Pro AudioSpectrum Developer's Toolkit Reference

Media Vision

Pro AudioSpectrum Developer's Toolkit Reference

Media Vision, Inc.

47221 Fremont Boulevard

Fremont, CA 94538

Technical Support (510) 770-9905

BBS (510) 770-0968

Fax (510) 770-8648

July 1992

Trademarks and Copyrights

(7) Some states do not allow the exclusion of limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Media Vision, Inc.

47221 Fremont Boulevard

Fremont, CA 94538

Technical Support (510) 770-9905

BBS (510) 770-0968

FAX (510) 770-8648

Printed in the United States of America.

This book was written, illustrated, and produced using FrameMaker® workstation publishing software and the ITC Times®, Helvetica® Condensed, Palatino® and Courier® families of typefaces.

Program team: Doug Cody, Bart Crane, Jim Gifford, and Debbie Gronski. Technical writing by Tansy Associates.

Table of Contents

Appendices

List of Tables

List of Figures

1 *Introduction*

The Pro AudioSpectrum Developer's Toolkit provides a fast and easy method for PC developers to write software for Media Vision's Pro AudioSpectrum family of multi-media products.

This chapter includes:

- A description of the contents of the Pro AudioSpectrum Developer's Toolkit
- A description of how this manual is organized
- An overview of Pro AudioSpectrum software and hardware API's
- Technical support information

Pro AudioSpectrum Developer's Toolkit contents

Included with the Pro AudioSpectrum Developer's Toolkit is a set of software and hardware API's. The API's consist of source code, header files, useful utilities, and sound clip files, plus supporting documentation.

To see the directory structure of the Developer's Kit diskettes and descriptions of the files on the diskette, see the `CONTENTS.LST` file on the first Developer's Kit diskette.

About this manual

This book, combined with the `READ.ME` files contained on the first diskette, comprises a comprehensive reference to Pro AudioSpectrum development techniques and API's.

The manual is organized as follows:

General information (Chapters 1-3)

Chapter 1, "Introduction," provides a brief descriptions of the Toolkit and its major features. Chapter 2, "Installing the Pro AudioSpectrum Developer's Toolkit," provides installation instructions. Chapter 3, "Common Function Calls and Hardware Registers," describes the function calls and hardware register functions used by more than one Pro AudioSpectrum feature.

Multiple sections, organized by feature

A separate section of the manual is devoted to each major product feature. These sections are divided into several chapters which typically provide the following information:

- Theory of operation for the feature
- Step-by-step programming procedures
- Background material that helps you use the function call interface(s) and hardware register functions more efficiently
- One or more software software function call references
- A hardware register function reference

Appendices

The appendices cover general PC programming techniques, charts and tables that are used with function calls and register functions, and other related information.

Manual conventions

File names, function call names, and function call parameters are presented in `Courier` font within the text. "C" language function calls are presented inaccordance with ANSI standards with the types listed next to parameters. For example, `int cdstatus (int ccdrive)`.

Software and hardware API's

The Developer's Toolkit provides maximum flexibility and control over Pro AudioSpectrum features. Each feature is accessible by at least one software API and one direct hardware API.

PCM digital audio sampler

The Pro AudioSpectrum family of products uses 8- and 16-bit, pulse code modulation (PCM) sampling to record and play back digital audio stereo sounds. Pro AudioSpectrum supports recording and playback rates from 4 Hz, and 44.1kHz. At each sampling rate, a variable input filter is programmed at the correct Nyquist frequency to prevent imaging, aliasing, and other undesirable digital effects.

The Pro AudioSpectrum Developer's Toolkit accompanying this manual provides three interfaces to the PCM sampler:

- High-level source code

A set of "C" language function calls which are contained in the `PCMIO.H` file. This is a file- and block-oriented API offering the highest level of abstraction from the PCM hardware. You can find information on this interface in Chapter 5, "High-Level PCM Function Call Reference."

- Low-level source code

  A set of "C" language function calls which have been prototyped in the `MVSOUND.H` file. You can find information on this interface in Chapter 6, "Low-Level PCM Function Call Reference."

- Hardware register interface

  The lowest level interface upon which the software function call interfaces were written. You can find information on this interface in Chapter 7, "PCM Hardware Register Functions."

Stereo FM synthesizer

Multi-timbral, polyphonic, 11-voice FM synthesizers function together to produce 22 different instruments and can play up to 22 different notes simultaneously. The FM synthesizers are completely AdLib compatible.

Two API's are available for the stereo FM synthesizer:

- Low-level source code

  A set of "C" language function calls which are contained in the `3812A.ASM` file. You can find information on this interface in Chapter 9, "Low-Level FM Synthesizer Function Call Reference."

