory used by 8 bit PCM buffer.
    for( i = ( _BUFFER_COUNT - 1 ); i != -1; i-- )
    {
       if( _dos_freemem( pcmSegment[ i ] ) )
          printf( "ERROR : Cannot free memory!\n" );
    }
}
```

Description Records 8 bit PCM data in the background using DMA.

Prototype **WORD dmaRecord(LPSTR** *bufferStart*, **LONG** *bufferSize*,
BYTE *recordRate*);

Parameters

bufferStart A far pointer (**LPSTR**) to a buffer within the first 1 MB of system memory. This buffer will temporarily hold the sound data while recording under DMA.

bufferSize Length (in bytes) of the buffer pointed to by *bufferStart*.

recordRate A **BYTE** value from 0 to 209, indicating the sampling rate at which to record the sound buffer.

If *recordRate* is 0, the default value of 132 (9622 Hz) is used. If *recordRate* is larger then 209, it is set to 209 (25,386 Hz).

See **Appendix B** for rate value descriptions.

Remarks This function places a DMA record request in the DMA queue and, if no other requests are in the queue, starts recording 8 bit PCM data in the background using DMA. If other requests are in the queue, **dmaRecord** first waits until they are complete, then sets the **_dmaInProgressFlag** flag (a **BYTE**) to 0 and activates interrupt 0x15 with register AX = 0x91F0 (A device post command with device set to 0xF0).

Return Value If successful, **dmaRecord** returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_DMA_NOT_INITIALIZED
_ERROR_QUEUE_FULL

See the last page of the chapter for error code descriptions.

Example

```
// DMAREC.C
// This program records using DMA. Four buffers are queued for
// DMA recording. After each buffer has completed being recorded
// it is written to a file.
//
// The buffers that are used in this program are allocated
// using _dos_allocmem. Each buffer is set to 32K.

#include <stdio.h>
#include <bios.h>
#include <dos.h>
```

```
#include <errno.h>
#include <fcntl.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE  0x8000       // Size of each DMA buffer.
#define _BUFFER_COUNT 4            // Number of DMA buffers.
#define _PCM_FILE     "TEST1.V8"   // File for recorded data.

BYTE cvxHeader[] = { 0xFF,0x55,0xFF,0xAA,0xFF,0x55,0xFF,0xAA,
                     0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 };

extern _dmaInProgressFlag;
extern _dmaDevice;


VOID main( VOID )
{
   HANDLE  pcmFileHandle;
   WORD    pcmSegment[ _BUFFER_COUNT ];
   LPSTR   pcmBuffer[ _BUFFER_COUNT ];
   WORD    portAddress;
   WORD    bytesWritten;
   WORD    phase;
   WORD    initError;
   WORD    i;
   // Allocate memory for buffers used to record 8 bit PCM.
   for( i = 0; i < _BUFFER_COUNT; i++ )
   {
      if( ( _dos_allocmem( ( WORD )( ( _BUFFER_SIZE / 16 ) + 1 ),
                           &pcmSegment[ i ] ) ) )
      {
         printf( "ERROR : Cannot allocate memory!\n" );
         exit( 0 );
      }
      else
      {
         // dmaRecord() requires a far pointer (LPSTR)
         FP_SEG( pcmBuffer[ i ] ) = pcmSegment[ i ];
         FP_OFF( pcmBuffer[ i ] ) = 0x0000;
      }
   }

   // Initialize DMA. Setting each parameter to _AUTODETECT
   // causes dmaInit to perform a search for the Port,
   // DMA channel, and IRQ setting respectively.
   initError = dmaInit( _AUTODETECT, _AUTODETECT, _AUTODETECT,
                        &portAddress );

   // If the variable _dmaDevice equals 0 then the DMA
   // sub-system was not initialized correctly.
   if( _dmaDevice == 0 )
      printf( "ERROR = %d : dmaInit failed\n", initError );
   else
   {
      // Create file for recorded 8 bit PCM data.
      if( _dos_creat( _PCM_FILE, _A_NORMAL, &pcmFileHandle ) )
         printf( "ERROR : Cannot create %s.\n", _PCM_FILE );
      else
      {
         // Put rate and format into header.
         cvxHeader[ _HEADER_RATE_OFFSET ]   = _CVX_RATE_DEFAULT;
         cvxHeader[ _HEADER_FORMAT_OFFSET ] = _FORMAT_V8;
```

```
            // Write header information to file.
            _dos_write( pcmFileHandle, cvxHeader,
                        ( WORD )_HEADER_LENGTH, &bytesWritten );

            // Queue all buffers for recording.
            for( i = ( _BUFFER_COUNT - 1 ); i != -1; i-- )
               dmaRecord( pcmBuffer[ i ], ( LONG )_BUFFER_SIZE,
                          ( WORD )_CVX_RATE_DEFAULT );

            // The variable 'phase' will indicate when a buffer has
            // been filled with recorded sound data. One is subtracted
            // from phase so that when a buffer is finished, phase
            // will equal the return value of dmaNumberInQueue.
            phase = _BUFFER_COUNT - 1;

            // Loop here until there are no more buffers left
            // in queue.
            do
            {
               // Loop until a spot in the queue opens up.
               while( dmaNumberInQueue() > phase );

               // Write recorded buffer to file.
               _dos_write( pcmFileHandle, pcmBuffer[ phase ],
                           _BUFFER_SIZE, &bytesWritten );

               // Decrement phase so that we can tell when the
               // next buffer has been recorded.
               phase--;

            } while( phase > 0 );

            // At this point the last buffer is being recorded.
            while( _dmaInProgressFlag );

            // Write last recorded buffer to file.
            _dos_write( pcmFileHandle, pcmBuffer[ phase ],
                        _BUFFER_SIZE, &bytesWritten );
         }
      }

      // Clear all requests from the Play and Record Queue.
      if( dmaFlushQueue() )
         printf( "DMA uninit failure : could not flush queue.\n" );

      // Uninitialize the DMA system.
      if( dmaUninit() )
         printf( "DMA uninit failure.\n" );

      // Close the sound file.
      _dos_close( pcmFileHandle );

      // Free memory used by 8 bit PCM buffer.
      for( i = ( _BUFFER_COUNT - 1 ); i != -1; i-- )
      {
         if( _dos_freemem( pcmSegment[ i ] ) )
            printf( "ERROR : Cannot free memory!\n" );
      }
   }
```

Description Uninitializes the DMA sub-system.

Prototype **WORD dmaUninit(VOID);**

Parameters None.

Remarks This function uninitializes the DMA capabilities of a Covox sound board that has been initialized with **dmaInit**. **dmaUninit** must be called before exiting in order to disable the hardware and restore the system back to its original state.

Return Value If successful, **dmaUninit** returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_DMA_NOT_INITIALIZED

See the last page of the chapter for error code descriptions.

Example

See **dmaRecord** and **dmaPlay**.

dmaBytesRemaining

Description Returns the number of bytes that have not been processed by the current DMA request.

Prototype **WORD dmaBytesRemaining(VOID);**

Parameters None.

Remarks This function reads the DMA controller and returns the number of bytes that have not been processed by the current DMA request.

Return Value The **WORD** value of the Count Register on the DMA controller.

Example

```
// DMACNT.C
// This program reads one buffer of 8 bit PCM sound data
// from a file. This buffer is played back under DMA.
// The results of the function dmaBytesRemaining() are printed
// to the screen immediately after dmaPlay() is called and
// again after DMA has completed.
//
// The buffers that are used in this program are allocated
// using _dos_allocmem. Each buffer is set to 32K.

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include <fcntl.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE  0x8000       // Size of DMA buffer.
#define _PCM_FILE     "TEST1.V8"   // File containing sound data.

extern _dmaInProgressFlag;
extern _dmaDevice;

VOID main( VOID )
{
   HANDLE  pcmFileHandle;
   WORD    pcmSegment;
   LPSTR   pcmBuffer;
   WORD    bytesRead;
   WORD    portAddress;
   BYTE    dmaRate;
   WORD    dmaTempCount;
   WORD    repeatCount = 1;
   WORD    initError;

   // Allocate memory for buffer used to read 8 bit PCM
   // data from file.
   if( ( _dos_allocmem( ( WORD )( ( _BUFFER_SIZE / 16 ) + 1 ),
                        &pcmSegment ) ) )
   {
      printf( "ERROR : Cannot allocate memory!\n" );
      exit( 0 );
```

```
    }
    else
    {
       // dmaPlay() requires a far pointer (LPSTR)
       FP_SEG( pcmBuffer ) = pcmSegment;
       FP_OFF( pcmBuffer ) = 0x0000;
    }

    // Initialize DMA. Setting each parameter to _AUTODETECT
    // causes dmaInit to perform a search for the Port,
    // DMA channel, and IRQ setting respectively.
    initError = dmaInit( _AUTODETECT, _AUTODETECT, _AUTODETECT,
                         &portAddress );

    // If the variable _dmaDevice equals 0 then the DMA
    // sub-system was not initialized correctly.
    if( _dmaDevice == 0 )
       printf( "ERROR = %d : dmaInit failed\n", initError );
    else
    {
       // Open a file containing 8 bit PCM data.
       if( _dos_open( _PCM_FILE, O_RDONLY, &pcmFileHandle ) )
          printf( "ERROR : %s not found.\n", _PCM_FILE );
       else
       {
          // Get header information, if it exists, from file.
          _dos_read( pcmFileHandle, ( LPVOID )pcmBuffer,
                     _HEADER_LENGTH, &bytesRead );

          // Get rate from header.
          dmaRate = pcmBuffer[ _HEADER_RATE_OFFSET ];

          // Fill buffer with sound data.
          _dos_read( pcmFileHandle, ( LPVOID )pcmBuffer,
                     ( WORD )_BUFFER_SIZE, &bytesRead );

          // Insert buffer into DMA queue.
          dmaPlay( ( LPSTR )pcmBuffer, ( LONG )bytesRead,
                   dmaRate, repeatCount );

          // Loop until DMA has completed.
          while( _dmaInProgressFlag );

          printf( "dmaBytesRemaining() = %d\n", dmaBytesRemaining() );
       }
    }

    // Clear all requests from the Play and Record Queue.
    if( dmaFlushQueue() )
       printf( "DMA uninit failure : could not flush queue.\n" );

    // Uninitialize the DMA sub-system.
    if( dmaUninit() )
       printf( "DMA uninit failure.\n" );

    // Close the sound file.
    _dos_close( pcmFileHandle );

    // Free memory used by 8 bit PCM buffer.
    if( _dos_freemem( pcmSegment ) )
       printf( "ERROR : Cannot free memory!\n" );
}
```

dmaGetChannel

Description Retrieves information about the DMA channel the device is using.

Prototype **WORD dmaGetChannel(WORD *** *dmaChannel* **);**

Parameters *dmaChannel* A pointer to a **WORD** variable, indicating the DMA Channel that the Covox board is configured to. Possible values:

_DMA_CHANNEL_1
_DMA_CHANNEL_3

See **Appendix C** for DMA Channel value descriptions.

Remarks After DMA has been initialized with **dmaInit**, information about the DMA channel the device is using can be retrieved with this function.

Return Value If successful, **dmaGetChannel** returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_DMA_NOT_INITIALIZED

See the last page of the chapter for error code descriptions.

Example

```
// DMAGETCH.C
// This program installs the DMA sub-system and calls
// the function dmaGetChannel to determine the channel
// setting of the Voice Master or Sound Master II.

#include <stdio.h>
#include <dos.h>
#include "cvxdigi.h"

extern _dmaDevice;

VOID main( VOID )
{
   WORD dmaChannel;
   WORD initError;
   WORD portAddress;

   // Initialize DMA. Setting each parameter to _AUTODETECT
   // causes dmaInit to perform a search for the Port,
   // DMA channel, and IRQ setting respectively.
   initError = dmaInit( _AUTODETECT, _AUTODETECT, _AUTODETECT,
                        &portAddress );

   // If the variable _dmaDevice equals 0 then the DMA
   // sub-system was not initialized correctly.
   if( _dmaDevice == 0 )
      printf( "ERROR = %d : dmaInit failed\n", initError );
   else
```

```
    {
        // Call function to to get the DMA channel setting
        // of the Voice Master or Sound Master II.
        dmaGetChannel( &dmaChannel );

        // Display DMA channel to the screen.
        printf( "DMA using channel %x\n", dmaChannel );

        // Uninitialize the DMA sub-system.
        if( dmaUninit() )
            printf( "DMA uninit failure.\n" );
    }
}
```

Description Retrieves the device port address after initialization.

Prototype **WORD dmaGetPort(WORD * *dmaPort*);**

Parameters *dmaPort* A pointer to a **WORD** variable, indicating the port address that the Covox board is configured to. Possible values:

_CVX_VM0	**_CVX_VM2**
_CVX_VM1	**_CVX_VM3**

See **Appendix C** for port value descriptions.

Remarks After DMA has been initialized with **dmaInit**, information about device port address can be retrieved with this function.

Return Value If successful, **dmaGetPort** returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_DMA_NOT_INITIALIZED

See the last page of the chapter for error code descriptions.

Example

```
// DMAGETP.C
// This program installs the DMA sub-system and calls
// the function dmaGetPort to determine the jumper
// setting of the Voice Master or Sound Master II.

#include <stdio.h>
#include <dos.h>
#include "cvxdigi.h"

extern _dmaDevice;

VOID main( VOID )
{
   WORD dmaPort;
   WORD initError;
   WORD portAddress;

   // Initialize DMA. Setting each parameter to _AUTODETECT
   // causes dmaInit to perform a search for the Port,
   // DMA channel, and IRQ setting respectively.
   initError = dmaInit( _AUTODETECT, _AUTODETECT, _AUTODETECT,
                        &portAddress );

   // If the variable _dmaDevice equals 0 then the DMA
   // sub-system was not initialized correctly.
   if( _dmaDevice == 0 )
      printf( "ERROR = %d : dmaInit failed\n", initError );
   else
```

```
    {
        // Call function to to get the device jumper setting
        // of the Voice Master or Sound Master II.
        dmaGetPort( &dmaPort );

        // Display DMA port to the screen.
        printf( "DMA using port %x\n", dmaPort );

        // Uninitialize the DMA sub-system.
        if( dmaUninit() )
            printf( "DMA uninit failure.\n" );
    }
}
```

dmaGetIRQNumber

Description Retrieves information about the IRQ channel the device is using.

Prototype **WORD dmaGetIRQNumber(WORD *** ***dmaIrqNumber*** **);**

Parameters

dmaIrq-Number	A pointer to a **WORD** variable, indicating the IRQ channel that the Covox board is configured to. Possible values:

_IRQ_2 **_IRQ_5**
_IRQ_3 **_IRQ_7**
_IRQ_4

See **Appendix C** for IRQ value descriptions.

Remarks After DMA has been initialized with **dmaInit**, information about the IRQ channel the device is using can be retrieved with this function.

Return Value If successful, **dmaGetIrqNumber** returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_DMA_NOT_INITIALIZED

See the last page of the chapter for error code descriptiosns.

Example

```
// DMAGETI.C
// This program installs the DMA sub-system and calls
// the function dmaGetIRQNumber to determine the IRQ
// setting of the Voice Master or Sound Master II.

#include <stdio.h>
#include <dos.h>
#include "cvxdigi.h"

extern _dmaDevice;

VOID main( VOID )
{
   WORD dmaIRQNumber;
   WORD initError;
   WORD portAddress;

   // Initialize DMA. Setting each parameter to _AUTODETECT
   // causes dmaInit to perform a search for the Port,
   // DMA channel, and IRQ setting respectively.
   initError = dmaInit( _AUTODETECT, _AUTODETECT, _AUTODETECT,
                        &portAddress );

   // If the variable _dmaDevice equals 0 then the DMA
   // sub-system was not initialized correctly.
   if( _dmaDevice == 0 )
      printf( "ERROR = %d : dmaInit failed\n", initError );
```

```
    else
    {
       // Call function to to get the DMA IRQ setting
       // of the Voice Master or Sound Master II.
       dmaGetIRQNumber( &dmaIRQNumber );

       // Display DMA IRQ number to the screen.
       printf( "DMA using IRQ %x\n", dmaIRQNumber );

       // Uninitialize the DMA sub-system.
       if( dmaUninit() )
          printf( "DMA uninit failure.\n" );
    }
}
```

dmaPause

Description Pauses DMA playback or recording.

Prototype **BOOL dmaPause(VOID);**

Parameters None.

Remarks This function disables the DMA sub-system by disabling the DMA controller on the motherboard.

Return Value If **_TRUE**, DMA was successfully paused.

If **_FALSE**, DMA cannot be paused for one of two possible reasons. Either the value of **_dmaInProgressFlag** is **_FALSE**, or DMA has not been initialized.

Example

```
// DMAPAUSE.C
// This program reads 8 bit PCM sound data from a file.
// The 8 bit sound data is then played back using DMA.
// If a key is struck on the keyboard during the playback,
// then DMA is paused. Another keystoke will restart DMA.
//
// The buffers that are used in this program are allocated
// using _dos_allocmem. Each buffer is set to 16K.

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include <fcntl.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE  0x4000       // Size of each DMA buffer.
#define _BUFFER_COUNT 4            // Number of DMA buffers.
#define _PCM_FILE     "TEST1.V8"   // File containing sound data.

extern _dmaInProgressFlag;
extern _dmaDevice;

VOID main( VOID )
{
   HANDLE  pcmFileHandle;
   WORD    pcmSegment[ _BUFFER_COUNT ];
   LPSTR   pcmBuffer[ _BUFFER_COUNT ];
   WORD    bytesRead;
   WORD    portAddress;
   BYTE    dmaRate;
   WORD    repeatCount = 1;
   WORD    initError;
   WORD    i;

   // Allocate memory for buffers used to read 8 bit PCM
   // data from file.
   for( i = 0; i < _BUFFER_COUNT; i++ )
   {
```

```
if( ( _dos_allocmem( ( WORD )( ( _BUFFER_SIZE / 16 ) + 1 ),
                                &pcmSegment[ i ] ) ) )
      {
         printf( "ERROR : Cannot allocate memory!\n" );
         exit( 0 );
      }
      else
      {
         // dmaPlay() requires a far pointer (LPSTR)
         FP_SEG( pcmBuffer[ i ] ) = pcmSegment[ i ];
         FP_OFF( pcmBuffer[ i ] ) = 0x0000;
      }
   }

   // Initialize DMA. Setting each parameter to _AUTODETECT
   // causes dmaInit to perform a search for the Port,
   // DMA channel, and IRQ setting respectively.
   initError = dmaInit( _AUTODETECT, _AUTODETECT, _AUTODETECT,
                        &portAddress );

   // If the variable _dmaDevice equals 0 then the DMA
   // sub-system was not initialized correctly.
   if( _dmaDevice == 0 )
      printf( "ERROR = %d : dmaInit failed\n", initError );
   else
   {
      // Open a file containing 8 bit PCM data.
      if( _dos_open( _PCM_FILE, O_RDONLY, &pcmFileHandle ) )
         printf( "ERROR : %s not found.\n", _PCM_FILE );
      else
      {
         // Get header information, if it exists, from file.
         _dos_read( pcmFileHandle, ( LPVOID )pcmBuffer[ 0 ],
                    ( WORD )_HEADER_LENGTH, &bytesRead );

         // Get rate from header.
         dmaRate = pcmBuffer[ 0 ][ _HEADER_RATE_OFFSET ];

         //  Loop here until we have queued all 4 buffers.
         for( i = 0; i < _BUFFER_COUNT; i++ )
         {
            // Fill buffer with sound data.
            _dos_read( pcmFileHandle, ( LPVOID )pcmBuffer[ i ],
                       ( WORD )_BUFFER_SIZE, &bytesRead );

            // Insert buffer into DMA queue.
            dmaPlay( ( LPSTR )pcmBuffer[ i ], ( LONG )bytesRead,
                     dmaRate, repeatCount );
         }

         printf( "Press any key to pause DMA.\n" );

         // Loop until DMA has completed.
         while( _dmaInProgressFlag )
         {
            // If a key is struck, pause DMA playback.
            if( _bios_keybrd( _KEYBRD_READY ) )
            {
               dmaPause();

               // Clear all keystrokes.
               while( _bios_keybrd( _KEYBRD_READY ) )
```

```
                    getch();

                printf( "\nDMA has been paused.\n" );
                printf( "Press any key to restart.\n" );

                // Loop until a key is struck.
                while( !_bios_keybrd( _KEYBRD_READY ) )

                // Restart DMA playback.
                dmaUnpause();

                // Clear all key stokes.
                while( _bios_keybrd( _KEYBRD_READY ) )
                    getch();

                printf( "\nPress any key to pause DMA.\n" );
            }
        }
    }
}

// Clear all requests from the Play and Record Queue.
if( dmaFlushQueue() )
    printf( "DMA uninit failure : could not flush queue.\n" );

// Uninitialize the DMA system.
if( dmaUninit() )
    printf( "DMA uninit failure.\n" );

// Close the sound file.
_dos_close( pcmFileHandle );

// Free memory used by 8 bit PCM buffer.
for( i = ( _BUFFER_COUNT - 1 ); i != -1 ; i-- )
{
    if( _dos_freemem( pcmSegment[ i ] ) )
        printf( "ERROR : Cannot free memory!\n" );
}
}
```

Description Detects which, if any, Covox DMA-capable board is installed.

Prototype **WORD dmaPortDetect(WORD** *portJumper* **);**

Parameters *portJumper* The base port addresses to be tested for occupancy.

_AUTODETECT	**_CVX_VM2**
_CVX_VM0	**_CVX_VM3**
_CVX_VM1	

See **Appendix C** for port value descriptions.

Remarks This function performs a hardware detection to find out if the Covox Voice Master or Sound Master II board is present. This will not detect the earlier Voice Master cards without DMA capabilities.

Return Value Voice Master or Sound Master II base port address (0x220, 0x240, 0x280, or 0x2C0). **dmaPortDetect** returns 0 if one of these boards could not be found at the given address.

Example

```
// DMAPORT.C
// This program determines which port the Voice Master
// or Sound Master II card can use for DMA control.

#include <stdio.h>
#include <dos.h>
#include "cvxdigi.h"

VOID main( VOID )
{
   WORD dmaPort;

   // Call function to find port value.
   dmaPort = dmaPortDetect( _AUTODETECT );

   printf( "port = %x\n", dmaPort );
}
```

Description Sets the DMA I/O rate for the active Covox device.

Prototype **WORD dmaSetRate(BYTE** *sampleRate* **);**

Parameters *sampleRate* A **BYTE** value indicating the sampling rate at which to record or play back a sound buffer.

If recording, *sampleRate* can be a value from 0 to 209 (25,386 Hz). If playing, *sampleRate* can be from 0 to 229 (44,191 Hz).

If *sampleRate* is 0, the default value of 132 (9622 Hz) is used. If *sampleRate* is larger than 209 when recording, it is set to 209. If *sampleRate* is larger than 229 when playing, it is set to 229 .

See **Appendix B** for rate value descriptions.

Remarks This function sets the DMA hardware timer on either the Voice Master or Sound Master II, depending on which device was selected with **dmaInit**.

Generally not used by developer, but used by higher level routines in the library.

Return Value If successful, **dmaSetRate** returns a value of **_ERROR_NONE**. The possible return values are:

_ERROR_NONE
_ERROR_DMA_NOT_INITIALIZED

See the last page of this chapter for error code descriptions.

Example

```
// DMASETRT.C
// This program reads 8 bit PCM sound data from a file.
// The 8 bit sound data is then played back using DMA.
// The rate of buffer being played is increased every time
// a key is struck on the keyboard.
//
// The buffer that is used in this program is allocated
// using _dos_allocmem. The buffer is set to 64K.

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <errno.h>
#include <fcntl.h>
#include "cvxdigi.h"

#define _BUFFER_SIZE   0xFFFF        // Size of each DMA buffer.
#define _PCM_FILE      "TEST1.V8"    // File containing sound data.
#define _RATE_STEP     5             // Value used to increase rate.

extern _dmaInProgressFlag;
```

```
extern _dmaDevice;

VOID main( VOID )
{
   HANDLE  pcmFileHandle;
   WORD    pcmSegment;
   LPSTR   pcmBuffer;
   WORD    bytesRead;
   WORD    portAddress;
   BYTE    dmaRate;
   WORD    repeatCount = 1;
   WORD    initError;

   // Allocate memory for buffer used to read 8 bit PCM
   // data from file.
   if( ( _dos_allocmem( ( WORD )( ( _BUFFER_SIZE / 16 ) + 1 ),
                        &pcmSegment ) ) )
   {
      printf( "ERROR : Cannot allocate memory!\n" );
      exit( 0 );
   }
   else
   {
      // dmaplay() requires a far pointer (LPSTR)
      FP_SEG( pcmBuffer ) = pcmSegment;
      FP_OFF( pcmBuffer ) = 0x0000;
   }

   // Initialize DMA. Setting each parameter to _AUTODETECT
   // causes dmaInit to perform a search for the Port,
   // DMA channel, and IRQ setting respectively.
   initError = dmaInit( _AUTODETECT, _AUTODETECT, _AUTODETECT,
                        &portAddress );

   // If the variable _dmaDevice equals 0 then the DMA
   // sub-system was not initialized correctly.
   if( _dmaDevice == 0 )
      printf( "ERROR = %d : dmaInit failed\n", initError );
   else
   {
      // Open a file containing 8 bit PCM data.
      if( _dos_open( _PCM_FILE, O_RDONLY, &pcmFileHandle ) )
         printf( "ERROR : %s not found.\n", _PCM_FILE );
      else
      {
         // Fill buffer with sound data.
         _dos_read( pcmFileHandle, ( LPVOID )pcmBuffer,
                    ( WORD )_BUFFER_SIZE, &bytesRead );

         // Get rate from header.
         dmaRate = pcmBuffer[ _HEADER_RATE_OFFSET ];

         // Bypass header and insert buffer into DMA queue.
         dmaPlay( ( LPSTR )pcmBuffer + _HEADER_LENGTH,
                  ( LONG )bytesRead - _HEADER_LENGTH,
                  dmaRate, repeatCount );

         printf( "Press key increase rate by %d.\n", _RATE_STEP );

         // Loop until DMA has completed.
         while( _dmaInProgressFlag )
         {
```

```
                // If a key is struck,increase rate of DMA output.
                if( _bios_keybrd( _KEYBRD_READY ) )
                {
                   // Make sure that rate does not go above maximum.
                   if( ( dmaRate + _RATE_STEP) < _CVX_RATE_IN_MAXIMUM )
                   {
                      // Increase rate.
                      dmaRate = dmaRate + _RATE_STEP;

                      // Set new DMA rate.
                      dmaSetRate( dmaRate );
                   }
                   else
                      // Display that the rate can be set no higher.
                      printf( "Rate at maximum.\n" );

                   // Clear all key strokes.
                   while( _bios_keybrd( _KEYBRD_READY ) )
                      getch();
                }
             }
          }
       }

       // Uninitialize the DMA system.
       if( dmaUninit() )
          printf( "DMA uninit failure.\n" );

       // Close the sound file.
       _dos_close( pcmFileHandle );

       // Free memory used by 8 bit PCM buffer.
       if( _dos_freemem( pcmSegment ) )
          printf( "ERROR : Cannot free memory!\n" );
    }
```

Description Restarts DMA playback or recording paused by **dmaPause.**

Prototype **BOOL dmaUnpause(VOID);**

Parameters None.

Remarks This function enables the DMA sub-system by enabling the DMA controller on the motherboard.

Return Value If **_TRUE**, the DMA controller was successfully re-enabled.

If **_FALSE**, DMA can not be enabled for one of two reasons. Either **dmaPause** was not previously called, or the DMA sub-system has not been initialized.

Example

See **dmaPause.**

_ERROR_NONE

No errors.

_ERROR_DMA_DETECT_FAILED

The port, IRQ or DMA channel was not found.

_ERROR_DMA_ALREADY_INITIALIZED

The **dmaInit** function has previously been called.

_ERROR_DMA_NOT_INITIALIZED

The **dmaInit** function has not yet been called or DMA could not be initialized.

_ERROR_QUEUE_FULL

No room in FIFO queue for DMA I/O request.

CHAPTER 4

TSRPLAY Functions

Structure

_TSRPLAY_INFO

Functions

tsrFlushFiles	tsrResident
tsrGetVersionString	tsrSetupNewFiles
tsrGetProgramID	tsrStart
tsrPause	tsrUnpause
tsrRemoveSystem	

The Covox 'Terminate and Stay Resident Playback System' (TSRPLAY) is used to play back one or more files in the background using the DMA capabilities of the Voice Master and Sound Master II. This system will play back a sound file of any Covox format.

The interface variables needed to use the functions listed above are defined in the structure **_TSRPLAY_INFO**. **_tsrPlayInfo**, a global variable of this type, is defined in **CVXDIGI.H** and must be used when calling the TSRPLAY interface functions.

After initializing the members of **_tsrPlayInfo**, a call to **tsrStart** will install the TSRPLAY system.

The function **tsrResident** returns **_TRUE** if TSRPLAY has been installed.

After TSRPLAY has been installed, a call to **tsrSetupNewFiles** will replace the currently active list of files and associated sample rates.

The functions **tsrGetVersionString** and **tsrGetProgramID** return the version and program ID of TSRPLAY.

The functions **tsrPause** and **tsrUnpause** may be used to temporarily stop and start TSR playback.

tsrFlushFiles will remove all active files and cause TSRPLAY to become inactive until the function **tsrSetupNewFiles** is called again.

The function **tsrRemoveSystem** completely removes the TSRPLAY system, which includes all memory and interrupt handlers.

NOTE:

The TSRPLAY functions have only been compiled for small model. Therefore, only **SDIGIMC.LIB** and **SDIGIBC.LIB** contain the TSRPLAY functions.

IMPORTANT:

Many of the TSRPLAY functions make calls to the Covox utility functions (see Chapter 9). Therefore, compliled modules that contain TSRPLAY calls must be linked to both **SDIGIMC.LIB** and **SUTILMC.LIB** (Microsoft) or **SDIGIBC.LIB** and **SUTILBC.LIB** (Borland).

Description Structure containing all information needed to call TSRPLAY functions.

Structure

```
typedef struct _tagtsrPlayInfo
{
   WORD fileCount;
   BYTE fileName[ _FILE_COUNT_MAX ][ _FILE_PATH_LENGTH ];
   WORD fileRate[ _FILE_COUNT_MAX ];
   WORD fileRepeat[ _FILE_COUNT_MAX ];
   WORD bufferSize;
   WORD bufferCount;
   WORD dmaPort;
   WORD dmaChannel;
   WORD dmaIRQNumber;
} _TSRPLAY_INFO;
```

Structure Elements

fileCount Number of files (up to 35) queued for playback.

fileName A two-dimensional array specifying the name of each file to be played back.

fileRate An array specifying the sampling rate, in hertz (Hz), of the corresponding file in *fileName*. Valid values are from 4,679 to 44,191. The default value is 9,622 Hz.

fileRepeat A one-dimensional array indicating the number of times to repeat the file specified in *fileName*. The maximum value is 255; the default is 1. A value of -1 causes indefinite repeat.

bufferSize The size, in kilobytes (default 32), of each buffer to be queued up for DMA playback. *bufferSize* is used with a call to **tsrStart**. It cannot be changed after TSRPLAY has been installed. Valid values are from 1 to 64.

bufferCount Number of buffers of *bufferSize* to be queued for DMA playback (default 3). It cannot be changed after TSRPLAY has been installed. Valid values are from 2 to 5.

dmaPort The port jumper setting on the Voice Master or Sound Master II. Valid values are **_CVX_VM0**, **_CVX_VM1**, **_CVX_VM2**, **_CVX_VM3** and **_AUTODETECT** (default).

dmaChannel The DMA Channel jumper setting on the Voice Master or Sound Master II. Valid values are **_DMA_CHANNEL_1**, **_DMA_CHANNEL_3** and **_AUTODETECT** (default).

dmaIRQ-Number — The IRQ jumper setting on the Voice Master or Sound Master II. Valid values are **_IRQ_2**, **_IRQ_3**, **_IRQ_4**, **_IRQ_5**, **_IRQ_7**, and **_AUTODETECT** (default).

Description Removes all active playback files from TSR.

Prototype **VOID tsrFlushFiles(VOID);**

Parameters None.

Remarks TSRPLAY remains in memory after this function is called.

Return Value None.

Example

```
// TSRFLUSH.C
// This program checks to see if TSRPLAY has been installed.
// If it has been installed then all files are flushed
// from TSRPLAY.

#include <stdlib.h>
#include <dos.h>
#include "cvxdigi.h"

VOID main( VOID )
{
   // Check to see if TSRPLAY is resident.
   if( tsrResident() )
   {
      printf( "Flushing %s files.\n", tsrGetProgramID() );

      // Clear all sound files from TSRPLAY.
      tsrFlushFiles();
   }
   else
   {
      printf( "ERROR : %s not installed\n", tsrGetProgramID() );
   }
}
```

tsrGetProgramID

Description Returns the TSRPLAY program ID.

Prototype **PSTR tsrGetProgramID(VOID);**

Parameters None.

Remarks The program ID is of the form "COVOX TSRPLAY".

Return Value A pointer to a string (**PSTR**) containing the program ID of TSRPLAY.

Example

```
// TSRINFO.C
// This program calls tsrGetProgramID and tsrGetVersionString
// and displays the return values to the screen.

#include <stdlib.h>
#include <dos.h>
#include "cvxdigi.h"

VOID main( VOID )
{
   PSTR tsrID;
   PSTR tsrVersion;

   // Get TSRPLAY program ID.
   tsrID = tsrGetProgramID();

   // Get TSRPLAY version.
   tsrVersion = tsrGetVersionString();

   // Display results to the screen.
   printf( " %s version %s\n", tsrID, tsrVersion );
}
```

Description Returns the version of TSRPLAY.

Prototype **PSTR tsrGetVersionString(VOID);**

Parameters None.

Remarks None.

Return Value A pointer to a string (**PSTR**) containing the release version of TSRPLAY. The string will have the following format:

< *major version* > < . > < *minor version* >

(for example, "2.00").

Example

See **tsrGetProgramID**.

tsrPause

Description Pauses TSR playback.

Prototype **VOID tsrPause(VOID);**

Parameters None.

Remarks TSRPLAY is paused by disabling the DMA controller on the motherboard. This function sets an internal flag to reflect the paused state.

Return Value None.

Example

```
// TSRPAUSE.C
// This program checks to see if TSRPLAY has been installed.
// If it has been installed then TSRPLAY is paused. Link the .OBJ
// to both a DIGI and a UTIL library.

#include <stdlib.h>
#include <dos.h>
#include "cvxdigi.h"

VOID main( VOID )
{
   // Check to see if TSRPLAY is resident.
   if( tsrResident() )
   {
      printf( "Pausing %s\n", tsrGetProgramID() );

      // Pause TSR playback.
      tsrPause();
   }
   else
      printf( "ERROR : %s not installed\n", tsrGetProgramID() );
}
```

Description Removes TSRPLAY from memory.

Prototype **VOID tsrRemoveSystem(VOID);**

Parameters None.

Remarks This function removes all hooked interrupts and frees all memory allocated by the function **tsrStart**.

Return Value None.

Example

```
// TSRQUIT.C
// This program checks to see if TSRPLAY has been installed.
// If it has been installed then it is removed from memory
// with a call to tsrRemoveSystem. Otherwise an error message
// is displayed to the screen.

#include <stdlib.h>
#include <dos.h>
#include "cvxdigi.h"

VOID main( VOID )
{
   // Check to see if TSRPLAY is resident.
   if( tsrResident() )
   {
      printf( "Uninstalling %s\n", tsrGetProgramID() );

      // Remove TSRPLAY from memory.
      tsrRemoveSystem();
   }
   else
      printf( "ERROR : %s not installed\n", tsrGetProgramID() );
}
```

tsrResident

Description Checks to see if TSRPLAY code is resident.

Prototype **BOOL tsrResident(VOID);**

Parameters None.

Remarks None.

Return Value **_TRUE** if TSRPLAY code is resident. **_FALSE** if TSRPLAY has not been installed.

Example

See **tsrStart**.

Description Replaces existing playback files with new list of files.

Prototype **VOID tsrSetupNewFiles(VOID);**

Parameters None.

Remarks The header file **CVXDIGI.H** contains the structure **_TSRPLAY_INFO**, which contains an array of file names and the variable indicating the number of files in the array.

Return Value None.

Example

```
// TSRFILES.C
// This program checks to see if TSRPLAY has been installed.
// If it has been installed then a new file list is sent
// to the resident copy of TSRPLAY.

#include <stdlib.h>
#include <string.h>
#include <dos.h>
#include "cvxdigi.h"

#define _FILE_COUNT   3
#define _FILE_1       "TEST1.V8"
#define _FILE_2       "TEST2.V8"
#define _FILE_3       "TEST3.V8"

VOID main( VOID )
{
   // Check to see if TSRPLAY is resident.
   if( tsrResident() )
   {
      printf( "Replacing files for resident %s\n",
              tsrGetProgramID() );

      // Setup 3 files in _tsrPlayInfo structure.
      _tsrPlayInfo.fileCount = _FILE_COUNT;

      // Put new file names into structure.
      strcpy( _tsrPlayInfo.fileName[ 0 ], _FILE_1 );
      strcpy( _tsrPlayInfo.fileName[ 1 ], _FILE_2 );
      strcpy( _tsrPlayInfo.fileName[ 2 ], _FILE_3 );

      // Set 2nd file to repeat twice.
      _tsrPlayInfo.fileRepeat[ 1 ] = 2;

      // Make call into TSRPLAY to replace files.
      tsrSetupNewFiles();
   }
   else
      printf( "ERROR : %s not installed\n", tsrGetProgramID() );
}
```

Description Installs TSR file playback code.

Prototype **WORD tsrStart(VOID);**

Parameters None.

Remarks The file **CVXDIGI.H** contains the structure **_TSRPLAY_INFO** used by **tsrStart**. The only members of this structure that must be set up before calling **tsrStart** are **fileName[][]** and **fileCount**.

The other members of the structure **_TSRPLAY_INFO** are initialized by the TSRPLAY libarary routines.

tsrStart will not return if TSRPLAY is successfully installed.

The TSRPLAY system uses approximately 30K plus memory allocated for each DMA playback buffer of size **_TSRPLAY_INFO.bufferSize.**

The number of buffers is determined by **_TSRPLAY_INFO.bufferCount.**

Return Value If TSRPLAY is installed successfully, **tsrStart** does not return. If an error occurs, one of the following values is returned:

_ERROR_OPENING_FILE
_ERROR_FILE_COUNT
_ERROR_DMA_INIT_FAILED
_ERROR_MEMORY_ALLOC
_ERROR_INVALID_BUFFER_SIZE
_ERROR_ILLEGAL_BUFFER_COUNT
_ERROR_ILLEGAL_PORT
_ERROR_ILLEGAL_DMA_CHANNEL
_ERROR_ILLEGAL_IRQ
_ERROR_ALREADY_RESIDENT

See the last page of the chapter for error code descriptions.

Example

```
// TSRSTART.C
// This program checks to see if TSRPLAY has been installed.
// If TSRPLAY is not resident, the size and number of each DMA
// buffer, the file list and file count in _tsrPlayInfo are
// initialized.

#include <stdlib.h>
#include <string.h>
#include <dos.h>
```

```
#include "cvxdigi.h"

#define _FILE_COUNT             2
#define _FILE_1                 "TEST1.V8"
#define _FILE_2                 "TEST2.V8"
#define _BUFFER_SIZE_DEFAULT    32
#define _BUFFER_COUNT_DEFAULT   3

VOID main( VOID )
{
   // Check to see if TSRPLAY is resident.
   if( !tsrResident() )
   {
      printf( "Installing %s\n", tsrGetProgramID() );

      // Setup 2 files in _tsrPlayInfo structure.
      _tsrPlayInfo.fileCount = _FILE_COUNT;

      // Put new file names into structure.
      strcpy( _tsrPlayInfo.fileName[ 0 ], _FILE_1 );
      strcpy( _tsrPlayInfo.fileName[ 1 ], _FILE_2 );

      // Set up memory for TSRPLAY
      _tsrPlayInfo.bufferSize  = _BUFFER_SIZE_DEFAULT;
      _tsrPlayInfo.bufferCount = _BUFFER_COUNT_DEFAULT;

      // Make call to install TSRPLAY system.
      tsrStart();
   }
   else
      printf( "ERROR : %s has already been installed\n",
              tsrGetProgramID() );
}
```

tsrUnpause

Description Restarts TSR playback after playback was paused by the function **tsrPause.**

Prototype **VOID tsrUnpause(VOID);**

Parameters None.

Remarks None.

Return Value None.

Example

```
// TSRUNPAU.C
// This program checks to see if TSRPLAY has been installed.
// If it has been installed then playback is restarted
// with a call to tsrUnpause.

#include <stdlib.h>
#include <dos.h>
#include "cvxdigi.h"

VOID main( VOID )
{
   // Check to see if TSRPLAY is resident.
   if( tsrResident() )
   {
      printf( "Restarting %s\n", tsrGetProgramID() );

      // Resume TSR playback.
      tsrUnpause();
   }
   else
      printf( "ERROR : %s not installed\n", tsrGetProgramID() );
}
```

_ERROR_NONE

No errors.

_ERROR_ALREADY_RESIDENT

A call to **tsrStart** was attempted after TSRPLAY was already installed.

_ERROR_FILE_COUNT

The file count specified in **_TSRPLAY_INFO.fileCount** was not between 1 and 255.

_ERROR_OPENING_FILE

A file name specified in **_TSRPLAY_INFO.fileName** was not found.

_ERROR_ILLEGAL_BUFFER_SIZE

The value specified in **_TSRPLAY_INFO.bufferSize** was not between 1 and 64.

_ERROR_ILLEGAL_BUFFER_COUNT

The buffer count specified in **_TSRPLAY_INFO.bufferCount** was not between 2 and 5.

_ERROR_ILLEGAL_PORT

The port specified in **_TSRPLAY_INFO.dmaPort** was not one of the valid port values.

_ERROR_ILLEGAL_DMA_CHANNEL

The DMA channel specified in **_TSRPLAY_INFO.dmaChannel** was not one of the valid DMA Channel values.

_ERROR_ILLEGAL_IRQ

The IRQ specified in **_TSRPLAY_INFO.dmaIRQNumber** was not one of the valid IRQ values.

_ERROR_DMA_INIT_FAILED

Initialization of DMA on the Voice Master or Sound Master II failed. Possible hardware conflict.

_ERROR_MEMORY_ALLOC

Requested allocation is larger than the total memory available in system (This error constant is defined in **CVXUTILS.H**).

ERROR_WRONG_DOS_VERSION

The call to **tsrStart** was made in a DOS environment earlier than version 3.1.

CHAPTER 5

BIOS Interrupt 1A Sound Support Using DMA and Timer Interrupt

Low-Level Functions

INT 1Ah : AH = 81h	*Get Sound Status*
INT 1Ah : AH = 82h	*Record Sound Buffer*
INT 1Ah : AH = 83h	*Play Sound Buffer*
INT 1Ah : AH = 84h	*Stop Sound I/O*

The **INT1ATSR.EXE** program contains a set of simulated BIOS routines to record and play 8 bit PCM sound data. With the appropriate values placed into the AH register, interrupt 1A can be invoked to queue a buffer for recording or playback.

There are two types of hardware I/O control in the INT1ATSR program: DMA and timer interrupt. Either type can be selected by means of a command line switch (see the section titled **Running INT1ATSR.EXE**).

The DMA mode has the superior sound quality of the two and will record at a rate of 25,386 Hz and playback at a rate of 44,191 Hz. The timer interrupt mode does not give good results above 8 kHz. Be aware that different types of computers will affect the sound quality differently.

Both Tandy and PC modes are supported and must be selected using command line switches. The main difference between the two modes is how sample rate values are expressed. In PC mode the rate of recording or playback represents the actual frequency in Hz, whereas in Tandy mode a rate value from 3 (fastest) to 4095 (slowest) is used.

When Tandy mode is selected, the volume of sound playback may be adjusted through software. The range of values is 0 to 7; 0 signifies that sound is disabled. PC mode does not support software volume control.

Note: The interrupt 1A I/O queue has a maximum of 155 entries. When the queue is full, an I/O call will cause the carry flag to be set. Therefore, the carry flag should be monitored to determine if a spot is freed in the queue. (See the programming examples for usage)

Running INT1ATSR.EXE

Description: A Terminate and Stay Resident (TSR) utility that hooks interrupt vector 1A. INT1ATSR installs interrupt service routines for recording and playback of a FIFO queue of 8 bit PCM data.

Syntax: **INT1ATSR [OPTIONS]**

Options:

/Pxxxx - The port to use for I/O; xxxx can be one of the following: VM0, VM1, VM2, VM3 for the internal Voice Master (or Sound Master II); or LPT1, LPT2, LPT3 for the external Voice Master or Speech Thing (output only). Use DMA for DMA mode using the internal Voice Master or Sound Master II. The default port is VM0.

/Mx - Select **T** for Tandy sound mode and **P** for PC sound mode. The default is **P**.

/Q - Uninstall INT1ATSR program.

The following apply only to Voice Master and Sound Master II with DMA.

/Jx - Select port jumper setting (0-3). -1 (default) for sequential search.

/Cx - Select DMA channel (1 or 3). -1 (default) checks 1 first, then 3.

/Ix - Select IRQ (2, 3, 4, 5, or 7). -1 (default) checks 7 first, then 2, 3, 4, and 5, in that order.

In the following examples, the address of the interrupt 1A sound support routine is placed into the BIOS, after which the program stays resident.

EXAMPLE: Initialize and direct input and output to the Voice Master II attached to LPT1.

```
C:\intlatsr /pLPT1
```

EXAMPLE: Prepare for DMA output on the Sound Master II board. IRQ 7 and DMA channel 3 have been selected. If there is no Sound Master II board at these settings then an error message will be returned.

```
C:\intlatsr /pDMA /i7 /c3
```

Call With:	*AH*	= 81h
Returns:	*AL*	= 0C4h
	AH	= Number of buffers remaining in queue.
	Carry Flag	= 1 if busy.
		= 0 if not busy.

Notes: This interrupt call will set the carry flag if I/O is being performed, or clear the carry flag if no I/O is being performed. The number of buffers left in the queue is placed in AH upon return.

Example

See the **Record Sound Buffer** and **Play Sound Buffer** examples.

Call With:	*AH*	= 82h
	ES:BX	= Address of buffer.
	CX	= Length of buffer at ES:BX.
	DX	= Sampling Rate.
Returns:	*AH*	= 00
	Carry Flag	= 1 if busy.
		= 0 if not busy.

Notes: This interrupt call is used to queue an empty buffer for PCM recording of sound. If the maximum number of buffers have been queued then the carry flag is set. The carry flag is cleared if the queue request was successful.

Example

```
// I1AREC.C
// This program records _BUFFER_SIZE bytes and writes the
// data to to _FILE_NAME.
//
// NOTE : Before this program can be executed, INT1ATSR.EXE
//        must be loaded into memory.
//
#include <dos.h>
#include <fcntl.h>
#include <errno.h>
#include "cvxdefs.h"

#define  _FILE_NAME    "TEST1.V8" // New sound file.
#define  _BUFFER_SIZE  0x8000     // Bytes to record.

BYTE     buffer[_BUFFER_SIZE];    // Buffer to store
// recorded data.
VOID main( VOID )
{
   WORD   fileHandle;
   WORD   bytesWritten;
   BYTE   carryFlag;

   do
   {
      _asm   mov   bx, SEG buffer    // Load ES:BX with
      _asm   mov   es, bx            // segment and offset
      _asm   mov   bx, OFFSET buffer // of buffer.

      _asm   mov   cx, _BUFFER_SIZE  // Set buffer length.

      _asm   mov   dx, 9622          // Set sample rate.

      _asm   mov   ah, 82h           // Output sound.
      _asm   int   1Ah               //

      _asm   lahf                    // Get carry flag state.
      _asm   and   ah, 1             // If CF = 1 we need to
      _asm   mov   carryFlag, ah     // try request again.
```

```
    } while( carryFlag );

    // Loop here until all output is completed.
    do
    {
       _asm   mov   ah, 81h          // Check sound status.
       _asm   int   1Ah

       _asm   lahf                   // Get carry flag state.
       _asm   and   ah, 1            //
       _asm   mov   carryFlag, ah    //

    } while( carryFlag );

    // Uninitialize the sound I/O facility.
    //
    _asm   mov   ah, 84h
    _asm   int   1Ah

    // Create new file for recorded data.
    if( _dos_creat( _FILE_NAME, _A_NORMAL, &fileHandle ) )
    {
       printf( "ERROR : Cannot create file.\n" );
       printf( "        Sound data lost.\n" );
    }

    // Write recorded data to file.
    if( _dos_write( fileHandle, ( LPVOID )buffer,
                    ( WORD )_BUFFER_SIZE, &bytesWritten ) )
    {
       printf( "ERROR : Cannot write to file.\n" );
       printf( "        Sound data lost.\n" );
    }

    // Close new file.
    _dos_close( fileHandle );
}
```

Call With:	*AH*	= 83h
	ES:BX	= Address of buffer.
	CX	= Length of buffer at ES:BX.
	DX	= Sampling Rate.
	AL	= Volume (Tandy mode only).
Returns:	*AH*	= 00
	Carry Flag	= 1 if Busy.
		= 0 if Not Busy.

Notes: This interrupt call is used to queue a buffer of PCM data for playback. If the maximum number of buffers have been queued then the carry flag is set. The carry flag is cleared if the queue request was successful.

Example

```
// I1APLAY.C
// This program reads in _BUFFER_SIZE bytes from _FILE_NAME.
// A request is made to INT 1Ah to play the sound data from
// _FILE_NAME.
//
// NOTE : Before this program can be executed, INT1ATSR.EXE
//        must be loaded into memory.
//
#include <dos.h>
#include <fcntl.h>
#include <errno.h>
#include "cvxdefs.h"

#define _FILE_NAME    "TEST1.V8"   // Sound data file.
#define _BUFFER_SIZE  0x8000       // Bytes to read from file.

BYTE    buffer[_BUFFER_SIZE];      // Buffer to store data
                                   // read from _FILE_NAME.

VOID main( VOID )
{
   WORD   fileHandle;
   WORD   bytesRead;
   BYTE   carryFlag;

   // Open file containing sound data.
   if( _dos_open( _FILE_NAME, O_RDONLY, &fileHandle ) )
   {
      printf( " ERROR : file %s not found\n", _FILE_NAME );
      exit( 0 );
   }

   // Read sound data from file.
   _dos_read( fileHandle, ( LPVOID )buffer,
              ( WORD )_BUFFER_SIZE, &bytesRead );

   do
   {
```

```
        _asm   mov   bx, SEG buffer    // Load ES:BX with
        _asm   mov   es, bx            // segment and offset
        _asm   mov   bx, OFFSET buffer // of buffer.

        _asm   mov   cx, bytesRead     // Set buffer length.

        _asm   mov   dx, 9622          // Set sample rate.

        _asm   mov   al, 4             // Set volume.

        _asm   mov   ah, 83h           // Output sound.
        _asm   int   1Ah               //

        _asm   lahf                    // Get carry flag state.
        _asm   and   ah, 1             // If CF = 1 we need to
        _asm   mov   carryFlag, ah     // try request again.

    } while( carryFlag );

    // Loop here until all output is completed.
    do
    {
        _asm   mov   ah, 81h           // Check sound status.
        _asm   int   1Ah

        _asm   lahf                    // Get carry flag state.
        _asm   and   ah, 1             //
        _asm   mov   carryFlag, ah     //

    } while( carryFlag );

    // Uninitialize the sound I/O facility.
    _asm     mov   ah, 84h
    _asm     int   1Ah

    // Close sound file.
    _dos_close( fileHandle );
}
```

Call With:	*AH*	= 84h
Returns:	Nothing	
Notes:	This interrupt call stops all sound I/O.	

Example

See the **Record Sound Buffer** and **Output Sound Buffer** examples.

CHAPTER 6

FoxPro/dBase Loadable Sound Drivers

Modules

CVXDMA	CVXPORT
CVXINIT	CVXRATE
CVXIRQ	CVXREC
CVXPLAY	CVXREC2
CVXPLAY2	CVXRECD
CVXPLAYD	CVXUNIN

FoxPro, FoxPro2 and dBase offer the option to load and call a binary file, or module. Covox has designed a set of modules, which are described in this chapter, to record and play back 8 bit digitized sound files using FoxPro, FoxPro2 and dBase.

When recording or playing back digitized sound using the non-dma modules **CVXREC, CVXREC2, CVXPLAY** or **CVXPLAY2**, a buffer is used to temporarily store the sound data being transferred to or from a Covox sound file. This buffer, which determines the length of time available for playing or recording sound, has a fixed size because of the 32K file size limitation in FoxPro and dBase.

Because FoxPro2 allows a loadable file size of up to 64K, developers using FoxPro2 have the option of using the **CVXPLAY2** and **CVXREC2** modules. These two modules have been compiled with a larger buffer to allow longer recording and playback times.

The **CVXRECD** and **CVXPLAYD** modules allow recording and playback to the extent of the hard drive. These modules can use any DMA-capable Covox device for DMA I/O, but will not work when using non-DMA-capable devices such as the Speech Thing or Voice Master II.

Loading a Module

In FoxPro, FoxPro2 or dBase, modules must first be loaded before use, as follows:

LOAD "module name"

Calling a Module

When a module is called in FoxPro, FoxPro2 and dBase applications, a string can be passed to that module. The following syntax must be used:

CALL "CVXPORT" WITH "0"

In this example, **CVXPORT** is the module being called, and **0** is the string. All modules that accept parameters expect the data to be passed as a string.

Module Usage

After all of the modules that are to be used are loaded, the system must be initialized, which is accomplished by calling **CVXINIT**.

All modules use the default settings (Port, DMA, IRQ). To change any one of these settings, a call to **CVXPORT, CVXDMA** or **CVXIRQ** must be made.

After the system has been initialized, a file may be recorded or played back by calling the **CVXRATE** module with the desired rate and then calling any one of the following modules: **CVXPLAY, CVXPLAY2, CVXPLAYD, CVXREC, CVXREC2,** and **CVXRECD**.

Before leaving the FoxPro, FoxPro2 or dBase application, **CVXUNIN** must be called.

Description: Informs all modules at which DMA channel the Covox board is configured.

Parameters: The DMA Channel setting. **CVXDMA** expects one of the following DMA channel values:

1	DMA Channel 1
3	DMA Channel 3

Remarks: The channel passed to **CVXDMA** should coincide with the DMA channel configuration of the board is configured. If the channel is not a legal value, the **CVXRECD** and **CVXPLAYD** modules will not work.

The **CVXDMA** module is only needed when using **CVXRECD** or **CVXPLAYD**. and only works if a DMA-capable device is installed.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXDMA.BIN"
LOAD "CVXRECD.BIN"
LOAD "CVXPLAYD.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Set the DMA channel to 1 (default) **
CALL "CVXDMA" WITH "1"

** Record a large file **
CALL "CVXRECD" WITH "TEST.V8"

** Play back the file **
CALL "CVXPLAYD" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXDMA.BIN"
RELEASE MODULE "CVXRECD.BIN"
RELEASE MODULE "CVXPLAYD.BIN"
```

CVXINIT

Description: Initializes an interrupt handler for transferring information between modules.

Parameters: None.

Remarks: **CVXINIT** is used to set up an interrupt handler on interrupt 2Fh (multiplex interrupt). This handler will be used as a 'link' to pass Port, DMA, IRQ and Rate information between all modules.

This module must be called before any other module.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXPLAY.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Play back the TEST.V8 file **
CALL "CVXPLAY" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXPLAY.BIN"
```

Description: Informs the modules at which IRQ channel the Covox board is configured.

Parameters: The IRQ setting. **CVXIRQ** expects one of the following IRQ values:

2	IRQ 2
3	IRQ 3
4	IRQ 4
5	IRQ 5
7	IRQ 7

Remarks: The IRQ setting passed to **CVXIRQ** should coincide with the IRQ channel that the board is configured to. If these are not the same, the **CVXRECD** and **CVXPLAYD** modules will fail.

The **CVXIRQ** module is only needed when using **CVXRECD** or **CVXPLAYD** and will only operate if a DMA-capable device is installed.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXIRQ.BIN"
LOAD "CVXRECD.BIN"
LOAD "CVXPLAYD.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Set the IRQ channel to 7 (default) **
CALL "CVXIRQ" WITH "7"

** Record a large file **
CALL "CVXRECD" WITH "TEST.V8"

** Play back the file **
CALL "CVXPLAYD" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXIRQ.BIN"
RELEASE MODULE "CVXRECD.BIN"
RELEASE MODULE "CVXPLAYD.BIN"
```

CVXPLAY

Description: Uses polled output to play a previously recorded 8 bit PCM file.

Parameters: The name of the file to be played.

Remarks: **CVXPLAY** plays back any standard 8 bit PCM file. Due to the limited module size in FoxPro and dBase, only a few seconds of sound can be played back.

CVXPLAY does not return to the calling procedure until playback is complete.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXPLAY.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Play back the TEST.V8 file **
CALL "CVXPLAY" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXPLAY.BIN"
```

Description: Uses polled output to play a previously recorded 8 bit PCM file (FoxPro2 only).

Parameters: The name of the file to be played back.

Remarks: **CVXPLAY2** will play back any standard 8 bit PCM file.

Due to the limited module size in FoxPro2, only a few seconds of sound can be played back. The allotted time is twice that of **CVXPLAY**.

CVXPLAY2 does not return to the calling procedure until playback has finished.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXPLAY2.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Play back the TEST.V8 file (FoxPro2 only) **
CALL "CVXPLAY2" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXPLAY2.BIN"
```

CVXPLAYD

Description: Uses DMA output to play a previously recorded 8 bit PCM file.

Parameters: The name of the file to be played back.

Remarks: **CVXPLAYD** can play back any standard 8 bit PCM file in its entirety. This module requires that the file being played be located on a hard drive.

CVXPLAYD does not return to the calling procedure until playback has finished.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXPLAYD.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Play back the TEST.V8 file **
CALL "CVXPLAYD" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXPLAYD.BIN"
```

Description: Inform the modules which port to use for recording and playback.

Parameters: The port address setting. **CVXPORT** expects one of the following port values:

0	Covox board set to port 0x22X.
1	Covox board set to port 0x24X.
2	Covox board set to port 0x28X.
3	Covox board set to port 0x2CX.
4	Speech Thing or Voice Master System II attached to LPT1.

Remarks: If an internal card address is selected, that address should coincide with the jumper position selected on the Covox board. If they are not the same, no I/O can take place.

The **CVXRECD** and **CVXPLAYD** modules will not work if **4** is selected.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXPORT.BIN"
LOAD "CVXPLAY.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Set the port to jumper 0 (default) **
CALL "CVXPORT" WITH "0"

** Play back the TEST.V8 file **
CALL "CVXPLAY" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXPORT.BIN"
RELEASE MODULE "CVXPLAY.BIN"
```

CVXRATE

Description: Sets the sampling rate at which all recording and playback will occur.

Parameters: The desired sampling rate. The rate must be passed as a 3 character string, representing a value from 001 to 209 for recording, from 001 to 229 for playback, or 255 for default settings (see **Remarks** below). If the rate is less than 100, put 0 in the first position(s) of the string (i.e. "062" for 62; "007" for 7).

Remarks: **CVXRATE** uses the standard Covox rate values.

If a rate of 255 is specified, **CVXRATE** will inform the playback modules to use the rate stored in the digitized file header and the recording modules to default to a rate of 132. **CVXRATE** is automatically set to 255 when loaded.

All recording and playback modules will use the sampling rate specified in **CVXRATE**.

See **Appendix B** for rate value descriptions.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXRATE.BIN"
LOAD "CVXPLAY.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Set the rate to 132 **
CALL "CVXRATE" WITH "132"

** Play back the file **
CALL "CVXPLAY" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXRATE.BIN"
RELEASE MODULE "CVXPLAY.BIN"
```

Description: Uses polled input to record a standard 8 bit PCM file.

Parameters: The name of the file to be recorded.

Remarks: **CVXREC** stops recording when the internal buffer space is full or a key is pressed, at which time it writes the recorded data to a file.

Due to the limited module size in FoxPro and dBase, only a few seconds of sound can be recorded with this function.

CVXREC does not return to the calling procedure until recording is complete.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXREC.BIN"
LOAD "CVXPLAY.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Record a small file **
CALL "CVXREC" WITH "TEST.V8"

** Play back the file **
CALL "CVXPLAY" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXREC.BIN"
RELEASE MODULE "CVXPLAY.BIN"
```

CVXREC2

Description: Uses polled input to record an 8 bit PCM file (FoxPro2 only).

Parameters: The name of the file to be recorded.

Remarks: **CVXREC2** continues recording until the internal data space is full or a key is pressed, at which time it writes the sound data to a file.

Due to the limited module size in FoxPro2, only a few seconds of sound can be recorded with this function. The allotted recording time is twice that of **CVXREC**.

CVXREC2 does not return to the calling procedure until recording is complete.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXREC2.BIN"
LOAD "CVXPLAY2.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Record a file (FoxPro2 only) **
CALL "CVXREC2" WITH "TEST.V8"

** Play back the file (FoxPro2 only) **
CALL "CVXPLAY2" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXREC2.BIN"
RELEASE MODULE "CVXPLAY2.BIN"
```

Description: Uses DMA input to record an 8 bit PCM file.

Parameters: The name of the file to be recorded.

Remarks: **CVXRECD** continues recording until a key is pressed or the hard drive is full. This module requires that the file being recorded be located on a hard drive.

CVXRECD does not return to the calling procedure until recording is complete.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXRECD.BIN"
LOAD "CVXPLAYD.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Record a large file **
CALL "CVXRECD" WITH "TEST.V8"

** Play back the file **
CALL "CVXPLAYD" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXRECD.BIN"
RELEASE MODULE "CVXPLAYD.BIN"
```

CVXUNIN

Description: Removes the interrupt handler set up by **CVXINIT.**

Parameters: None.

Remarks: This module must be called before exiting the FoxPro/dBase application.

Example

```
** Load in the sound driver modules **
LOAD "CVXINIT.BIN"
LOAD "CVXUNIN.BIN"
LOAD "CVXPLAY.BIN"

** Initialize the sound system**
CALL "CVXINIT"

** Play back the TEST.V8 file **
CALL "CVXPLAY" WITH "TEST.V8"

** Uninitialize the sound system **
CALL "CVXUNIN"

** Free the modules **
RELEASE MODULE "CVXINIT.BIN"
RELEASE MODULE "CVXUNIN.BIN"
RELEASE MODULE "CVXPLAY.BIN"
```

CHAPTER 7

FM Synthesizer Functions

High-Level Functions

fmInit	fmSetFrequency
fmNoteOff	fmSetInstrument
fmNoteOn	fmSetMode
fmReset	fmUninit

Low-Level Function

fmSetRegister

The library functions in this chapter are designed to play notes using the Yamaha YM3812 FM synthesis chip. All calling programs must include the **CVXFMSY.H** file, which in turn includes all other FM Synthesizer header files.

The **fmInit** function initializes the FM library and the YM3812 chip. Any time after the chip has been initialized, use the **fmReset** function to halt all chip actions and reset the initial register values. Both functions set all voices to piano mode.

Any voice on the chip can be set to a specific instrument by using the **fmSetInstrument** function. These instruments must be of the **_BNK_INSTRUMENT** format defined in **CVXFMSY.H**. Call the **fmSetMode** function to select either melodic or percussive mode.

After a voice is set to an instrument, activate that voice to a particular note with the **fmNoteOn** function. Once a voice has been activated, set it to a different frequency using the **fmSetFrequency** function. To turn the note off, call the **fmNoteOff** function.

A call to **fmUninit** uninitializes the FM library system.

You may also use the **fmSetRegister** function to access the registers directly. See the chapter on **Programming the FM Synthesizer** for a brief outline of these registers.

Description Initializes the FM system and the YM3812 chip.

Prototype **WORD fmInit(_FM_INIT_DATA *** *fmInitData* **)**

Parameters *fmInitData* A structure containing all of the information needed to initialize the FM system and the YM3812 chip.

```
typedef struct
{
   // YM3812 Port.
   WORD port;

} _FM_INIT_DATA;
```

Remarks This function must be called prior to any other of the FM library functions.

Return Value If successful the function returns the value *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_INVALID_PORT

See the last page of the chapter for error code descriptions.

Example

```
// FMTEST.C
// This program initializes, then uninitializes the chip.

#include <stdio.h>
#include "cvxfmsy.h"

VOID main( VOID )
{
   _FM_INIT_DATA   fmInitData;     // Initialization structure.

   // Set the port to port A( 388h )
   fmInitData.port = _FM_PORT_A;

   // Initialize the FM libraries.
   if( !( fmInit( ( _FM_INIT_DATA * )&fmInitData ) ) )
   {
      // Uninitialize the FM system.
      if( fmUninit() )
         printf( "ERROR : fmUninit() not successful.\n" );
   }
   else
      printf( "ERROR : fmInit() not successful.\n" );
}
```

Description Turns off an FM note.

Prototype **WORD fmNoteOff(BYTE** *fmVoice* **)**

Parameters *fmVoice* Voice to deactivate.

Remarks None.

Return Value If successful the function returns the value *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_SYSTEM_NOT_INITIALIZED
_ERROR_VOICE_OUT_OF_RANGE
_ERROR_VOICE_NOT_ACTIVATED

See the last page of the chapter for error code descriptions.

Example

See **fmNoteOn**.

fmNoteOn

Description Turns on an FM note.

Prototype **WORD fmNoteOn(BYTE** *fmVoice*, **WORD** *fmFrequency* **)**

Parameters

fmVoice The voice to activate.

fmFrequency The frequency of the note to play.

Remarks Any note with a value pre-defined in the Covox FM libraries may be called according to the following format:

```
fmNoteOn( 0, _fmMidiScale[ (subscript value) ] );
```

See **CVXFMSY.H** in **Appendix A** for a definition of the **_fmMidiScale** array and the acceptable subscript values (i.e C0, DS4, G7). These values correspond to MIDI note values, offset by one octave. (C0 corresponds to MIDI note 12, C1 corresponds to MIDI note 24, etc.)

Return Value If successful the function returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_SYSTEM_NOT_INITIALIZED
_ERROR_VOICE_OUT_OF_RANGE
_ERROR_INSTRUMENT_NOT_DEFINED
_ERROR_INVALID_INSTRUMENT

See the last page of the chapter for error code descriptions.

Example

```
// FMNOTE.C
// This program first initializes the FM Synthesizer chip, then
// configures voice 0 to piano. It then plays a middle C over
// that voice, turns off the note, and uninitializes the chip.

#include <stdio.h>
#include "cvxfmsy.h"

VOID main( VOID )
{
   _FM_INIT_DATA   fmInitData;     // Initialization structure.
   _BNK_INSTRUMENT bnkInstrument  =
   {
       0,  0,  1,  3,  3, 15,  5,  0,  1,  3,
      15,  0,  0,  0,  1,  0,  7,  0, 15,  7,
       0,  2,  4,  0,  0,  0,  1,  1,  0,  0

   }; // Piano
```

```
    // Set the port to port A( 388h )
    fmInitData.port = _FM_PORT_A;

    // Initialize the FM librarie.s
    if( !( fmInit( ( _FM_INIT_DATA * )&fmInitData ) ) )
    {
       // Set the instrument on voice 0.
       if(!( fmSetInstrument((_BNK_INSTRUMENT far *)&bnkInstrument,
                             0 )))
       {
          // Turn the note on, voice 0 - middle C.
          if( !fmNoteOn( 0, _fmMidiScale[ C5 ] ) )
          {
             // Turn the note off on channel 0.
             if( fmNoteOff( 0 ) )
                printf( "ERROR : fmNoteOff() not successful.\n" );
          }
          else
             printf( "ERROR : fmNoteOn() not successful.\n" );
       }
       else
          printf( "ERROR : fmSetInstrument() not successful.\n" );

       // Uninitialize the FM system.
       if( fmUninit() )
          printf( "ERROR : fmUninit() not successful.\n" );
    }
    else
       printf( "ERROR : fmInit() not successful.\n" );
}
```

fmReset

Description Resets the FM system and the YM3812 chip.

Prototype **WORD fmReset(VOID)**

Parameters None.

Remarks This function is used to reset the FM system and YM3812 chip. It turns off any voices that are currently sounding. Instruments that have a very long release time will continue to sound after a note off has been sent.

Return Value If successful the function returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_SYSTEM_NOT_INITIALIZED

See the last page of the chapter for error code descriptions.

Example

```
// FMREST.C
// This program first initializes the FM Synthesizer chip, then
// configures voice 0 to piano. It then plays a middle C over
// that voice, turns off the note, and resets the chip.

#include <stdio.h>
#include "cvxfmsy.h"

VOID main( VOID )
{
   _FM_INIT_DATA   fmInitData;     // Initialization structure.
   _BNK_INSTRUMENT bnkInstrument  =
   {
       0,  0,  1,  3,  3, 15,  5,  0,  1,  3,
      15,  0,  0,  0,  1,  0,  7,  0, 15,  7,
       0,  2,  4,  0,  0,  0,  1,  1,  0,  0

   }; // Piano

   // Set the port to port A( 388h )
   fmInitData.port = _FM_PORT_A;

   // Initialize the FM libraries.
   if( !( fmInit( ( _FM_INIT_DATA * )&fmInitData ) ) )
   {
      // Set the instrument on voice 0.
      if(!( fmSetInstrument((_BNK_INSTRUMENT far *)&bnkInstrument,
                            0 )))
      {
         // Turn the note on, voice 0 - middle C.
         if( !fmNoteOn( 0, _fmMidiScale[ C5 ] ) )
         {
            // Turn the note off on channel 0.
```

```
            if( fmNoteOff( 0 ) )
               printf( "ERROR : fmNoteOff() not successful.\n" );
         }
         else
            printf( "ERROR : fmNoteOn() not successful.\n" );
      }
      else
         printf( "ERROR : fmSetInstrument() not successful.\n" );

      // Reset the FM Synthesizer chip. This turns off any notes
      // that are currently sounding.
      if( fmReset )
         printf( "ERROR : fmReset() not successful.\n ");

      // Uninitialize the FM system.
      if( fmUninit() )
         printf( "ERROR : fmUninit() not successful.\n" );
   }
   else
      printf( "ERROR : fmInit() not successful.\n" );
}
```

fmSetFrequency

Description Sets the frequency of a currently active voice.

Prototype **WORD fmSetFrequency(BYTE** *fmVoice*, **WORD** *fmFrequency* **)**

Parameters

fmVoice The voice on which to set the frequency.

fmFrequency The frequency of the selected voice.

Remarks You may change the frequency of a currently active voice to that of any note with a value pre-defined in the Covox FM libraries with this function, as follows:

```
fmSetFrequency( 0, _fmMidiScale[ (subscript value) ] );
```

See **CVXFMSY.H** in **Appendix A** for a definition of the **_fmMidiScale** array and the acceptable subscript values (i.e C0, DS4, G7). These values correspond to MIDI note values, offset by one octave. (C0 corresponds to MIDI note 12, C1 corresponds to MIDI note 24, etc.)

Return Value If successful the function returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_SYSTEM_NOT_INITIALIZED

See the last page of the chapter for error code descriptions.

Example

```
// FMSETFRQ.C
// This program first initializes the FM Synthesizer chip, then
// sets voice 0 to a piano configuration. It then plays a middle
// C over that voice, the frequency of which is changed several
// times by the fmSetFrequency() function, called within a loop.
// The program then turns off the note and uninitializes the chip.

#include <stdio.h>
#include "cvxfmsy.h"

VOID main( VOID )
{
   WORD   counterI, counterJ;
   _FM_INIT_DATA   fmInitData;     // Initialization structure.
   _BNK_INSTRUMENT bnkInstrument  =
   {
       0,  0,  1,  3,  3, 15,  5,  0,  1,  3,
      15,  0,  0,  0,  1,  0,  7,  0, 15,  7,
       0,  2,  4,  0,  0,  0,  1,  1,  0,  0

   }; // Piano

   // Set the port to port A( 388h )
```

```
    fmInitData.port = _FM_PORT_A;

    // Initialize the FM libraries.
    if( !( fmInit( ( _FM_INIT_DATA * )&fmInitData ) ) )
    {
       // Set the instrument on voice 0.
       if(!( fmSetInstrument((_BNK_INSTRUMENT far *)&bnkInstrument,
                             0 )))
       {
          // Turn the note on, voice 0 - middle C.
          if( !fmNoteOn( 0, _fmMidiScale[ C5 ] ) )
          {
             // Loop through a series of tones.
             for( counterI = 0; counterI < 86; counterI++ )
             {
                // Do a delay loop.
                for( counterJ = 0; counterJ < 65534; counterJ++ );

                // Set the frequency to a specific tone on voice 0.
                if( fmSetFrequency( 0, _fmMidiScale[ counterI ] ) )
                {
                   printf( "ERROR : fmSetFrequency()" );
                   printf( " not successful.\n" );
                }
             }

             // Turn the note off on channel 0.
             if( fmNoteOff( 0 ) )
                printf( "ERROR : fmNoteOff() not successful.\n" );
          }
          else
             printf( "ERROR : fmNoteOn() not successful.\n" );
       }
       else
          printf( "ERROR : fmSetInstrument() not successful.\n" );

       // Uninitialize the FM system.
       if( fmUninit() )
          printf( "ERROR : fmUninit() not successful.\n" );
    }
    else
       printf( "ERROR : fmInit() not successful.\n" );
}
```

Description Sets up an instrument for playing notes.

Prototype **WORD fmSetInstrument(_BNK_INSTRUMENT far *** *fmInstrument*, **BYTE** *fmVoice* **)**

Parameters

fmInstrument Pointer to the structure type **_BNK_INSTRUMENT**, which has all of the data to define an instrument. See **CVXFMSY.H** in **Appendix A** for the definition of this structure.

fmVoice The voice on which to set the instrument.

Remarks The libraries were originally designed to interface with the **.BNK** file type defined by Ad Lib.

Return Value If successful the function returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_SYSTEM_NOT_INITIALIZED
_ERROR_VOICE_OUT_OF_RANGE
_ERROR_INVALID_VOICE_FOR_MODE
_ERROR_INVALID_INSTRUMENT

See the last page of the chapter for error code descriptions.

Example

```
// FMINSTR.C
// This program first initializes the FM Synthesizer chip, then
// configures voice 0 to piano. It then uninitializes the chip.

#include <stdio.h>
#include "cvxfmsy.h"

VOID main( VOID )
{
   _FM_INIT_DATA   fmInitData;     // Initialization structure.
   _BNK_INSTRUMENT bnkInstrument  =
   {
       0,  0,  1,  3,  3, 15,  5,  0,  1,  3,
      15,  0,  0,  0,  1,  0,  7,  0, 15,  7,
       0,  2,  4,  0,  0,  0,  1,  1,  0,  0

   }; // Piano

   // Set the port to port A( 388h )
   fmInitData.port = _FM_PORT_A;

   // Initialize the FM libraries.
   if( !( fmInit( ( _FM_INIT_DATA * )&fmInitData ) ) )
   {
```

```
        // Set the instrument on voice 0.
        if( fmSetInstrument((_BNK_INSTRUMENT far *)&bnkInstrument,
                            0 ))
           printf( "ERROR : fmSetInstrument() not successful.\n" );

        // Uninitialize the FM system.
        if( fmUninit() )
           printf( "ERROR : fmUninit() not successful.\n" );
     }
     else
        printf( "ERROR : fmInit() not successful.\n" );
}
```

fmSetMode

Description Sets the FM system playback mode.

Prototype **WORD fmSetMode(BYTE *fmPlaybackMode*)**

Parameters *fmPlayback-Mode* The mode to which the chip is set.

_MELODIC Voices 0-9 are active as melodic voices and no percussive voices are available.

_PERCUSSIVE Voice 0-5 are active as melodic voices and 6-10 are percussive voices.

6 - Bass Drum

7 - Snare Drum

8 - Tom Drum

9 - Cymbal

10 - High Hat

Remarks The default mode is **_MELODIC**.

Return Value If successful the function returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_SYSTEM_NOT_INITIALIZED

See the last page of the chapter for error code descriptions.

Example

```
// FMMODE.C
// This program first initializes the FM Synthesizer chip, then
// configures the mode to _MELODIC and voice 0 to piano.
// It then uninitializes the chip.

#include <stdio.h>
#include "cvxfmsy.h"

VOID main( VOID )
{
   _FM_INIT_DATA   fmInitData;     // Initialization structure.
   _BNK_INSTRUMENT bnkInstrument  =
   {
       0,  0,  1,  3,  3, 15,  5,  0,  1,  3,
      15,  0,  0,  0,  1,  0,  7,  0, 15,  7,
       0,  2,  4,  0,  0,  0,  1,  1,  0,  0

   }; // Piano
```

```
    // Set the port to port A( 388h )
    fmInitData.port = _FM_PORT_A;

    // Initialize the FM libraries.
    if( !( fmInit( ( _FM_INIT_DATA * )&fmInitData ) ) )
    {
        // Set the chip to melodic mode.
        if( fmSetMode( _MELODIC ) )
            printf( "ERROR : fmSetMode() not successful.\n" );

        // Set the instrument on voice 0.
        if( fmSetInstrument((_BNK_INSTRUMENT far *)&bnkInstrument,
                            0 ))
            printf( "ERROR : fmSetInstrument() not successful.\n" );

        // Uninitialize the FM system.
        if( fmUninit() )
            printf( "ERROR : fmUninit() not successful.\n" );
    }
    else
        printf( "ERROR : fmInit() not successful.\n" );
}
```

Description Uninitializes the FM system and the YM3812 chip.

Prototype **WORD fmUninit(VOID)**

Parameters None.

Remarks None.

Return Value If successful the function returns a value of *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_SYSTEM NOT INITIALIZED

See the last page of this chapter for error code descriptions.

Example

```
// FMTEST.C
// This program initializes, then uninitializes the chip.

#include <stdio.h>
#include "cvxfmsy.h"

VOID main( VOID )
{
   _FM_INIT_DATA   fmInitData;     // Initialization structure.

   // Set the port to port A( 388h ).
   fmInitData.port = _FM_PORT_A;

   // Initialize the FM libraries.
   if( !( fmInit( ( _FM_INIT_DATA * )&fmInitData ) ) )
   {
      // Uninitialize the FM system.
      if( fmUninit() )
         printf( "ERROR : fmUninit() not successful.\n" );
   }
   else
      printf( "ERROR : fmInit() not successful.\n" );
}
```

Description Writes data to a specific register on the YM3812.

Prototype **VOID fmSetRegister(BYTE** *fmRegister*, **BYTE** *fmData* **)**

Parameters *fmRegister* The destination register.

fmData The data to write to the register.

Remarks There is a small delay internal to this procedure. The delay is needed to allow the YM3812 chip to 'settle'. All functions in the FM library use this function.

Return Value None.

Example

```
// FMSETREG.C
// This program first initializes the FM Synthesizer chip, then
// uses the fmSetRegister() function to set the chip wave select.
// It then uninitializes the chip.

#include <stdio.h>
#include "cvxfmsy.h"

VOID main( VOID )
{
   _FM_INIT_DATA   fmInitData;  // Initialization structure.

   // Set the port to port A( 338h )
   fmInitData.port = _FM_PORT_A;

   // Initialize the FM libraries.
   if( fmInit( ( _FM_INIT_DATA * )&fmInitData ) )
   {
      // Turn off the wave select. This will cause the
      // voices to default to sine wave.
      fmSetRegister( _TEST, _WAVE_SELECT_ON );

      // Uninitialize the FM system.
      if( fmUninit() )
         printf( "ERROR : fmUninit() not successful.\n" );
   }
   else
      printf( "ERROR : fmInit() not successful.\n" );
}
```

_ERROR_NONE

No errors.

_ERROR_INVALID_PORT

Invalid port sent to initialize.

_ERROR_SYSTEM_NOT_INITIALIZED

System is currently not initialized.

_ERROR_INVALID_VOICE_FOR_MODE

An attempt was made to set an instrument on an incompatible voice.
e.g. Bass drum sent to be set up on voice 0.

_ERROR_INVALID_INSTRUMENT

An attempt was made to set an invalid instrument.

_ERROR_INSTRUMENT_NOT_DEFINED

An attempt was made to turn on a voice that does not have an instrument set.

_ERROR_VOICE_NOT_ACTIVATED

An attempt was made to turn off a voice that is not currently active.

_ERROR_VOICE_OUT_OF_RANGE

An attempt was made to set an instrument on a voice that is out of range for the current playback mode.

CHAPTER 8

MIDI Functions

Functions

midiInit
midiFetchByte
midiOutByte
midiUninit

The function **midiInit** initializes the MIDI board, sets up an IRQ handler and a 'First In, First Out' (FIFO) queue to store the MIDI data.

When a byte is sent from a MIDI device to the board, an IRQ 'fires'. In the interrupt handler, the byte sent from the MIDI device is stored in the queue, the counter **midiBytesInQueue** is incremented and the flag **midiByteReady** is set to **_TRUE.**

A call to the function **midiFetchByte** returns a byte from the queue, decrements the counter **midiBytesInQueue** and, if the queue is empty, sets the flag **midiByteReady** to **_FALSE.**

The function **midiOutByte** sends a byte of data directly to the MIDI device.

The function **midiUninit** disables the MIDI board and removes the interrupt handler that was installed by **midiInit.**

Description Initializes the Covox MIDI system.

Prototype **VOID midiInit(WORD** *midiPort*, **BYTE** *irqChannel* **);**

Parameters *midiPort* The port to which the Covox MIDI board is set. Valid ports are:

_MIDI_PORT_A
_MIDI_PORT_B

See **CVXMIDI.H** in **Appendix A** for port value descriptions.

irqChannel The IRQ line to which the MIDI board is set. Valid IRQs are:

_IRQ_2 **_IRQ_5**
_IRQ_3 **_IRQ_7**

See **CVXMIDI.H** in **Appendix A** for IRQ value descriptions.

Remarks The **midiInit** function, in addition to initializing the Covox MIDI board, sets up an interrupt handler to be used by the MIDI system. This IRQ handler takes over the IRQ channel specified in the call to **midiInit.**

This function must be called prior to any other of the MIDI library functions.

Return Value None.

Example

```
// MIDIPLAY.C
// This program initializes the Covox MIDI system using the
// midiInit function, plays a middle C using the midiOutByte
// function, then calls the midiUninit function when finished.

#include "cvxmidi.h"

VOID main( VOID )
{
   // Set up the MIDI system with the default port and IRQ values.
   midiInit( _MIDI_PORT_A, _IRQ_2 );

   // Turn on a middle C note.
   midiOutByte( 0x90 );
   midiOutByte(   60 );
   midiOutByte(   64 );

   // Turn the note off.
   midiOutByte(   60 );
   midiOutByte(    0 );

   // Uninitialize the MIDI system.
   midiUninit();
}
```

Description Gets a byte from the MIDI byte queue.

Prototype **BYTE midiFetchByte(VOID);**

Parameters None.

Remarks When bytes are received from a MIDI device, they are placed into a queue. The **midiFetchByte** retrieves a byte from the queue and adjusts the queue indexes appropriately.

If an attempt is made to fetch a byte from the queue when the queue is empty, the internal queue indexes will be corrupted. To prevent this, a flag (**midiByteReady**) and an internal counter (**midiBytesInQueue**) are available for use.

BYTE midiByteReady	**_TRUE** data in queue. **_FALSE** queue empty.
WORD midiBytesInQueue	The number of bytes in the queue. This value decreases by 1 every time **midiFetchByte** is called.

Return Value The next MIDI byte in the queue.

Example

```
MIDIFTCH.C
// This program initializes the Covox MIDI system, retrieves a
// byte from the MIDI queue using the midiFetchByte function,
// then calls the midiUninit function when finished.

#include "cvxmidi.h"

extern BYTE midiByteReady;
extern WORD midiBytesInQueue;

VOID main( VOID )
{
   // Set up the MIDI system with the default port and IRQ channel
   // values.
   midiInit( _MIDI_PORT_A, _IRQ_2 );

   // Wait for a keypress to terminate routine.
   while( !kbhit() )
   {
      // If a byte is ready, print out the byte and the number
      // of bytes in the queue.
      if( midiByteReady )
         printf( "MIDI Byte: %x Number Of Bytes In Queue: %x\n",
                 midiFetchByte(), midiBytesInQueue );
   }

   // Uninitialize the MIDI system.
   midiUninit();
}
```

midiOutByte

Description Sends a byte to a MIDI device.

Prototype **VOID midiOutByte(BYTE** *midiData* **);**

Parameters *midiData* Byte to send to the MIDI device.

Remarks The **midiOutByte** function has an internal delay that allows the MIDI system enough time to 'settle' between the outputting of bytes.

Return Value None.

Example

```
MIDIPLAY.C
// This program initializes the Covox MIDI system using the
// midiInit function, plays a middle C using the midiOutByte
// function, then calls the midiUninit function when finished.

#include "cvxmidi.h"

VOID main( VOID )
{
   // Set up the MIDI system with the default port and IRQ values.
   midiInit( _MIDI_PORT_A, _IRQ_2 );

   // Turn on a middle C note.
   midiOutByte( 0x90 );
   midiOutByte(   60 );
   midiOutByte(   64 );

   // Turn the note off.
   midiOutByte(   60 );
   midiOutByte(    0 );

   // Uninitialize the MIDI system.
   midiUninit();
}
```

Description Uninitializes the MIDI system.

Prototype **VOID midiUninit(VOID);**

Parameters None.

Remarks The **midiUninit** function restores the original IRQ handler that was replaced earlier by **midiInit**.

Return Value None.

Example

```
MIDIPLAY.C
// This program initializes the Covox MIDI system using the
// midiInit function, plays a middle C using the midiOutByte
// function, then calls the midiUninit function when finished.

#include "cvxmidi.h"

VOID main( VOID )
{
   // Set up the MIDI system with the default port and IRQ values.
   midiInit( _MIDI_PORT_A, _IRQ_2 );

   // Turn on a middle C note.
   midiOutByte( 0x90 );
   midiOutByte(   60 );
   midiOutByte(   64 );

   // Turn the note off.
   midiOutByte(   60 );
   midiOutByte(    0 );

   // Uninitialize the MIDI system.
   midiUninit();
}
```

CHAPTER 9

Covox Utility Functions

Functions

cvxBufferAlloc	cvxFileOpen
cvxBufferFree	cvxFileRead
cvxHzToRate	cvxFileWrite
cvxFileClose	cvxRateToHz
cvxFileCreate	

The functions in this chapter are used for a variety of purposes. They are primarily designed to be used in conjunction with the functions from Chapters 1, 2 and 3.

See the individual function descriptions for information.

cvxBufferAlloc

Description Creates a buffer in which to store data.

Prototype **LPSTR cvxBufferAlloc(LONG** *bufferSize,* **LONG *** *bytesAvailable* **);**

Parameters

bufferSize The desired size, in bytes, of the buffer to be created.

bytesAvailable Returned in this pointer (**LONG ***) is the number of bytes available in the system. If the function was unsuccessful, developers may use this variable to determine how much memory to specify in a subsequent call to *cvxBufferAlloc*.

Remarks The buffer created is of the same type used in the record/play, pack/unpack and dmaPlay/dmaRecord functions (see Chapters 1, 2 and 3).

Return Value A far pointer (**LPSTR**) addressing the actual location in memory of the buffer that was created. If the function was unsuccessful, a **_NULL** pointer is returned.

Example

```
// UTILBUFR.C
// This program creates a buffer of size _BUFFER_SIZE.
// After a buffer is successfully created, the buffer memory is
// deallocated with the function bufferFree.

#include <stdio.h>
#include "cvxutil.h"

#define  _BUFFER_SIZE   300000

VOID main( VOID )
{
   LPSTR  bufferStart;
   LONG   bytesAvailable;

   // Allocate memory for buffer.
   bufferStart = cvxBufferAlloc( _BUFFER_SIZE, &bytesAvailable );

   // Make sure no error has occurred.
   if( bufferStart == _NULL  )
   {
      printf( "ERROR : requested buffer size larger than" );
      printf( "        available memory.\n\n" );
   }
   else
   {
      printf( "Memory requested = %ld\n", _BUFFER_SIZE );
      printf( "Memory available = %ld\n", bytesAvailable );

      // Free memory.
      if( cvxBufferFree( bufferStart ) )
         printf( "ERROR : Memory de-allocation failed.\n" );
   }
}
```

Description Frees memory previously allocated by **cvxBufferAlloc**.

Prototype **WORD cvxBufferFree(LPSTR** *bufferStart* **);**

Parameters *bufferStart* A far pointer addressing the location in memory of the buffer to be destroyed.

Remarks All data that may have been stored in the specified buffer will be lost.

Return Value If successful, **cvxBufferFree** returns the value *_ERROR_NONE*. The possible return values are:

ERROR_NONE
ERROR_MEMORY_DEALLOC

See the last page of the chapter for error code descriptions.

Example

```
// UTILBUFR.C
// This program creates a buffer of size _BUFFER_SIZE.
// After a buffer is successfully created, the buffer memory is
// deallocated with the function bufferFree.

#include <stdio.h>
#include "cvxutil.h"

#define  _BUFFER_SIZE   300000

VOID main( VOID )
{
   LPSTR  bufferStart;
   LONG   bytesAvailable;

   // Allocate memory for buffer.
   bufferStart = cvxBufferAlloc( _BUFFER_SIZE, &bytesAvailable );

   // Make sure no error has occurred.
   if( bufferStart == _NULL )
   {
      printf( "ERROR : requested buffer size larger than" );
      printf( "        available memory.\n\n" );
   }
   else
   {
      printf( "Memory requested = %ld\n", _BUFFER_SIZE );
      printf( "Memory available = %ld\n", bytesAvailable );

      // Free memory.
      if( cvxBufferFree( bufferStart ) )
         printf( "ERROR : Memory de-allocation failed.\n" );
   }
}
```

cvxHzToRate

Description Converts a hertz frequency value to a standard Covox rate value.

Prototype **BYTE cvxHzToRate(WORD** *rateHertz* **);**

Parameters *rateHertz* A **WORD** value representing the actual frequency value (in hertz) to be converted. *rateHertz* must be a value between 4679 and 44,191, or the function will exit immediately and return a value of 0.

To convert from hertz to the corresponding *cvxRate* value, this function uses the following formula:

$$cvxRate = 256 - (1{,}193{,}180 / rateHertz)$$

Remarks This function can be very helpful in determining the proper rate value to use when recording a Covox sound file.

Return Value A **BYTE** value representing a standard Covox sampling rate value.

Example

```
// UTILR2HZ.C
// This program uses the functions cvxRateToHz() and
// cvxHzToRate() to calcuate between the Covox rate and
// the rate in hertz.

#include <dos.h>
#include "cvxutil.h"

VOID main( VOID )
{
   printf( "    1 = %u Hz\n", cvxRateToHz(   1 ) );
   printf( "  132 = %u Hz\n", cvxRateToHz( 132 ) );
   printf( "  210 = %u Hz\n", cvxRateToHz( 210 ) );
   printf( "  211 = %u Hz\n", cvxRateToHz( 211 ) );
   printf( "  223 = %u Hz\n", cvxRateToHz( 223 ) );
   printf( "  229 = %u Hz\n", cvxRateToHz( 229 ) );

   printf( " 4679 = %u cvx \n" , cvxHzToRate(  4679 ) );
   printf( " 9622 = %u cvx \n" , cvxHzToRate(  9622 ) );
   printf( "25938 = %u cvx \n" , cvxHzToRate( 25938 ) );
   printf( "26515 = %u cvx \n" , cvxHzToRate( 26515 ) );
   printf( "36156 = %u cvx \n" , cvxHzToRate( 36156 ) );
   printf( "44191 = %u cvx \n" , cvxHzToRate( 44191 ) );
}
```

Description Closes a file with a handle.

Prototype **WORD cvxFileClose(HANDLE fileHandle);**

Parameters *fileHandle* This value of type **HANDLE** identifies the file to be closed.

Remarks The function **cvxFileClose** uses DOS Int 21h function 3Eh to close a file created with **cvxFileCreate** or opened with **cvxFileOpen.**

Return Value If successful, **cvxFileClose** returns the value *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_INVALID_HANDLE

See the last page of the chapter for error code descriptions.

Example

See **cvxFileCreate.**

Description Creates a new file.

Prototype **WORD cvxFileCreate(LPSTR** *fileName*, **WORD** *fileAttribute*,
HANDLE * *fileHandle* **);**

Parameters

fileName A far pointer (**LPSTR**) that points to a zero-terminated ASCII string representing the full path and file name of the file to create.

fileAttribute A **WORD** value indicating the attibutes of the new file.

_CREATE_NORMAL
_CREATE_R_ONLY

fileHandle A pointer to a variable of type **HANDLE**.

Remarks The funtion **cvxFileCreate** uses DOS Int 21h function 3Ch to create a new file. If *fileName* exists, it is opened and its length truncated to zero.

Return Value If successful, **cvxFileCreate** returns the value *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_PATH_NOT_FOUND
_ERROR_TOO_MANY_FILES_OPEN
_ERROR_ACCESS_DENIED

See the last page of the chapter for error code descriptions.

Example

```
// UTILRW.C
// This program writes the contents of one file (_FILE_1)
// into a newly created file (_FILE_2) using the functions
// cvxFileRead() and cvxFileWrite(). The file being copied cannot
// be larger than the available system memory.
// Memory is allocated with the function cvxBufferAlloc.

#include <io.h>
#include <dos.h>
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include "cvxutil.h"

#define  _FILE_1       "TEST1.V8"
#define  _FILE_2       "TEST2.V8"

VOID main( VOID )
{
```

```
    LPSTR  bufferStart;
    LONG   bufferSize;
    HANDLE handle1, handle2;
    LONG   bytesRead, bytesWritten;
    LONG   bytesAvailable;

    // Open file containing sound data.
    if( cvxFileOpen( _FILE_1, _OPEN_R_ONLY, &handle1 ) )
    {
       printf( "ERROR : file %s not found.\n", _FILE_1 );
       exit( 0 );
    }

    // Set number of bytes to allocate.
    bufferSize = filelength( handle1 );

    // Allocate memory for buffer.
    bufferStart = cvxBufferAlloc( bufferSize, &bytesAvailable );

    // Error handling.
    if( bufferStart == _NULL  )
    {
       printf( "Error allocating memory.\n" );
       printf( "Memory available = %ld\n", bytesAvailable );
       printf( "Memory requested = %ld\n", bufferSize     );
       exit( 0 );
    }

    // Read from file.
    if( cvxFileRead( handle1, bufferStart, bufferSize, &bytesRead ) )
       printf( "ERROR : Cannot read %s.\n", _FILE_1 );
    else
    {
       // Create new file.
       if( cvxFileCreate( _FILE_2, _CREATE_NORMAL, &handle2 ) )
          printf( "ERROR : Cannot create %s.\n", _FILE_2 );
       else
       {
          // Write to new file.
          if( cvxFileWrite( handle2, bufferStart, bytesRead,
                            &bytesWritten ) )
             printf( "ERROR : Cannot write to %s.\n", _FILE_2 );

          cvxFileClose( handle2 );
       }
    }

    cvxFileClose( handle1 );

    // Free memory used by bufferStart.
    if( cvxBufferFree( bufferStart ) )
       printf( "Error de-allocating memory.\n" );
}
```

cvxFileOpen

Description Opens a file and assigns a file handle.

Prototype **WORD cvxFileOpen(LPSTR** *fileName*, **WORD** *fileAccess*,
HANDLE * *fileHandle*);

Parameters

fileName A far pointer (**LPSTR**) that points to a zero-terminated ASCII string representing the full path and file name of the file to create.

accessMode A **WORD** value indicating the access mode to open file.

_OPEN_R_ONLY
_OPEN_W_ONLY
_OPEN_RW

fileHandle A pointer to a variable of type **HANDLE**.

Remarks The funtion **cvxFileOpen** uses DOS Int 21h function 3Dh to open a file.

Return Value If successful, **cvxFileOpen** returns the value *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_FILE_NOT_FOUND
_ERROR_PATH_NOT_FOUND
_ERROR_TOO_MANY_FILES_OPEN
_ERROR_ACCESS_DENIED
_ERROR_INVALID_ACCESS

See the last page of the chapter for error code descriptions.

Example

See **cvxFileCreate**.

Description Reads in the contents of a file and places the data into a buffer allocated by **cvxBufferAlloc.**

Prototype **WORD cvxFileRead(HANDLE** *fileHandle*, **LPSTR** *bufferStart*, **LONG** *bufferSize*, **LONG *** *bytesRead* **);**

Parameters

fileHandle A handle specifying the file to be read.

bufferStart A far pointer (**LPSTR**) addressing the location in memory of a buffer that will contain data read from the specified file.

bufferSize A **LONG** variable specifying the size of the buffer pointed to by *bufferStart*.

bytesRead A pointer to a variable of type **LONG**, indicating the number of bytes actually read from the file.

Remarks Before using this function, the file must first be opened with a DOS file handle function. Likewise, developers must close the file after file access is complete.

Return Value If successful, **cvxFileRead** returns the value *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_FILE_ACCESS_DENIED
_ERROR_INVALID_HANDLE
_ERROR_FILE_READ_FAILED

See the last page of the chapter for error code descriptions.

Example

See **cvxFileCreate.**

Description Writes the contents of a buffer into a file.

Prototype **WORD cvxFileWrite(HANDLE** *fileHandle*, **LPSTR** *bufferStart*,
LONG *bufferSize*, **LONG *** *bytesWritten* **);**

Parameters

fileHandle	An arbitrary handle name for accessing the file to be read.
bufferStart	A far pointer (**LPSTR**) addressing the location in memory of a buffer that contains data to be written to the specified file.
bufferSize	A **LONG** variable specifying the number of bytes to be written to the file.
bytesWritten	A pointer to a variable of type **LONG**, indicating the number of bytes actually written to the file.

Remarks Before using this function, the file must first be opened with a DOS file function. Likewise, developers must close the file after file access is complete.

Return Value If successful, **cvxFileWrite** returns the value *_ERROR_NONE*. The possible return values are:

_ERROR_NONE
_ERROR_FILE_ACCESS_DENIED
_ERROR_INVALID_HANDLE
_ERROR_FILE_WRITE_FAILED

See the last page of the chapter for error code descriptions.

Example

See **cvxFileCreate**.

Description Converts a standard Covox rate value to a hertz frequency value.

Prototype **WORD cvxRateToHz(BYTE** *cvxRate* **);**

Parameters *cvxRate* A **BYTE** value representing the sampling rate value used in most Covox functions. *cvxRate* may be a value between 1 (4679 Hz) and 229 (44,191 Hz).

If *cvxRate* is 0, or if it is greater than 229, the function exits with a return value of 0.

To convert from *cvxRate* to a rate measured in hertz, this function uses the following equation:

hertz = 1,193,180 / (256 - *cvxRate*).

Remarks This function can be very helpful in determining the actual sampling rate of a pre-recorded Covox sound file.

Return Value A **WORD** value representing the actual frequency (in hertz) of the Covox rate value passed to the function. If unsuccessful, **cvxRateToHz** returns 0.

Example

```
// UTILR2HZ.C
// This program uses the functions cvxRateToHz() and
// cvxHzToRate() to calcuate between the Covox rate and
// the rate in hertz.

#include <dos.h>
#include "cvxutil.h"

VOID main( VOID )
{
   printf( "    1 = %u Hz\n", cvxRateToHz(   1 ) );
   printf( "  132 = %u Hz\n", cvxRateToHz( 132 ) );
   printf( "  210 = %u Hz\n", cvxRateToHz( 210 ) );
   printf( "  211 = %u Hz\n", cvxRateToHz( 211 ) );
   printf( "  223 = %u Hz\n", cvxRateToHz( 223 ) );
   printf( "  229 = %u Hz\n", cvxRateToHz( 229 ) );

   printf( " 4679 = %u cvx \n" , cvxHzToRate(  4679 ) );
   printf( " 9622 = %u cvx \n" , cvxHzToRate(  9622 ) );
   printf( "25938 = %u cvx \n" , cvxHzToRate( 25938 ) );
   printf( "26515 = %u cvx \n" , cvxHzToRate( 26515 ) );
   printf( "36156 = %u cvx \n" , cvxHzToRate( 36156 ) );
   printf( "44191 = %u cvx \n" , cvxHzToRate( 44191 ) );
}
```

_ERROR_NONE

No errors.

_ERROR_MEMORY_ALLOCATION

Size of memory request exceeds available memory in system.

_ERROR_INVALID_HANDLE

File not open.

_ERROR_FILE_READ_FAILED

Unknown file read failure.

_ERROR_FILE_WRITE_FAILED

Unknown file write failure.

_ERROR_FILE_NOT_FOUND

Non-existent file specified in cvxFileOpen.

_ERROR_PATH_NOT_FOUND

Invalid path specified.

_ERROR_TOO_MANY_FILES_OPEN

No more handles available in system.

_ERROR_ACCESS_DENIED

File could not be accessed.

_ERROR_INVALID_ACCESS

Invalid mode during open/create.

CHAPTER 10

Programming With Polled (Non-DMA) I/O

A programmer must time the input and output to ensure accurate sample rates when recording or playing digitized audio files. Polling is one method of timing I/O. There are two types of I/O polling, referred to as software polling and hardware polling. Both types employ a 'loop' to determine the interval between successive I/O instructions.

Software polling uses a simple delay loop, and is not recommended for most applications, especially if they are to be used on systems with varying CPU clock speeds. Hardware polling is a commonly used and accurate form of timing I/O delays.

Software Polling

Software polling utilizes loop constructs, such as 'for-next' in BASIC, 'for' in C, or 'loop' in assembly. A loop counter can then be used to determine when the next I/O instruction should be performed. This method is not recommended because the same software loop will execute differently on different processors (i.e., an 8088 processor executes loops much slower than a 80386 processor executes the same loop).

In the C code example shown below, a byte is output to the sound card port, **vmPort** at the end of every delay loop. The variable **bufferStart** is a (**LPSTR**) and points to a buffer of PCM data. The loop is exited when **bufferStart** points to the last PCM byte which is pointed to by the **LPSTR** variable **bufferEnd.**

```
while( ( loopCounter < _LOOP_SIZE ) &&
       ( bufferStart != bufferEnd ) )
{
   // See if it's time to output a PCM byte.
```

```
        if( _LOOP_SIZE == loopCounter++ )
        {
            // Output byte of PCM data.
            outp( vmPort, *bufferStart++ );

            // Reset the loop counter.
            loopCounter = 0;
        }
    }
```

Hardware Polling

Hardware polling is more consistent among different processors than software polling. The PC's timer chip, the Intel 8254, is used to time the interval between I/O instructions. This timer has three independent timer channels designated as 0, 1, and 2. Timer channels 0 and 2 are available for use in timing I/O. Using timer 0 is referred to as 'interrupt driven I/O' because an interrupt is fired when its counter value decrements to zero. Timer 0 is used by DOS to update the system time. Timer 1 is used for memory refresh and should not be used. Timer 2 is the most convenient channel to use for hardware polling.

All three timer channels have a unique port containing a counter value that is decremented at a rate of 1,193,180 times per second (Hertz). Sampling periods can be implemented by setting the length of time it takes the timer to count down between I/O instructions. With proper initialization, when the timer value reaches 0x00, it wraps around to 0xFF and continues decrementing. Setting the least significant byte (LSB) of the timer to 0x00 and waiting for it to decrement to a pre-calculated terminal value will determine the sampling rate. Use the following formula to calculate the value for the timer to count down to:

terminalCount = 256 - (1,193,180 / hertzRate)

In the following C code, the timer on the PC is programmed to count down to a calculated value determined by the rate stored in **_RATE_IN_HERTZ.** The timer chip is initialized with the value 0x98 (counter 2, read/write LSB, software triggered strobe, binary). The maximum countdown value is then written to the timer port. This port is polled while the countdown value remains greater than the variable **terminalCount. bufferStart** is a far pointer to a string **(LPSTR)** that points to the PCM data being output. When **bufferEnd (LPSTR)** is equal to **bufferStart** the 'while' loop is exited and the timer is set back to its original rate. Timer 0 must be disabled during polling so that slower computers will not inadvertently be affected by interrupt 8. Timer 2 must be enabled by setting bit 0 of the Programmable Peripheral Interface (port 61).

For a working programming example, see **HARDPOLL.C.**

```
//*****************************************************
// The following code fragment plays back a buffer of
// sound data by polling timer counter 2.
//*****************************************************

// Calculate the value to count down to.
terminalCount = 256 - ( 1193180 / _RATE_IN_HERTZ );

// Save off current contents of Programmable Peripheral
// Interface.
port61 = inp( 0x61 );

// Enable timer 2 clock input.
outp( 0x61, ( port61 | 0x01 ) );

// Initialize the 8254 timer counter 2.
// Select R/W LSB, software triggered strobe, binary.
outp( 0x43, 0x98 );

// Set timer 2 to maximum countdown value (LSB only).
outp( 0x42, 0x00 );

// Disable timer 0 (int 8).
outp( 0x21, ( inp( 0x21 ) | 0x01 );

// Loop here and output all PCM bytes in the buffer
// pointed to by bufferStart.
while( bufferStart != bufferEnd )
{
   // Loop while timer 2 value is greater than
   // terminalCount.
   while( inp( 0x42 ) > terminalCount );

   // Output sound data byte and increment buffer to next
   // byte.
   outp( vmPort, *bufferStart++ );

   // Reset timer 2 to maximum countdown value.
   outp( 0x42, 0x00 );
}

// Initialize the 8254 timer counter 2.
outp( 0x43, 0xB6 );

// Write the LSB of the original timer countdown value.
outp( 0x42, 0xD0 );

// Write the MSB of the original timer countdown value.
outp( 0x42, 0x04 );

// Restore original contents of PPI.
outp( 0x61, port61 );

// Reenable timer 0 (int 8).
outp( 0x21, ( inp( 0x21 ) & 0xFE ) );
```

8254 Timer Bit Settings

The Intel 8254 timer chip or its equivalent is found on both the PC motherboard and all DMA-capable Covox boards. The following is a description of the format of the 8254 Control Word. On the PC, the port of the 8254 Control Word is 0x43. On the Voice Master and Sound Master II cards, it is found at the Base Port plus offset 0x0B.

Bits 7 - 6 These bits select which hardware timer is being manipulated.

00	Select timer 0
01	Select timer 1
10	Select timer 2

Bits 5 - 4 Read/write format specification.

00	Counter latch command
01	Read/Write only LSB
10	Read/Write only MSB
11	Read/Write LSB, then MSB

Bits 3 - 1 This bit settings select the count mode of the specified counter. Mode 3 should be selected when setting up counter for polling I/O.

000	Mode 0 : Interrupt on terminal count
001	Mode 1 : Hardware retriggerable one-shot
010	Mode 2 : Rate generator
011	Mode 3 : Square wave mode
100	Mode 4 : Software triggered strobe
101	Mode 5 : Hardware triggered strobe

bit 0 PCs only utilize the binary counter setting.

0	Binary counter
1	BCD counter

CHAPTER 11

Programming with Direct Memory Access

The Covox Voice Master and Sound Master II are equipped for playback and recording using 'Direct Memory Access' (DMA). In applications that use digitized sound, DMA substantially reduces the CPU overhead.

During normal operation, the CPU provides address and control information by driving the system bus and must be programmed to move data between I/O (the Covox DMA sound card) and memory. The CPU is not required to use its own processing time to read a byte from, or write a byte to memory when DMA is used.

The DMA controller has four DMA channels. Two of the channels are used internally by the PC. Channels 1 and 3 are available to be used by hardware interfaces such as the Covox DMA boards.

DMA Cycle (hardware)

This is a step-by-step description of a DMA cycle on the PC. Each step occurs during every DMA cycle.

Step 1. The Covox DMA board sends a DRQ (DMA ReQuest) signal to the DMA controller, requesting a DMA transfer on a specific channel.

Step 2. The CPU waits for the current bus cycle to finish and waits long enough to allow the DMA controller time to access the bus.

Step 3. The DMA controller responds by issuing a DACK (DMA ACKnowledge). The Covox DMA card then reads a byte from the A/D converter (if it's a DMA read cycle), or writes a byte to the D/A converter (if it's a DMA write cycle).

Step 4. Once the data is available on the bus, it is placed into a memory location with an address the DMA controller generates automatically. The memory pointer is then incremented or decremented depending upon how the DMA controller is programmed.

After the programmed amount of data has been transferred, the DMA controller issues a TC (Terminal Count) signal on the bus. The DMA card uses this signal to generate an IRQ (Interrupt ReQuest) so as to inform the software that the memory transfer is complete. The IRQ that is fired is jumper selectable on the board. If a handler is set up, another DMA sequence can be started within the handler.

Overview of Hardware Initialization

Several steps are required to initialize both the DMA controller on the PC motherboard and the Covox DMA board. Steps 1-4 and 8 are for the DMA controller; steps 5-7 and 9 are for the Covox DMA board.

Note: These steps are listed in the recommended order for initializing a DMA sequence, but it is possible to modify the order to some extent.

Step 1. Disable the DMA channel.
Step 2. Set the processing mode (write or read).
Step 3. Set the physical address of the DMA sequence to process.
Step 4. Set the number of bytes for a DMA sequence

Step 5. Disable the DMA section of the Covox sound board.
Step 6. Initialize the timer on the DMA board and set the sample rate.
Step 7. Enable the interrupt latch on the DMA board.

Step 8. Enable the DMA controller.
Step 9. Enable the Covox DMA board to begin the DMA process.

The above steps are explained in detail on the following pages.

Step 1. Disable the DMA Channel.

Port 0x0A = DMA Mask Register

Before setting up the DMA controller, the DMA channel being utilized should be disabled. The register used to disable and enable a specific DMA channel is the DMA Mask Register at port 0x0A.

Bits 7 - 3 Not used.

Bit 2 This bit enables or disables the DMA channel specified in bits 1-0.

1 = Disable channel
0 = Enable channel

Bits 1 - 0 These bits specify the DMA channel to be enabled or disabled.

00 = Channel 0
01 = Channel 1
10 = Channel 2
11 = Channel 3

For example, to disable channel 1, output a 0x05 to the DMA Mask Register.

Step 2. Set the processing mode.

Port 0x0B = DMA Mode Register

The DMA mode is set using the DMA Mode Register port on the DMA controller (0x0B). The standard mode used for transferring sound data is: single mode, address increment, no reinitialization, read/write. The channel you select is determined by the DMA channel the DMA board is using (jumper select).

Bits 7 - 6 Transfer Mode: Demand, Single, Block, or Cascade. Single mode is used for standard sound data transfer. This implies that only one byte is transferred at each DREQ.

00 = Demand mode
01 = Single mode (normally used)
10 = Block mode
11 = Cascade mode

Bit 5 Addressing Mode: Increment, Decrement. This specifies whether the address being accessed by the DMA controller is incremented or decremented after each DMA cycle.

0 = Address increment (normally used)
1 = Address decrement

Bit 4 Automatic Reinitialization. If this bit is selected, the DMA controller will automatically restart itself when it reaches the end of the DMA sequence. Otherwise DMA processing stops. Normally, automatic reinitialization is disabled.

0 = No reinitialization (normally used)
1 = Automatic reinitialization

Bits 3 - 2 Transfer Direction: Read, Write, Verify. Read is used to transfer from memory to the bus. Write is used to transfer from the bus to memory. Verify is not used.

00 = Verify
01 = Read (play)
10 = Write (record)

Bits 1 - 0 Channel on which to initialize mode parameters. A separate mode can be set for each DMA channel. The following describes the DMA Mode Register bits:

00 = Channel 0
01 = Channel 1 (used by Covox DMA board)
10 = Channel 2
11 = Channel 3 (used by Covox DMA board)

Step 3. Set the physical address of the DMA sequence to process.

Port 0x02 = DMA Address Register for channel 1.
Port 0x06 = DMA Address Register for channel 3.
Port 0x83 = DMA Page Register for channel 1.
Port 0x82 = DMA Page Register for channel 3.

In standard PC programming, memory addressing is accessed as "segment : offset". Each segment starts on a paragraph (16-byte) boundary and is 64K bytes long. Segments overlap each other every paragraph. The offset is the index into the segment. In physical memory, addressing memory is separated into contiguous pages of 64K byte chunks. There are 16 of these chunks per megabyte of memory.

The CPU performs the conversion from "segment : offset" to physical "page : offset" internally, but the DMA controller does not have this ability. Therefore, the conversion must be performed before you can write to the DMA controller.

The conversion from "segment : offset" to physical "page : offset" is performed by shifting the segment left by four and adding the offset to the result. Shifting left by four effectively multiplies the segment by 16. The resulting 20 bit value allows addressing of up to a megabyte (1M). See the following C code example:

```
(LONG)physAddress = ((LONG)segment << 4) + (LONG)offset;
```

The physical page can be extracted by shifting the physical address right by 16. The page value is only 4 bits, but must be output as a byte to the DMA Page Register. In C, this is accomplished by the following method:

```
(BYTE)physPage = (LONG)physAddress >> 16;
```

The lower 2 bytes of the 20 bit physical address represent the offset from the physical page. For example,

```
(WORD)physOffset = (WORD)physAddress;
```

Step 4. Set the number of bytes for a DMA sequence.

Port 0x03 = DMA Count Register for channel 1.
Port 0x07 = DMA Count Register for channel 3.

The size of the sequence to process is simply the number of bytes that you wish to transfer, up to a maximum of 0xFFFF bytes (64K).

Note: These control registers are both read and write. Therefore it is possible to monitor the Count Register for managing a circular buffer or for timing synchronization.

Covox DMA Board Initialization

All registers on the Covox DMA boards are at an offset from the Base Port, which is configured by a jumper on the board. The valid ports are 0x220, 0x240, 0x280 and 0x2C0. These examples will assume the base port is at the factory default, 0x220.

Step 5. Disable the DMA section of the Covox DMA Board.

Before altering any of the DMA controls on the DMA board, DMA must first be disabled. To do this, output a 0 to the Disable DMA Offset (Port + 0x0D). In C, this is done as follows:

```
outp( ( WORD )( 0x220 + 0x0D ), 0 );
```

Any value written to the Disable DMA Offset will disable DMA on the board.

Step 6. Initialize the timer on the Covox DMA Board and set the sample rate.

The DMA board was designed with a timer (Intel 82C54), to time the firing of DREQ signals, which determines the sample rate. To use the timer it must first be initialized. It may be initalized at any time, except during the processing of a DMA sequence. However, initialization is only required once after each hardware reset (cold boot).

To initialize the timer, a 0xB6 must be written to the Base Port plus 0x0B. For example, in C this can be accomplished as follows:

```
outp( ( WORD )( 0x220 + 0x0B ), 0xB6 );
```

After initialization of the 82C54, set it to the desired sampling rate using the following formula (in this case, expressed as a C code statement):

```
timerValue = 7,190,900 / sampleRate
```

In this statement, *sampleRate* is the desired sample rate in hertz, and *timerValue* is the value to send to the timer. For example, an 8000 Hz sample rate would require the timer to be programmed with an 899 (decimal). Since the timer can only accept 8 bit values, send the value out in LSB/MSB format.

The timer port on the DMA board is located at the Base Port + 0x0A. In C, the timer could be programmed as follows:

```
outp( (WORD)(0x220 + 0x0A), (BYTE)timerValue );
outp( (WORD)(0x220 + 0x0A), (BYTE)(timerValue >> 8) );
```

Step 7. Enable the interrupt latch on the Covox DMA board.

After the DMA sequence has finished, prompt the PC to respond to an IRQ by writing an arbitrary value to the Base Port + 0x0C. For example (in C),

```
outp( ( WORD )( 0x220 + 0x0C ), 0 );
```

Starting the DMA Process

Step 8. Enable the DMA controller.

Port 0x0A = Mask Register

The DMA controller is enabled by setting the enable/disable bit of port 0x0A.

Bits 7 - 3	Not used
Bit 2	1 = Disable channel 0 = Enable channel
Bits 1 - 0	00 = Channel 0 01 = Channel 1 10 = Channel 2 11 = Channel 3

e.g. To enable channel 1, write a 0x01 to the Mask Register.

Step 9. Enable the Covox DMA Board to begin the DMA process.

After DMA has been enabled on the DMA controller, DMA can be enabled on the DMA board by writing an arbitrary value to the Enable DMA Offset. In C, this port could be accessed by the following statement:

```
outp( ( WORD )( 0x220 + 0x0E ), 0 );
```

The DMA sequence will now begin. When the sequence finishes, an IRQ will fire (if the IRQ Enable flip-flop is enabled on the DMA board).

Programming Considerations

- All of the registers on the DMA controller accept only one byte at a time. When outputting a two byte value, you must first output the LSB, followed by the MSB. The Clear LSB/MSB flip-flop register at port 0x0C must be output to first to insure that the DMA controller accepts the bytes in LSB/MSB order. Any value can be output to the Clear LSB/MSB flip/flop register.

- Disabling and enabling of the DMA subsystem can cause an IRQ to fire.

- When the terminal count is reached on the 8237 DMA controller, the DACK line on the Covox DMA sound board goes high. To set this line to low and thus allow the next interrupt to occur, you must perform the following statements in the interrupt handler: The base port 0x220 used in the following C code example.

```
// Enable IRQ latch on the Covox DMA board.
outp( ( 0x220 + 0x0C ), 0 );

// Send 'End Of Interrupt' signal to the IRQ controller.
outp( 0x20, 0x20 );
```

Multiple Page DMA Processing

The DMA hardware can only access one 64K physical page at any one time. What if a buffer lies across more than one page? Unfortunately the only way to handle this is to break it into smaller sequences.

This can be done by comparing the size of the buffer to process with the number of bytes between the current offset of the buffer into the page and the end of the page. If the number of bytes to process extends beyond the page boundary, then only the number of bytes between the offset into the page and the end of the page can be processed as one sequence.

At the end of a DMA sequence the DMA controller generates an interrupt. An interrupt handler must then calculate the new page and offset values. The new page value is calculated by adding 1 to the previous page value, and the offset into the page is set to zero. If the number of bytes left in the buffer is greater than 64K, the next sequence will again have to handle the page crossing calculations in the interrupt handler. These steps are repeated until the entire buffer has been processed.

CHAPTER 12

Programming The FM Synthesizer

Overview

The Covox Sound Master II contains an FM synthesizer chip designed by Yamaha. This chip, the YM3812, is the same chip used by Ad Lib. All software written to support the Ad Lib sound card also supports the FM Synthesizer of the Sound Master II.

FM is an abbreviation for Frequency Modulation, a system of modulating sound waves to create combinations of higher harmonics. This allows for the generation of waveforms containing high harmonics and non-harmonic sounds.

Voice Modes

The YM3812 is equipped with a total of three voicing modes:

a. **9 Melodic Voices Mode**

This mode allows for simultaneous voicing of nine FM sounds having different voices. Both the rhythm bit (R) and CSM bit must be set to 0.

b. **6 Melodic/5 Rhythm Voices Mode**

When the YM3812 is set to this mode, the number of melody sounds which can be simultaneously voiced is reduced by three to six, but five rhythm sounds are added (bass drum, snare drum, tom tom, cymbal, and high hat). The bass drum is created using FM sound generation, the tom tom by sine wave, and the other three rhythm instruments are simulated by composite frequencies.

c. **CSM Mode**

This mode is for future upgrades to the YM3812, and is not supported in these libraries.

Oscillators

The YM3812 chip includes a built-in vibrato oscillator and an amplitude modulation oscillator, which may be activated on selected voices to closely simulate the sound of natural instruments. This allows for a reduction in required programming.

The YM3812 is capable of producing 9 FM sounds over 9 different channels, each of which effectively utilizes two operator cells, in parallel or cascading modulation. Note, however, that the chip actually has only one operator cell, so that this operator cell must be accessed a total of 18 times for 9 FM sounds. The order (slot number) in which signals pass through this operator cell corresponds directly to the sequence of register numbers.

The F-number data for each channel controls both slots of that channel. The relationship between the two slots is such that the first slot is always the modulating wave, and the second slot is the carrier wave. The FM feedback mode affects the modulator slot, if activated.

This is a table of the relationship between channels and slots:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	2	3	1	2	3	4	5	6	4	5	6	7	8	9	7	8	9
1			2			1			2			1			2		
20	21	22	23	24	25	28	29	2A	2B	2C	2D	30	31	32	33	34	35
C0	C1	C2	C0	C1	C2	C3	C4	C5	C3	C4	C5	C6	C7	C8	C6	C7	C8

Slot Number
Channel Number
Slot Number for Each Channel
Data/Slot Relation
Data/Chan. Relation

On the following page is a chart depicting the registers of the YM3812 chip. Any of the registers may be accessed directly using the **setRegister** function. See the **FM Synthesizer Functions** chapter for a description of **setRegister**.

For additional information on the YM3812 chip, contact Yamaha.

Address	D7	D6	D5	D4	D3	D2	D1	D0	Comment
01	Test								Test Data of LSI
02	N/A								
03	N/A								
04	N/A								
08	CSM								CSM Bit For Future Upgrades
20	AM	VIB	EGTYP	KSR	Multiple				AM-Modulation Vibrato EG-Type Key-Scale Rate Multiple
35									
40	KSL		TL						Key-Scale Level Total Level
55									
60	AR				DR				Attack Rate Decay Rate
75									
80	SL				RR				Sustain Level Release Rate
95									
A0	F-Number (L)								Key-on Block F-Number
A8									
B0	N/A		KON	Block			F-Num (H)		
B8									
BD	AM DEP	VIB DEP	R	BD	SD	TOM	TC	HH	Depth (AM/VIB) Rhythm (BD, SD, TOM, TC, HH)
C0	N/A				FB			C	Feedback Connection
C8									
E0	N/A						WS		Wave Select
F5									

CHAPTER 13

Programming The Covox MIDI

Covox has designed an internal 'Musical Instrument Digital Interface' (MIDI) for the PC. This interface, which is contained in both the Covox MIDI Maestro and the Sound Master II cards, can be used to sample data from a MIDI device such as a keyboard or a MIDI guitar. The Covox MIDI uses a standard UART (Universal Asynchronous Receiver/Transmitter) and meets the 1.0 International Standards Organization (ISO) specifications..

The interface was designed so that information can be received and transmitted "simultaneously". The following documentation will explain how to receive information from and send information to a MIDI device using this interface. Three programs have been included with this documentation to better demonstrate how to access the interface (**TESTIN.C, TESTOUT.C** and **MIDI.ASM**).

MIDI Initialization

Setting up the interface to transmit data is a relatively simple task. First you must determine the port address that the interface is using. The two available port addresses for the Covox MIDI are 330h (default) and 338h.

Then, to initialize the card, write a 3h and then a 95h to the STATUS REGISTER (base port + 1). The interface will then be ready to transmit and receive data. The interface needs a little time to "settle" after a byte is written to it. To allow the user to know when the interface is ready to accept data, bit 1 (0 relative) of the STATUS REGISTER will be set. Before writing any data to the card this bit must be checked. If this bit is not HIGH, the data written will become garbled.

This is the procedure for setting up the card and writing out a byte:

```
;
; It is assumed that the port is 330h.
```

```
        ;
        ; Output a 3h to the STATUS REGISTER (base port + 1)
        ; to reset the interface.
        ;
        mov     dx, 330h        ; base port
        inc     dx              ; base port + 1
        mov     al, 3h
        out     dx, al

        ;
        ; Output a 95h to the STATUS REGISTER to setup the
        ; interface to transmit and receive data.
        ;
        mov     al, 95h
        out     dx, al
```

The above block of code is only necessary when initializing the interface. Initialization is the same for both transmitting and receiving.

```
        ;
        ; Poll bit 1 of the STATUS REGISTER to see if the
        ; interface is ready to accept data.
        ;
Poll:   in      al, dx

        ;
        ; "AND" the value read from the STATUS REGISTER with
        ; a 2h to see if bit 1 is HIGH.
        ;
        and     al, 2h
        cmp     al, 0

        ;
        ; If bit 1 is not HIGH jump back up to the
        ; Poll: loop until the  bit goes high.
        ;
        je      Poll

        ;
        ; Output the byte to the DATA REGISTER (base port).
        ;
        mov     al, byte
        dec     dx
        out     dx, al
```

When data is transmitted to the interface from a MIDI device, the interface "fires" an IRQ. In the IRQ handler, the byte that was transmitted must be read in from the DATA REGISTER. If the byte is not read, no further interrupts will occur. The MIDI interface can be configured to use interrupt request channel (IRQ) 2 (default), 3, 5 or 7.

```
        ;
        ; Read the MIDI byte from the DATA REGISTER and
        ; store it in al.
        ;
        mov     dx, 330h
        in      al, dx
```

For more information on how to use the interface, see **MIDI.ASM**.

Two executable programs have been included for testing (**TESTIN.EXE, TESTOUT.EXE**). Both of these require command line switches that specify which Port Address and IRQ Channel the board is configured to.

For example (with the MIDI set to the default settings):

```
testin  816 2
testout 816
```

The first parameter represents the Port Address (decimal) and the second represents the IRQ Channel.

To compile these programs type:

```
cl /c testin.c
tasm internal.asm
link testin+internal;

cl /c testout.c
tasm internal.asm
link testin+internal;
```

APPENDIX A

Covox Library Include Files

CVXDIGI.H

The following header file must be included in your program when using any of the non-DMA play/record, dmaPlay/dmaRecord, pack/unpack, or TSRPLAY functions.

```
//*****************************************************************************
// File              : CVXDIGI.H
// Date              : January 10, 1992
// Description       : Covox developers support library header file for
//                     play/record, pack/unpack, dmaPlay/dmaRecord and
//                     tsrplay functions.
//
// Last update       :
// Additional Notes  :
//*****************************************************************************
//          Copyright (c) 1992, Covox, Inc. All Rights Reserved
//*****************************************************************************

#ifndef _CVXDIGI_DEFINED
#define _CVXDIGI_DEFINED

#include "cvxdefs.h"

//*****************************************************************************
//  If compiled with Borland C, the header file CVXBRLND.H will be
//  automatically included.
//*****************************************************************************
#if defined(__TURBOC__)
#include "cvxbrlnd.h"
#endif
```

```
//*****************************************************************************
// Covox logical port identifiers
//*****************************************************************************
#define _CVX_VM0            0   // Port 0x22X
#define _CVX_VM1            1   // Port 0x24X
#define _CVX_VM2            2   // Port 0x28X
#define _CVX_VM3            3   // Port 0x2CX
#define _CVX_LPT1           4   // LPT 1
#define _CVX_LPT2           5   // LPT 2
#define _CVX_LPT3           6   // LPT 3
#define _CVX_ISP            8   // Internal speaker

//*****************************************************************************
// Enumeration of constants used to specify silence range for record and
// play functions.
//*****************************************************************************
enum
{
    _SILENCE_0,             // No silence encoding.
    _SILENCE_1,             // For relatively quite environments.
    _SILENCE_2,
    _SILENCE_3,             // The default value.
    _SILENCE_4,
    _SILENCE_5              // For relatively noisy environments. Some
};                          // clipping will result at this level.

//*****************************************************************************
// Play - Record format identifiers
//*****************************************************************************
#define _FORMAT_V2S         0x82    // 2 bit ADPCM with silence encoding.
#define _FORMAT_V3S         0x83    // 3 bit ADPCM with silence encoding.
#define _FORMAT_V4S         0x84    // 4 bit ADPCM with silence encoding.
#define _FORMAT_V8S         0x88    // 8 bit PCM with silence encoding.
#define _FORMAT_V8          0x08    // 8 bit PCM
#define _FORMAT_VMF         0xFE    // 8 bit PCM old Covox file format.

//*****************************************************************************
// Covox sound file header information.
//*****************************************************************************
#define _HEADER_LENGTH              16
#define _HEADER_FORMAT_OFFSET       8
#define _HEADER_RATE_OFFSET         9

#define _CVX_RATE_DEFAULT           132
#define _CVX_RATE_IN_MAXIMUM        209
#define _CVX_RATE_OUT_MAXIMUM       229

//*****************************************************************************
// DMA channel identifiers
//*****************************************************************************
#define _DMA_CHANNEL_1      0x01    // Channel 1 identifier
```

```
#define _DMA_CHANNEL_3          0x03    // Channel 3 identifier

//*****************************************************************************
// IRQ vector identifiers
//*****************************************************************************
#define _IRQ_2                  0x02    // IRQ 2 identifier
#define _IRQ_3                  0x03    // IRQ 3 identifier
#define _IRQ_4                  0x04    // IRQ 4 identifier
#define _IRQ_5                  0x05    // IRQ 5 identifier
#define _IRQ_7                  0x07    // IRQ 7 identifier

//*****************************************************************************
// Constant used by dmaInit for auto-detection for port IRQ and DMA channel.
//*****************************************************************************
#define _AUTODETECT             -1

//*****************************************************************************
// DMA (and IRQ) related port offsets
//*****************************************************************************
#define _DMA_MASK_REGISTER          0x0A    // Used to enable or disable specific
                                            //    DMA channels.
#define _DMA_MODE_REGISTER          0x0B    // Used to setup the DMA transfer.
#define _DMA_CLEAR_REGISTER         0x0C    // Used to reset DMA controller to
                                            //    LSB/MSB format.
#define _DMA_STATUS_REGISTER        0x08    // Used to determine which DMA channel
                                            //    is serviced during testing.
#define _DMA_RESET_REGISTER         0x0D    // DMA controller master reset port.
#define _DMA_CH1_PAGE_REGISTER      0x83    // DMA channel 1 page address register.
#define _DMA_CH1_ADDR_REGISTER      0x02    // DMA channel 1 address register.
#define _DMA_CH1_COUNT_REGISTER     0x03    // DMA channel 1 word count register.
#define _DMA_CH3_PAGE_REGISTER      0x82    // DMA channel 3 page address register.
#define _DMA_CH3_ADDR_REGISTER      0x06    // DMA channel 3 address register.
#define _DMA_CH3_COUNT_REGISTER     0x07    // DMA channel 3 word count register.
#define _IRQ_COMMAND_REGISTER       0x20    // Used to acknowledge interrupts.
#define _IRQ_MASK_REGISTER          0x21    // Used to enable or disable specific.
                                            //    interrupt sources.
#define _COM2_CONTROL_REGISTER      0x04    // Offset from base com2 port for modem
                                            //    control register.
#define _LPT1_CONTROL_REGISTER      0x02    // Offset from base lpt1 port for control
                                            //    register .

//*****************************************************************************
// DMA (and IRQ) related masks and vector numbers
//*****************************************************************************
#define _DMA_MODE_ENABLE            0x48    // Used to set the DMA transfer mode.
#define _DMA_DISABLE_MASK           0x04    // use to disable a DMA channel.
#define _IRQ3_VECTOR                0x0B    // IRQ 3 vector number.
#define _IRQ4_VECTOR                0x0C    // IRQ 4 vector number.
#define _IRQ5_VECTOR                0x0D    // IRQ 5 vector number.
#define _IRQ7_VECTOR                0x0F    // IRQ 7 vector number.
#define _IRQ3_MASK                  0x08    // Determines if IRQ3 is the interrupt.
```

```
#define _IRQ4_MASK              0x10    // Determines if IRQ4 is the interrupt.
#define _IRQ5_MASK              0x20    // Determines if IRQ5 is the interrupt.
#define _IRQ7_MASK              0x80    // Determines if IRQ7 is the interrupt.
#define _IRQ3_ENABLE_MASK       0xF7    // For enabling IRQ 3 via 8259 mask reg.
#define _IRQ4_ENABLE_MASK       0xEF    // For enabling IRQ 4 via 8259 mask reg.
#define _IRQ5_ENABLE_MASK       0xD7    // For enabling IRQ 5 via 8259 mask reg.
#define _IRQ7_ENABLE_MASK       0x7F    // For enabling IRQ 7 via 8259 mask reg.
#define _IRQ3_DISABLE_MASK      0x08    // For disabling IRQ 3 via 8259 mask reg.
#define _IRQ4_DISABLE_MASK      0x10    // For disabling IRQ 4 via 8259 mask reg.
#define _IRQ5_DISABLE_MASK      0x20    // For disabling IRQ 5 via 8259 mask reg.
#define _IRQ7_DISABLE_MASK      0x80    // For disabling IRQ 7 via 8259 mask reg.
#define _COM2_DISABLE_MASK      0xF7    // For disabling interrupts from COM 2.
#define _LPT1_DISABLE_MASK      0xEf    // For disabling interrupts from LPT 1.

//*****************************************************************************
// Error codes for play and record functions.
//*****************************************************************************
#undef _ERROR_NONE
#define _ERROR_NONE 0

// NOTE : Starting offset for all play and record functions errors is 1000.
enum
{
    _ERROR_PORT_INIT_FAILED = 1000,     // Unable to find specified port.
    _ERROR_INVALID_BUFFER_SIZE,         // bufferSize was found to be zero or
                                        //  less than 16.
    _ERROR_INVALID_FORMAT,              // Buffer format different from play call.
    _ERROR_PLAY8_BUSY                   // play8 function already active.
};

//*****************************************************************************
// Prototypes for low level routines used by all Play and Record routines.
//*****************************************************************************
WORD    playByteInit        ( WORD, BYTE, BOOL, WORD );
WORD    recordByteInit      ( WORD, BYTE, BOOL );
VOID    playByte            ( BYTE );
BYTE    recordByte          ( VOID );
WORD    playUninit          ( VOID );
WORD    recordUninit        ( VOID );

//*****************************************************************************
// Prototypes for all Play and Record routines.
//*****************************************************************************
WORD    record2s            ( LPSTR, LONG, BYTE, WORD, WORD, LONG * );
WORD    record3s            ( LPSTR, LONG, BYTE, WORD, WORD, LONG * );
WORD    record4s            ( LPSTR, LONG, BYTE, WORD, WORD, LONG * );
WORD    record8s            ( LPSTR, LONG, BYTE, WORD, WORD, LONG * );
WORD    record8             ( LPSTR, LONG, BYTE, WORD, LONG * );

WORD    playAny             ( LPSTR, LONG, BYTE, WORD, BOOL, BOOL );
WORD    play2s              ( LPSTR, LONG, BYTE, WORD, BOOL, BOOL );
```

```
WORD    play3s              ( LPSTR, LONG, BYTE, WORD, BOOL, BOOL );
WORD    play4s              ( LPSTR, LONG, BYTE, WORD, BOOL, BOOL );
WORD    play8s              ( LPSTR, LONG, BYTE, WORD, BOOL, BOOL );
WORD    play8               ( LPSTR, LONG, BYTE, WORD, BOOL );

//****************************************************************************
// Data structure and prototypes for unpackFirst and unpackNext
//****************************************************************************
typedef struct _tagCovertDataStruct
{
    LPSTR       sourcePointer;
    LONG        sourceLength;
    LONG        sourceUsed;
    LPSTR       destinationPointer;
    LONG        destinationLength;
    LONG        destinationFilled;
    WORD        bufferFormat;
    BYTE        sampleRate;
    WORD        silenceRange;
    BOOL        trebleFlag;
    BOOL        noiseFlag;
} _CONVERT_DATA;

WORD    unpackFirst         ( _CONVERT_DATA far * );
WORD    unpackNext          ( _CONVERT_DATA far * );
WORD    packFirst           ( _CONVERT_DATA far * );
WORD    packNext            ( _CONVERT_DATA far * );


//****************************************************************************
// Error codes for DMA functions.
//****************************************************************************

// NOTE : Starting offset for all DMA functions errors is 3000.
enum
{
    _ERROR_DMA_DETECT_FAILED = 3000,    // VM port, IRQ and channel not found.
    _ERROR_DMA_ALREADY_INITIALIZED,     // dmaInit called more than once.
    _ERROR_DMA_NOT_INITIALIZED,         // dmaInit not called first or DMA could
                                        //   not be initialized.
    _ERROR_QUEUE_FULL                   // No room in queue for DMA I/O request.
};

//****************************************************************************
// Prototypes for DMA play and record routines.
//****************************************************************************
WORD    dmaPlay             ( LPSTR, LONG, BYTE, WORD );
WORD    dmaRecord           ( LPSTR, LONG, BYTE );

WORD    dmaSetRate          ( BYTE );
WORD    dmaIRQReset         ( VOID );
```

```
WORD    dmaEnable           ( VOID );
WORD    dmaDisable          ( VOID );
WORD    dmaInit             ( WORD, WORD, WORD, WORD * );
WORD    dmaUninit           ( VOID );
BOOL    dmaPause            ( VOID );
BOOL    dmaUnpause          ( VOID );

// Voice Master and Sound Master II port detection functions.
WORD    dmaPortDetect       ( WORD );

// Functions called after dmaInit to find active DMA settings
WORD    dmaGetPort          ( WORD * );
WORD    dmaGetChannel       ( WORD * );
WORD    dmaGetIRQNumber     ( WORD * );

// I/O Queue Management
WORD    dmaNumberInQueue    ( VOID );
WORD    dmaBytesRemaining   ( VOID );
WORD    dmaFlushQueue       ( VOID );

//***************************************************************************
//  TSRPLAY header information
//***************************************************************************
//
#define _BUFFER_SIZE_MIN        1       // Minimum and maximum size in K bytes
#define _BUFFER_SIZE_MAX        64      //    allowed for each TSRPLAY buffer.
#define _BUFFER_COUNT_MIN       2       // Minimum number of DMA buffers.
#define _BUFFER_COUNT_MAX       5       // Maximum number of DMA buffers.
#define _FILE_REPEAT_MAX        255     // Maximum value for the
                                        //    variable fileRepeatCount.
#define _RATE_MINIMUM           4679    // Minimum and maximum rate setting
#define _RATE_MAXIMUM           44191   //    allowed in the array fileRate[].
#define _FILE_COUNT_MAX         35      // This constant specifies the maximum
                                        //    for playback in TSRPLAY.
#define _FILE_PATH_LENGTH       80      // Maximum length of file and path.

//***************************************************************************
//  Error codes for TSRPLAY
//***************************************************************************
// NOTE : Starting offset for all TSRPLAY errors is 4000.
enum
{
_ERROR_ALREADY_RESIDENT = 4000,     // A call was made to tsrStart after
                                    //    TSRPLAY had been previously installed.
_ERROR_FILE_COUNT,                  // _TSRPLAY_INFO.fileCount is out of range.
_ERROR_OPENING_FILE,                // _TSRPLAY_INFO.fileName not found or
                                    //    error opening file.
_ERROR_ILLEGAL_BUFFER_SIZE,         // _TSRPLAY_INFO.bufferSize is out of range.
_ERROR_ILLEGAL_BUFFER_COUNT,        // _TSRPLAY_INFO.bufferCount is out of range.
_ERROR_ILLEGAL_PORT,                // _TSRPLAY_INFO.dmaPort is not valid.
_ERROR_ILLEGAL_DMA_CHANNEL,         // _TSRPLAY_INFO.dmaChannel is not valid.
```

```
_ERROR_ILLEGAL_IRQ,              // _TSRPLAY_INFO.dmaIRQNumber is not valid.
_ERROR_DMA_INIT_FAILED,          // Initialization of DMA failed.
_ERROR_DMA_UNINIT_FAILED,        // Uninitialization of DMA failed.
_ERROR_DMA_PLAY_FAILED,          // Insert DMA queue playback request failed.
_ERROR_WRONG_DOS_VERSION         // DOS version is less than 3.1
};

//*************************************************************************
// Structure containing all information needed to call TSRPLAY interface
// functions.
//
typedef struct _tagtsrPlayInfo
{
    WORD    fileCount;                          // Number of files queued for playback.
    BYTE    fileName[ _FILE_COUNT_MAX ][ _FILE_PATH_LENGTH ]; // Playback files.
    WORD    fileRate[ _FILE_COUNT_MAX ];        // Rate for each file in fileName.
    WORD    fileRepeat[ _FILE_COUNT_MAX ];      // Stores the number of times to
                                                //   playback list of files.
    WORD    bufferSize;                         // Size of each buffer to be queued up
                                                //   for DMA playback.
    WORD    bufferCount;                        // Number of buffers to be queued for
                                                // DMA playback at one time.
    WORD    dmaPort;                            // Variables for DMA I/O of PCM data.
    WORD    dmaChannel;                         //
    WORD    dmaIRQNumber;                       //
} _TSRPLAY_INFO;

extern      _TSRPLAY_INFO _tsrPlayInfo;

//*************************************************************************
// TSRPLAY interface function prototypes.
//
WORD    tsrStart            ( VOID );
BOOL    tsrResident         ( VOID );
VOID    tsrRemoveSystem     ( VOID );
VOID    tsrPause            ( VOID );
VOID    tsrUnpause          ( VOID );
VOID    tsrFlushFiles       ( VOID );
VOID    tsrSetupNewFiles    ( VOID );
PSTR    tsrGetVersionString ( VOID );
PSTR    tsrGetProgramID     ( VOID );

#endif
//*************************************************************************
// end      CVXDIGI.H
//*************************************************************************
```

CVXFMSY.H

The following must be included in your program when using any of the Covox FM synthesizer functions.

```
//*****************************************************************************
// File              : CVXFMSY.H
// Date              : January 10, 1992
// Description       : Header file for the FM synthesizer libraries.
//
// Last update       :
// Additional Notes  :
//*****************************************************************************
//        Copyright (c) 1992, Covox, Inc. All Rights Reserved
//*****************************************************************************

#ifndef _CVXFMSY_DEFINED
#define _CVXFMSY_DEFINED

#include "cvxdefs.h"

//*****************************************************************************
// If compiled with Borland C, the header file CVXBRLND.H will be
// automatically included.
//*****************************************************************************
#if defined(__TURBOC__)
#include "cvxbrlnd.h"
#endif

//*****************************************************************************
// Structure definition for main header in .BNK files.
typedef struct
{
    WORD      fileVersion;          // File version lsb/msb.
    BYTE      ADLIB[ 6 ];           // Signature "ADLIB-".
    WORD      entriesUsed;          // Number of list entries used.
    WORD      entriesInFile;        // Number of list entries in file.
    LONG      nameListOffset;       // Absolute file offset of the name list.
    LONG      instrumentDataOffset; // Absolute file offset of instrument data.
    BYTE      filler[ 8 ];          // Filler( set to 0 ).
} _BNK_HEADER;

// Structure definition for instrument headers in .BNK files.
typedef struct
{
    WORD      instrumentDataIndex;  // Index of instrument into data section.
    BYTE      recordUsed;           // Record used flag ( 0 - used, 1 - not used ).
    BYTE      instrumentName[ 9 ];  // Instrument Name.
} _BNK_INSTRUMENT_NAME;
```

```
// Structure definition for .BNK modulator/carrier parameters.
typedef struct
{
    BYTE   mode;                 // Mode ( 0 - melodic, 1 - percussive ).
    BYTE   voiceNumber;          // Voice number( if percussive ).
    BYTE   modKSL;               // Key scaling level.
    BYTE   modFreqMultiplier;    // Frequency Multiplier.
    BYTE   modFeedbackLevel;     // Feedback Level.
    BYTE   modAttack;            // Attack Rate.
    BYTE   modSustain;           // Sustain Level.
    BYTE   modSustainStatus;     // Sustaining Sound Status
                                 //   ( 0 - diminishing, 1 - continuing ).
    BYTE   modDecay;             // Decay Rate.
    BYTE   modRelease;           // Release Rate.
    BYTE   modOutputLevel;       // Output Level.
    BYTE   modAmplitudeVibrato;  // Amplitude Vibrato.
    BYTE   modFrequencyVibrato;  // Frequency Vibrato.
    BYTE   modKSR;               // Key Scale Rate.
    BYTE   modConnection;        // Connection ( 1 - parallel, 0 - cascading ).
    BYTE   carKSL;               // Key scaling level.
    BYTE   carFreqMultiplier;    // Frequency Multiplier.
    BYTE   carFeedbackLevel;     // Feedback Level.
    BYTE   carAttack;            // Attack Rate.
    BYTE   carSustain;           // Sustain Level.
    BYTE   carSustainStatus;     // Sustaining Sound Status
                                 //   ( 0 - diminishing, 1 - continuing ).
    BYTE   carDecay;             // Decay Rate.
    BYTE   carRelease;           // Release Rate.
    BYTE   carOutputLevel;       // Output Level.
    BYTE   carAmplitudeVibrato;  // Amplitude Vibrato.
    BYTE   carFrequencyVibrato;  // Frequency Vibrato.
    BYTE   carKSR;               // Frequency Vibrato.
    BYTE   unused;               // Unused.
    BYTE   modWaveform;          // Unused.
    BYTE   carWaveform;          // Wave form for carrier.
} _BNK_INSTRUMENT;

// Structure for the initialization data to be passed to fmInit function.
typedef struct
{
    WORD  port;                  // Variable for the port of the Yamaha chip.
} _FM_INIT_DATA;

// Structure for the global parameters
typedef struct
{
    WORD      port;              // Variable for the port of the Yamaha chip.
    BYTE      systemInitialized; // System initialized flag.
    BYTE      mode;              // Variable for the mode
                                 //   ( 0 -Melodic, 0x30 - Percussive ).
} _FM_GLOBAL_DATA;
```

```
// Defines for modulator and carrier.
enum
{
    _MODULATOR,
    _CARRIER
};

// Defines for percussion instrument.
enum
{
    _BASS_DRUM    = 0x06,
    _SNARE_DRUM,
    _TOM,
    _CYMBAL,
    _HIGH_HAT
};

#define _FM_PORT_A          0x388  // The valid FM synthesizer ports on the
#define _FM_PORT_B          0x380  // Sound Master II.

#define _TEST               0x01   // The FM synthesizer control registers.
#define _AM_MULTI           0x20   //
#define _KSL_TL             0x40   //
#define _AR_DR              0x60   //
#define _SL_RR              0x80   //
#define _FNUM_L             0xA0   //
#define _KON_FNUM_H         0xB0   //
#define _PERC               0xBd   //
#define _FB_C               0xC0   //
#define _WS                 0xE0   //

#define _WAVE_SELECT_OFF    0x00   // Mask to enable/disable wave select.
#define _WAVE_SELECT_ON     0x20   //

#define _SLOTS              0x10   // Number of slots per register.

#define _MEL_VOICES         0x09   // The number of melodic voices.

#define _PERC_VOICES        0x0B   // The number of percussive voices.

#define _VOICES             0x0B   // The total number of voices.

#define _PERCUSSIVE         0x20   // For percussive and melodic mode.
#define _MELODIC            0x00   //

#define _PERC_CLEAR         0xC0   // Mask to clear percussive mode.

#define _RESET_DEPTH        0x00   // Mask to clear the depth register.

#define _RESET_KON          0xDF   // Mask to reset the kon bits.
```

```
#define _TOT_VOICES          0x0B    // The total number of voices on the chip.

#define _KON                 0x20    // The bit position of the kon bits.

// Enumeration for all of the available MIDI notes.
enum
{
    C0, CS0, D0, DS0, E0, F0, FS0, G0, GS0, A0, AS0, B0,
    C1, CS1, D1, DS1, E1, F1, FS1, G1, GS1, A1, AS1, B1,
    C2, CS2, D2, DS2, E2, F2, FS2, G2, GS2, A2, AS2, B2,
    C3, CS3, D3, DS3, E3, F3, FS3, G3, GS3, A3, AS3, B3,
    C4, CS4, D4, DS4, E4, F4, FS4, G4, GS4, A4, AS4, B4,
    C5, CS5, D5, DS5, E5, F5, FS5, G5, GS5, A5, AS5, B5,
    C6, CS6, D6, DS6, E6, F6, FS6, G6, GS6, A6, AS6, B6,
    C7, CS7, D7, DS7, E7, F7, FS7, G7, GS7, A7, AS7, B7,
    C8, CS8, D8, DS8, E8, F8, FS8
};

// External data references between modules.
extern  WORD                _fmPort;
extern  BYTE                _fmKSRMULTIReg      [];
extern  BYTE                _fmKSLTLReg         [];
extern  BYTE                _fmARDRReg          [];
extern  BYTE                _fmSLRRReg          [];
extern  BYTE                _fmFNUMReg          [];
extern  BYTE                _fmKONBLOCKReg      [];
extern  BYTE                _fmFBCReg           [];
extern  BYTE                _fmWSReg            [];
extern  BYTE                _fmInstrumentSet    [];
extern  BYTE                _fmNoteOn           [];
extern  BYTE                _fmPercussionRegister;
extern  BYTE                _fmMelodicSlots     [ _MEL_VOICES ][ 2 ];
extern  BYTE                _fmPercussiveSlots  [ _PERC_VOICES ][ 2 ];
extern  WORD                _fmMidiScale        [];
extern  _FM_GLOBAL_DATA     _fmGlobalParameters;

// Error definitions.
#undef  _ERROR_NONE
#define _ERROR_NONE     0

// NOTE : Starting offset for all FM library errors is 7000.
enum
{
    _ERROR_INVALID_PORT = 7000,         // Invalid FM port sent to fmInit
    _ERROR_SYSTEM_NOT_INITIALIZED,      // System has not been initialized.
    _ERROR_INVALID_VOICE_FOR_MODE,      // An invalid voice was specified.
    _ERROR_INVALID_INSTRUMENT,          // An invalid instrument was specified.
    _ERROR_INSTRUMENT_NOT_DEFINED,      // Specified instrument was not defined.
    _ERROR_VOICE_NOT_ACTIVATED,         // Specified voice not turned on.
    _ERROR_INVALID_VOICE,               // Specified voice does not exist.
    _ERROR_VOICE_OUT_OF_RANGE           // Voice out of range for current mode.
```

```
};

// Function prototypes.
extern VOID fmSetRegister ( BYTE, BYTE );

WORD   fmInit              ( _FM_INIT_DATA * );
WORD   fmUninit            ( VOID );
WORD   fmSetMode           ( BYTE );
WORD   fmSetInstrument     ( _BNK_INSTRUMENT far *, BYTE );
WORD   fmReset             ( VOID );
WORD   fmNoteOn            ( BYTE, WORD );
WORD   fmNoteOff           ( BYTE );
WORD   fmSetFrequency      ( BYTE, WORD );

#endif
//*****************************************************************************
// end      CVXFMSY.H
//*****************************************************************************
```

CVXMIDI.H

The following must be included in your program when using any of the Covox MIDI functions.

```
//*****************************************************************************
// File              : CVXMIDI.H
// Date              : January 10, 1992
// Description       : Header file for the MIDI libraries.
//
// Last update       :
// Additional Notes  :
//*****************************************************************************
//        Copyright (c) 1992,  Covox, Inc.  All Rights Reserved
//*****************************************************************************

#ifndef _CVXMIDI_DEFINED
#define _CVXMIDI_DEFINED

#include "cvxdefs.h"

//*****************************************************************************
//  If compiled with Borland C, the header file CVXBRLND.H will be
//  automatically included.
//*****************************************************************************
#if defined(__TURBOC__)
#include "cvxbrlnd.h"
#endif

#define _MIDI_PORT_A  0x330     // The valid MIDI ports on the MIDI Maestro and
#define _MIDI_PORT_B  0x338     // Sound Master II.

#define _IRQ_2          2       // The available IRQ channels.
#define _IRQ_3          3       //
#define _IRQ_5          5       // (only on Soumd Master II)
#define _IRQ_7          7       // (only on Sound Master II)

#endif
//*****************************************************************************
// end      CVXMIDI.H
//*****************************************************************************
```

CVXUTIL.H

The following must be included in your program when using any of the Covox Utility functions.

```
//*****************************************************************************
// File              : CVXUTIL.H
// Date              : January 10, 1992
// Description       : Covox utilities support header file.
//
// Last update       :
// Additional Notes  :
//*****************************************************************************
//          Copyright (c) 1992, Covox, Inc. All Rights Reserved
//*****************************************************************************

#ifndef _CVXUTIL_DEFINED
#define _CVXUTIL_DEFINED

#include "cvxdefs.h"

//*****************************************************************************
//  If compiled with Borland C, the header file CVXBRLND.H will be
//  automatically included.
//*****************************************************************************
#if defined(__TURBOC__)
#include "cvxbrlnd.h"
#endif

//*****************************************************************************
// Constants for cvxFileOpen and cvxFileCreate
//*****************************************************************************
#define _OPEN_R_ONLY        0x80
#define _OPEN_W_ONLY        0x84
#define _OPEN_RW            0x82
#define _CREATE_R_ONLY      0x0001
#define _CREATE_NORMAL      0x0000

//*****************************************************************************
// Error Codes for covox Utilities
//*****************************************************************************
#undef  _ERROR_NONE
#define _ERROR_NONE     0

// Error codes for Covox utilities
// NOTE : Starting offset for all Covox utilities errors is 9000.
enum
{
    _ERROR_NOT_ENOUGH_MEMORY = 9000,
```

```
    _ERROR_MEMORY_ALLOC,            // Error during memory allocation.
    _ERROR_MEMORY_DEALLOC,          // Error during memory deallocation.
    _ERROR_INVALID_HANDLE,          // File not open.
    _ERROR_FILE_WRITE_FAILED,       // Unknown file write failure.
    _ERROR_FILE_READ_FAILED,        // Unknown file read failure.
    _ERROR_FILE_NOT_FOUND,          // Non-existent file given in cvxFileOpen.
    _ERROR_PATH_NOT_FOUND,          // Invalid path given.
    _ERROR_TOO_MANY_FILES_OPEN,     // No more handles available in system.
    _ERROR_ACCESS_DENIED            // File could not be accessed.
    _ERROR_INVALID_ACCESS           // Invalid mode during open/create.
};

//***************************************************************************
// Prototypes for all Covox utility functions.
//***************************************************************************
LPSTR  cvxBufferAlloc     ( LONG, LONG * );
WORD   cvxBufferFree      ( LPSTR );
WORD   cvxFileRead        ( HANDLE, LPSTR, LONG, LONG * );
WORD   cvxFileWrite       ( HANDLE, LPSTR, LONG, LONG * );
WORD   cvxFileClose       ( HANDLE );
WORD   cvxFileOpen        ( LPSTR, BYTE, HANDLE * );
WORD   cvxFileCreate      ( LPSTR, WORD, HANDLE * );
WORD   cvxRateToHz        ( BYTE );
BYTE   cvxHzToRate        ( WORD );

#endif
//***************************************************************************
// end      CVXUTIL.H
//***************************************************************************
```

CVXBRLND.H

The following header file is automatically included by any of the header files described previously in this appendix, when compiling a program in Borland C.

```
//***************************************************************************
// File              : CVXBRLND.H
// Date              : January 10, 1992
// Description       : Redefinition of Borland C interface functions.
//
// Last update       :
// Additional Notes  : The following defines are used to bridge the differences
//                     between those Microsoft C 6.0 and Borland C 2.0 functions
//                     which are analogous but use different function names.
//***************************************************************************
//          Copyright (c) 1992, Covox, Inc. All Rights Reserved
//***************************************************************************

#ifndef _CVXBRLND_DEFINED
#define _CVXBRLND_DEFINED

#include "cvxdefs.h"

// DOS related functions
//
#define _dos_setvect( vect, func )   setvect( vect, func )
#define _dos_getvect( vect )         getvect( vect )
#define _enable()                    enable()
#define _disable()                   disable()
#define _chain_intr( intr )          (*intr)()

#define _asm                         asm

#endif
//***************************************************************************
// end       CVXBRLND.H
//***************************************************************************
```

CVXDEFS.H

CVXDEFS.H is automatically included when using any of the other include files accompanying the Covox libraries.

```
//*****************************************************************************
// File             : CVXDEFS.H
// Date             : January 10, 1992
// Description      : Covox specific defines.
//
// Last update      :
// Additional Notes :
//*****************************************************************************
//        Copyright (c) 1992, Covox, Inc. All Rights Reserved
//*****************************************************************************

#ifndef _CVXDEFS_DEFINED
#define _CVXDEFS_DEFINED

#undef  _TRUE
#undef  _FALSE
#undef  _NULL
enum
{
    _FALSE,
    _TRUE
};

#define _NULL     0

typedef void                    VOID;
typedef unsigned                BOOL;
typedef unsigned char           BYTE;
typedef unsigned                WORD;
typedef unsigned long           LONG;
typedef unsigned long           DWORD;

typedef BYTE            *       PBYTE;
typedef BYTE            *       PSTR;
typedef WORD            *       PWORD;
typedef LONG            *       PLONG;
typedef VOID            *       PVOID;

typedef BYTE    far     *       LPBYTE;
typedef BYTE    far     *       LPSTR;
typedef WORD    far     *       LPWORD;
typedef LONG    far     *       LPLONG;
typedef VOID    far     *       LPVOID;
```

```
typedef BYTE      huge  *     HPBYTE;
typedef BYTE      huge  *     HPSTR;
typedef WORD      huge  *     HPWORD;
typedef LONG      huge  *     HPLONG;
typedef VOID      huge  *     HPVOID;

typedef WORD                  HANDLE;

#endif
//*****************************************************************************
// end      CVXDEFS.H
//*****************************************************************************
```

APPENDIX B

Covox 16-Byte Sound File Header

A 16-byte header is added to the beginning of all files recorded with the Covox Developer Toolkit functions. This header has the following format (All values in hexadecimal):

FF 55 FF AA FF 55 FF AA xx yy 00 00 00 00 00 00

The first 8 bytes comprise a unique code that is guaranteed not to show up inside the rest of the file. This means that multiple sound files can be concatenated together and the locations found by looking for the headers.

xx - A one byte value that specifies the type of encoding done. It must be one of the following values:

0x08 - **_FORMAT_V8** - 8 bit PCM
0x82 - **_FORMAT_V2S** - 2 bit ADPCM with or without silence encoding
0x83 - **_FORMAT_V3S** - 3 bit ADPCM with or without silence encoding
0x84 - **_FORMAT_V4S** - 4 bit ADPCM with or without silence encoding
0x88 - **_FORMAT_V8S** - 8 bit PCM with or without silence encoding

yy - A one byte (*samplingRate*) value that tells the rate at which a file was recorded. This value also applies to *recordRate*, *playbackRate* and **CVXRATE** values.

To calculate the sampling rate in samples per second, use the following equation (where F represents the desired sample rate (frequency) and R represents the *samplingRate* value (in decimal):

F = 1,193,180 / (256 - R)

To convert from hertz to the corresponding *samplingRate* value (in decimal), use the following formula:

R = 256 - (1,193,180 / F)

00 - These bytes are reserved for future use.

APPENDIX C

Covox Hardware Devices

Part 1 of this appendix lists the physical port addresses and offsets of all Covox devices.

Part 2 of this appendix lists the logical port address, IRQ, and DMA Channel values that are used in the Covox library functions.

Part 3 lists the I/O capabilities of the various Covox devices.

Part 4 includes diagrams of the internal Covox devices.

Physical Ports

Digitizer Base Ports (Voice Master Key and Sound Master II)

The Digitizer Port jumper (0 to 3) selected on the Voice Master Key or Sound Master II card indicates the Base Port of the digitizer:

0 - 0x0220 (factory default),
1 - 0x0240,
2 - 0x0280,
3 - 0x02C0.

Digitizer Base Port Offsets (Voice Master Key and Sound Master II)

The Base Port value + the Base Port Offset value can be used to access the ports on the Voice Master. As can be seen below, only port offsets from 0x08 and 0x0F are used on the Voice Master and Sound Master II cards.

8254 TIMER 0 OFFSET	= 0x08	*Timer/Clock*
8254 TIMER 1 OFFSET	= 0x09	*Timer/Clock*
8254 TIMER 2 OFFSET	= 0x0A	*Timer/Clock*
8254 CONTROL OFFSET	= 0x0B	*Timer/Clock Control Register*

DMA Control Port Offsets (Voice Master Key and Sound Master II)

CLEAR IRQ OFFSET = 0x0C

The Clear IRQ Offset is to be utilized after every DMA terminal count is reached. It is common to find this port being written to inside an interrupt handler. Any value may be output to the port.

DISABLE DMA OFFSET = 0x0D

Writing to this port will disable DREQ (DMA Request) channel 1 or DREQ channel 3, thus preventing all DMA I/O.

ENABLE DMA OFFSET = 0x0E

Writing to this port should be done last and only after all clock and DMA setup has been completed. A write to this port will fire off a DREQ (cause the DREQ line to go low), thus allowing the DMA I/O to begin.

DAC OFFSET = 0x0F

The DAC OFFSET port should not be written to during DMA I/O.

FM Synthesizer Base Ports (Sound Master II)

0x0388 (factory default),
0x0380

MIDI Base Ports (MIDI Maestro and Sound Master II)

0x0330 (factory default),
0x0338

External Devices (Voice Master Key System II and Speech Thing)

The Speech Thing and Voice Master II can be connected to one of 3 LPT ports available on the PC. The actual port address values for LPT1, LPT2, and LPT3 are stored at the following memory locations (segment:offset) on the PC:

LPT1	40 : 08
LPT2	40 : 0A
LPT3	40 : 0C

Logical Port, IRQ, and DMA Channel Values

Logical DAC Port Values

_CVX_VM0 -	Voice Master Internal or Sound Master II DAC set to port 22X (factory default).
_CVX_VM1 -	VM (internal) or SM II DAC set to port 24X.
_CVX_VM2 -	VM (internal) or SM II DAC set to port 28X.
_CVX_VM3 -	VM (internal) or SM II DAC set to port 2CX.
_CVX_LPT1 -	VM System II (external) or Speech Thing DAC attached to LPT1.
_CVX_LPT2 -	VM System II (external) or Speech Thing DAC attached to LPT2.
_CVX_LPT3 -	VM System II (external) or Speech Thing DAC attached to LPT3.
_CVX_ISP -	The internal PC speaker.

Logical FM Port Values

_FM_PORT_A -	Sound Master II FM chip set to port 388.
_FM_PORT_B -	Sound Master II FM chip set to port 380.

Logical MIDI Port Values

_MIDI_PORT_A -	SM II MIDI or MIDI Maestro set to port 330.
_MIDI_PORT_B -	SM II MIDI or MIDI Maestro set to port 338.

Logical IRQ Values

_IRQ_2 -	IRQ 2 card jumper setting.
_IRQ_3 -	IRQ 3 card jumper setting.
_IRQ_4 -	IRQ 4 card jumper setting.
_IRQ_5 -	IRQ 5 card jumper setting.
_IRQ_7 -	IRQ 7 card jumper setting.

Logical DMA Channel Values

_DMA_CHANNEL_1 -	DMA Channel 1 card jumper setting.
_DMA_CHANNEL_3 -	DMA Channel 3 card jumper setting.

Hardware Capabilities

Some features of the Covox Developer's Toolkit may be used with some Covox hardware devices, but not with others. For each of the first 8 chapters of this manual, the applicable Covox devices are listed directly below.

CHAPTER 1: Non-DMA Record and Play Functions

1. Sound Master II
2. Internal Voice Master
3. External Voice Master System II
4. Speech Thing *(playback only)*

CHAPTER 2: Pack and Unpack Functions — Device Independent

CHAPTER 3: DMA Record and Play Functions

1. Sound Master II
2. Internal Voice Master

CHAPTER 4: TSRPLAY Record and Play Functions

1. Sound Master II
2. Internal Voice Master

CHAPTER 5: Interrupt 0x1A BIOS Record and Play Functions

1. Sound Master II
2. Internal Voice Master
3. External Voice Master System II *(non-DMA only)*
4. Speech Thing *(non-DMA playback only)*

CHAPTER 6: FoxPro/dBase Record and Play Modules

1. Sound Master II
2. Internal Voice Master
3. External Voice Master System II
4. Speech Thing *(non-DMA playback only)*

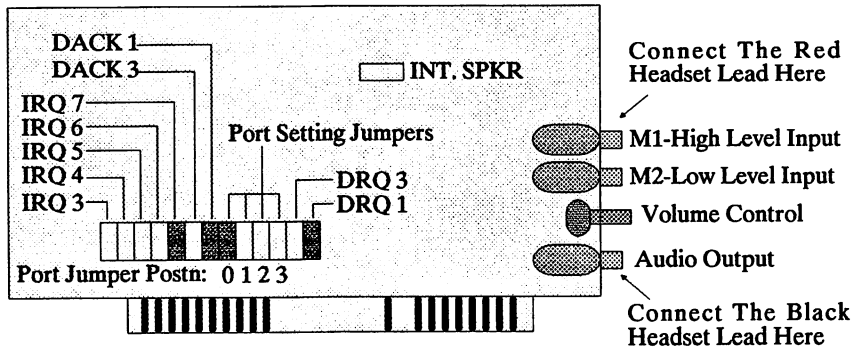
CHAPTER 7: FM Synthesizer Functions

1. Sound Master II

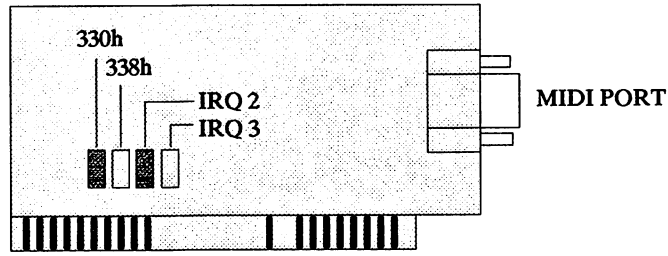
CHAPTER 8: MIDI Recording and Playback Devices

1. Sound Master II
2. MIDI Maestro

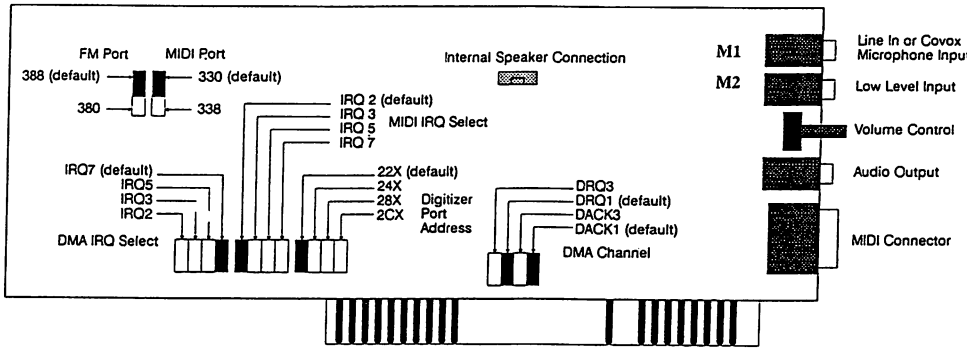
Voice Master Key Internal Card:



MIDI Maestro Internal Card:



Sound Master II card:



APPENDIX D

Keyboard Interrupt Handler

The program **HOOK_KB.C** contains a keyboard interrupt handler that sets the variable **_ioStopFlag** equal to **_TRUE** when a keypress event is detected from the keyboard. This variable is used in all the non-DMA play and record functions (eg., **play8**, **record3s**, etc.) to terminate the recording or playback process.

Before an I/O function is used, **hookKeyboard()** must be called to install the keyboard interrupt handler. Call **unhookKeyboard()** after the sound I/O function has terminated to restore the original interrupt handler. The object module **HOOK_KB.OBJ** must be linked with any application requiring this feature.

This function can be replaced with a similar function to set **_ioStopFlag**, terminating I/O without utilizing the keyboard interrupt handler. See the following source code file for specific help on trapping the keyboard interrupt.

```
// File        : HOOK_KB.C
//
// Version     : 1/10/92
//
// Description : This module contains three functions, which are
//               designed to hook and unhook into the keyboard
//               BIOS vector address. The function keyboardHandler
//               will determine if a keypress has been made. If it
//               has, the flag _ioStopFlag will be set to _TRUE.
//*************************************************************
//*    Copyright (c) 1992,  Covox, Inc.  All Rights Reserved    *
//*************************************************************

#include <dos.h>
#include "cvxdefs.h"

//*************************************************************
//   If Borland C is used to compile the library the interface
//   macros header  must be included.
//*************************************************************
#if defined(__TURBOC__)
```

```
#include "cvxbrlnd.h"
#endif

#define _KEYBOARD_INT 0x09

// This variable will contain the original address of the keyboard
// BIOS routine.
VOID ( interrupt far * old_keyboard_vect )( );

// This function is the new keyboard interrupt handler routine.
VOID interrupt far keyboardHandler( VOID );

// This flag is defined in the play and record functions.  It will
// be used throughout all of the record and play functions to
// determine if I/O should be terminated.
extern BYTE _ioStopFlag;

// Keyboard buffer head and tail pointer.
WORD head, tail;

// This function will save the original vector address of the
// keyboard BIOS routine and set up a new address to point to
// the function keyboardHandler.
VOID hookKeyboard( VOID )
{
   // Save off original keyboard vector address.
   old_keyboard_vect = _dos_getvect( _KEYBOARD_INT );

   // Hook int 9h (keyboard interrupt). This hook will set the
   // flag _ioStopFlag when a key has been struck.
   _asm   cli
   _dos_setvect( _KEYBOARD_INT, keyboardHandler );
   _asm   sti

   _asm  mov   _ioStopFlag, 0
}


// This function will return the vector address for the keyboard
// back to its original state.
VOID unhookKeyboard( VOID )
{
   // reset original keyboard vector.
   _asm   cli
   _dos_setvect( _KEYBOARD_INT, old_keyboard_vect );
   _asm   sti
}

// New keyboard handler. This handler will set a flag to indicate
// that a keyboard hit has occurred.
VOID interrupt far keyboardHandler( VOID )
{
   // Call original keyboard handler
   ( *old_keyboard_vect )();

   _asm  xor   ax, ax
   _asm  mov   es, ax

   // Check head and tail pointers to see if they are equal.
   _asm   mov   ax, WORD PTR es:0x41A
   _asm   mov   head, ax
   _asm   mov   ax, WORD PTR es:0x41C
```

```
    _asm    mov    tail, ax

    if( head != tail )
       _asm mov    _ioStopFlag, 1

    _asm    mov    al, 20H
    _asm    mov    20H, al
}
```

APPENDIX E

TSR DMA Sound File Playback Routine

The program **TSRPLAY.EXE** was designed to play back pre-recorded digitized sound and speech as a background task. It is ideally suited for adding sound to presentation programs and other multimedia applications.

General Description

Once **TSRPLAY.EXE** is loaded into memory, it will stay resident until it is explicitly removed with the '/Q' option.

TSRPLAY.EXE will accept up to 35 files for playback. If no file extension is specified with the file then 8 bit (.V8), is the assumed default. Each file can be given a rate or the rate will be read from the header. If the file does not have a Covox 16 byte header and no rate is specified on the command line then it will be assumed that the file is 8 bit and was recorded at the default rate of 132 (i.e., 9622 KHz).

Each file that is specified on the command line will be played back once and **TSRPLAY.EXE** will then 'sleep' until another file or group of files is specified. If the '/N' option is specified after a file then the file can be played back more than once.If the '/N-1' option is used the file will be played back indefinitely or until the program is removed from memory using the '/Q' option or another group of files is loaded. At any time during play back the program can be removed from memory using the '/Q' option.

The command line options '/K', '/B', '/J', '/I', and '/C' are only utilized during the first call to **TSRPLAY.EXE**. If these options are used after **TSRPLAY.EXE** has been placed into resident memory, they will simply be ignored.

The number of memory buffers and their size used by **TSRPLAY.EXE** can be controlled with the command line options '/K' and '/B. These buffers contain the sound data processed by DMA during play back. If the size of each buffer is too small, or if there are not enough buffers, then the playback may pause while a large program is loading. On the other hand, if too much memory is used up by **TSRPLAY.EXE** then there may not be enough available memory for some applications to load up. The default values should work with most 286 or 386 hard drive systems.

TSRPLAY.EXE utilizes DMA (direct memory access), for playing back sound files. When **TSRPLAY.EXE** is first loaded into memory an auto-detection function is run to determine what settings (IRQ, port, DREQ, and DACK), your Voice Master or Sound Master II card is using. This auto-detection can sometimes cause your computer to lock or not play back sound files correctly. If this is the case then you will need to use the command line options '/J', '/I', and '/C' to tell **TSRPLAY.EXE** not to perform the auto-detection.

Running TSRPLAY.EXE

Syntax: **TSRPLAY file1 [file2...file35] [OPTIONS]**

Options:

/Rx -	Rate in hertz to play back file (4679-44191). 9622 is the default. Note how this differs from **SAY.EXE**.
/Bx -	Number of playback buffers (2-4). 3 is the default.
/Kx -	Size in kilobytes of each of the buffers (1-64). 32 is the default.
/Nx -	Number of times to play file. 1 is default. -1 for infinite. 255 is maximum.
/Z-	Pause output. ('Z' for sleep)
/W-	Restart paused output. ('W' for wake)
/F-	Flush all files being played back.
/Q-	Remove **TSRPLAY.EXE** from memory.
/S-	Install/Uninstall quietly (no screen output).
/?-	List the available options.

The following Voice Master and Sound Master II DMA settings can be selected in case the auto-detect algorithm returns errors:

/Jx -	Select the logical port value determined by the jumper setting (1-4) on the card. -1 (default) for an automatic sequential search.
/Cx -	Select DMA channel (1 or 3). -1 (default) checks 1 then 3.
/Ix -	Select IRQ (2,3,4,5 or 7). -1 (default) for sequential search.

Examples

- Install TSRPLAY.EXE to play back 2 sound files.

C:\ TSRPLAY songs.v8 voice.v3s

- Install TSRPLAY.EXE to play back 1 sound file. Set the buffer size to 16 kilobytes.

C:\ TSRPLAY C:\songs.v8 /K16

- Install TSRPLAY.EXE to play back 1 sound file. Set the number of buffers to 4.

C:\ TSRPLAY songs.v8 /B4

- Install TSRPLAY.EXE to play back 2 sound files. The first file is played 3 times and the second is played once.

C:\ TSRPLAY C:\f1.v8 /N3 C:\f2.v3s

- Install TSRPLAY.EXE to play back 3 sound files. Each file is specified with a playback rate.

C:\ TSRPLAY f1.v8 /R8000 f2.v3s /R10000 f3.v4s /R12000

Using TSRPLAY With Presentation Software

Most presentation programs have a script which loads and controls the graphic screens. To add sound while a graphic is being loaded or animated, you need to be able to invoke a DOS command from the script. Most presentation programs have this capability. The strategy is simple: call **TSRPLAY** when you want to start playing sound. Use the /S option so that **TSRPLAY** doesn't print information to the screen during the presentation.

For example, when using Show Partner F/X, call TSRPLAY in the 'Picture Comment' window and set the 'Effect' to 'Command'. See page 7-20 for usage of Commands to affect flow control.You might have to specify the path to **TSRPLAY** as part of the command line syntax.

With GRASP, use the following line in the script:

```
EXEC TSRPLAY.EXE filename /S
```

Make sure **TSRPLAY** is in the same directory and/or path of GRASP, otherwise you can specify the pathname(s). Also note that you must specify the '.EXE' extension.

APPENDIX F

Covox Library Changes

From Version 3.x to Version 4.0

Version 3.x function name	Version 4.0 function prototype
calc_length2s	N/A
calc_length3s	N/A
calc_length4s	N/A
get_byte	BYTE **recordByte**(VOID);
init_play	WORD **playByteInit**(WORD, BYTE, BOOL, WORD);
init_record	WORD **recordByteInit**(WORD, BYTE, BOOL);
put_byte	VOID **playByte**(BYTE);
pack_first	WORD **packFirst**(_CONVERT_DATA far *);
pack_next	WORD **packNext**(_CONVERT_DATA far *);
record2s	WORD **record2s**(LPSTR, LONG, BYTE, WORD, WORD, LONG *);
record3s	WORD **record3s**(LPSTR, LONG, BYTE, WORD, WORD, LONG *);
record4s	WORD **record4s**(LPSTR, LONG, BYTE, WORD, WORD, LONG *);
record8s	WORD **record8s**(LPSTR, LONG, BYTE, WORD, WORD, LONG *);
record8	WORD **record8**(LPSTR, LONG, BYTE, WORD, LONG *);

Version 3.x function name	Version 4.0 function prototype
SD_DMA_Init	WORD **dmaInit**(WORD, WORD, WORD, WORD *);
SD_DMA_UnInit	WORD **dmaUninit**(VOID);
SD_DMA_Record	WORD **dmaRecord**(LPSTR, LONG, BYTE);
SD_DMA_Say	WORD **dmaPlay**(LPSTR, LONG, BYTE, WORD);
SD_Number_In_Queue	WORD **dmaNumberInQueue**(VOID);
SD_DMA_Bytes_Remaining	WORD **dmaBytesRemaining**(VOID);
SD_Flush_Queue	WORD **dmaFlushQueue**(VOID);
SD_absolute_address	N/A
SD_DMA_Disable	N/A (See **dmaPause**)
SD_DMA_Enable	N/A (See **dmaUnpause**)
SD_DMA_get_byte	N/A
SD_DMA_Init_get_byte	N/A
SD_DMA_Init_put_byte	N/A
SD_DMA_put_byte	N/A
SD_DMA_UnInit_get_byte	N/A
SD_DMA_UnInit_put_byte	N/A
SD_FindVMDMA	WORD **dmaPortDetect**(WORD);
SD_Get_Device	N/A
SD_Get_Device_Jumper	WORD **dmaGetPort**(WORD *);
SD_Get_DMA_Channel	WORD **dmaGetChannel**(WORD *);
SD_Get_IRQ_Number	WORD **dmaGetIRQNumber**(WORD *);
SD_IRQ_Reset	N/A
SD_Set_DMA_Rate	WORD **dmaSetRate**(BYTE);
say	WORD **playAny**(LPSTR, LONG, BYTE, WORD, BOOL, BOOL);
say2s	WORD **play2s**(LPSTR, LONG, BYTE, WORD, BOOL, BOOL);
say3s	WORD **play3s**(LPSTR, LONG, BYTE, WORD, BOOL, BOOL);

Version 3.x function name	Version 4.0 function prototype
say4s	WORD **play4s**(LPSTR, LONG, BYTE, WORD, BOOL, BOOL);
say8s	WORD **play8s**(LPSTR, LONG, BYTE, WORD, BOOL, BOOL);
say8	WORD **play8**(LPSTR, LONG, BYTE, WORD, BOOL);
say_in_background	N/A
stop_say_background	N/A
uninit_play	WORD **playUninit**(VOID);
uninit_record	WORD **recordUninit**(VOID);
unpack2s	N/A (See **unpack_first** and **unpack_next**)
unpack3s	N/A (See **unpack_first** and **unpack_next**)
unpack4s	N/A (See **unpack_first** and **unpack_next**)
unpack8s	N/A (See **unpack_first** and **unpack_next**)
unpack_first	WORD **unpackFirst**(_CONVERT_DATA far *);
unpack_next	WORD **unpackNext**(_CONVERT_DATA far *);

N/A — *These functions have been removed from the Covox libraries.*

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

Y

COVOX LICENSE AGREEMENT

PLEASE READ CAREFULLY BEFORE OPENING.

This is a legal agreement between you, the end user, and Covox, Incorporated. By opening this sealed disk package, you are agreeing to be bound by the terms of this agreement. If you do not agree to the terms of this agreement, promptly return the unopened disk package and the accompanying items (including written materials, binders, and hardware) to the place you obtained them for a refund.

COVOX SOFTWARE LICENSE

1. GRANT OF LICENSE. Covox grants to you the right to use one copy of the enclosed software program (the "SOFTWARE") on a single terminal connected to a single computer (i.e. with a single CPU). You may not network the SOFTWARE or otherwise use it on more than one computer or computer terminal at the same time.

2. COPYRIGHT, PATENTS, & TRADEMARKS. The SOFTWARE is owned by Covox, Inc. or its suppliers and is protected by the United States copyright laws and international treaty provisions. Therefore, you must treat the SOFTWARE like any other copyrighted material (e.g. a book or musical recording) except that you may either (a) make a copy of the SOFTWARE solely for backup or archival purposes, or (b) transfer the SOFTWARE to a single hard disk provided you keep the original solely for backup or archival purposes. You may not copy the written material accompanying the software. The voice recognition, data compression routines, and other routines that might be present in this disk package are subject to existing patents and patents pending. The names COVOX, VOICE MASTER, SPEECH THING, SOUND MASTER, VOICE HARP, and the Covox logo are registered trademarks of Covox, Inc.

3. OTHER RESTRICTIONS. You may not reverse engineer, decompile, or disassemble the compiled SOFTWARE.

4. DEVELOPMENT SOFTWARE. The compiled library routines supplied by Covox may be used in application software on a royalty-free basis provided that you; (a) distribute the compiled and linked library routines only in conjunction with and as a part of your software product; (b) the sign-on message for your software product must display that the sound and/or speech software is the copyright of Covox, Inc; (c) that the owners manual for your software product must state in a conspicuous location that the sound and/or speech software is licensed from Covox, Inc., that Covox hardware products are supported, and where Covox products might be obtained; (d) that if the SOFTWARE is used to support non-Covox sound hardware, Covox hardware must also be supported by your software product, and; (e) agree to indemnify, hold harmless, and defend Covox from and against any claims or lawsuits, including attorney fees, that arise or result from the use or distribution of our software product.

5. EXCEPTIONS. No part of the "SmoothTalker" text-to-speech software (SPEECHV2.EXE, SPEECHV3.EXE, STALK.EXE, or STDRIVER.SYS) nor ANY voice recognition software may be incorporated within your software product.

LIMITED WARRANTY

Covox, Inc. guarantees the software disks to be free of manufacturing or duplication defects for a period of one year from the date of purchase. Covox, Inc., will replace the diskette free of charge, provided it is returned to the factory via prepaid transportation.

LIABILITY DISCLAIMER

Software is provided on an "as is" basis. Covox, Inc. disclaims liability for direct, indirect or incidental damages arising from the use of this software, including but not limited to the interruption of service, loss of business or potential profits, legal actions or other consequential damages even if Covox has been advised of the possibility of such damages.

Control of computers or environmental factors by means of voice could expose the user to some risk. Automatic voice recognition by machine remains an unreliable technology due to uncontrollable variations in the way that normal speech is produced in an uncertain acoustic environment. Covox, Inc., specifically disclaims liability as stated in the preceding paragraph when applied to voice recognition.

Should you have any questions regarding this Agreement, please contact Covox, Inc., 675 Conger St., Eugene, OR 97402, U.S.A. Tel (503) 342-1271, Fax (503) 342-1283

ATTENTION BORLAND DEVELOPERS!

February 7, 1992

The TSRPLAY library functions described in chapter 4 do not yet work properly when compiled with Borland C. As a result, the TSRPLAY functions are still in beta for the library SDIGIBC.LIB.

At this time, to get these functions to operate properly, TLINK must be specified with a '/v' switch. Updates correcting this problem will be sent out as soon as they are available.

COVOX DEVELOPER'S TOOLKIT REGISTRATION CARD

Please fill out and mail back to us **WITHIN 30 DAYS** of purchase. This will qualify you for **software update notices, technical support**, and **developer's status** on our BBS. Thank you!

NAME: _____ PH. # _____

ADDRESS: _____ FAX # _____

_____

CITY/STATE (COUNTRY) _____ ZIP (POSTAL CODE) _____

Purchased From: _____ Date Purchased: _____

Where did you see the product advertised? _____

Notice: If you need a different software media, Covox will send it to you at no charge provided this warranty card and the original disks are returned postage-paid within 30 days from date of purchase. Check the box (one only) for the format you need: ☐ **360K 5.25" fomat** ☐ **720K 3.5"**

Programming Languages

Compiler **Version**

_____

_____

_____

Linkers Used: _____

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 147 EUGENE, OREGON

POSTAGE WILL BE PAID BY ADDRESSEE

COVOX INC
675 CONGER ST STE E
EUGENE OR 97402-9923