- Hardware register interface

  The lowest level interface upon which the software function call interface was written. You can find information on this interface in Chapter 11, "Standard FM Synthesizer Register Functions."

  The latest versions of the Pro AudioSpectrum come with the OPL3 stereo FM synthesizer chip set. You can find information on the additional register functions offered by this chip in Chapter 12, "Enhanced FM Synthesizer Register Functions."

MIDI sequencer

The MIDI controller provides high speed bi-directional transmission with two internal FIFO buffers of 16 bytes each.

Two API's are available for the MIDI sequencer:

- Low-level source code

A set of "C" language function calls which are contained in the `MIDIA.ASM` file. You can find information on this interface in Chapter 13, "MIDI Programming Essentials."

- Hardware interface

  The lowest level interface upon which the software function call interface was written. You can find information on this interface in Chapter 15, "MIDI Hardware Register Functions."

CD-ROM

The CD-ROM feature provides SCSI driven I/O to CDROM's, tapes, and hard drives and controls and plays back music CD's.

Three software API's are available for the CD-ROM that work directly with the CD-ROM device driver and Microsoft CD-ROM Extensions:

- High-level source code that works with the CD-ROM device driver

  A set of "C" language function calls which have been prototyped in the `CDMASTER.H` file. This API is currently under revision and will be upgraded to offer more functions and greater abstraction from the CD-ROM hardware. You can find information on this interface in Chapter 17, "High-Level CD-ROM Function Call Reference."

- Low-level source code that works with the CD-ROM device driver

  A set of "C" language function calls which have been prototyped in the `CDMASTER.H` file. You can find information on this interface in Chapter 18, "Low-Level CD-ROM Function Call Reference."

- Low-level source code that works with Microsoft CD-ROM extensions

  A set of "C" language function calls which have been prototyped in the `MSCDEX.PRO` file. You can find information on this interface in Chapter 19, "Microsoft CD-ROM Extension Function Call Reference."

Audio mixer

The audio mixer gives you complete control over audio recording and playback by providing the ability to connect input and output sources such as CD-ROM, PC speaker, microphone, FM synthesizers, and PCM circuitry; split monaural sound sources into a pair of channel signals; filter input and output mixer signals to remove aliases during PCM playback; and mix left- and right-channel input and output.

The mixer uses proprietary shielding and circuitry to produce high-fidelity, low- noise audio, with a frequency response of 30 Hz to 20 kHz. It interacts with all Pro AudioSpectrum devices and external sources like stereo equipment, microphones, and the PC's speaker.

Mixer software and hardware API's let you control parameters such as: input level for each source; stereo panning; treble and bass cut and boost; level control for each output channel; loudness; and, input/output device configuration.

Two software API's are available for the audio mixer:

- Text string command interface

  An English-like command interface that is integral to MS-DOS when the Pro AudioSpectrum device driver (`MVSOUND.SYS`) is loaded. You can find information on this interface in Chapter 21, "Command Line Mixer Interface."

- Low-level source code

  A set of "C" language function calls which link with the `MVSOUND.SYS` device driver and have been prototyped in the `MIXERS.H` file. You can find information on this interface in Chapter 22, "Low-Level Mixer Function Call Reference."

Bulletin board support

Media Vision provides developers with the latest software updates and technical support through its bulletin board. You can dial into (510) 770-0968 or (510) 770-0527 24 hours a day, 7 days a week to check the bulletin board.

Use the following protocol settings on your modem when dialing into the Media Vision bulletin board:

- 2400 or 1200 baud
- No parity
- 8 data bits
- 1 stop bit
- MNP level 5
- File compression with PKZIP version 1.1

End-users and developers can access this bulletin board. After signing on, send a message to the SYSOP requesting developer privileges. Once you submit a nondisclosure statement to Media Vision, you can download the most recent version of products in development and access confidential technical documentation. Additionally, you can exchange technical support messages with the SYSOP.

2 *Installing the Pro AudioSpectrum Developer's Toolkit*

This chapter provides the following information:

- A description of the PC system requirements for the Pro AudioSpectrum Developer's Toolkit
- Instructions describing how to install the Developer's Kit software
- A list of sample programs you can use to explore the Pro AudioSpectrum

PC system requirements

The Developer's Toolkit can run on virtually any PC that is suitable for C language development. Pro AudioSpectrum systems are ISA- and EISA-bus compatible for 286/386/486-based computer systems.

The Pro AudioSpectrum Developer's Toolkit software was developed and tested using the following software development tools:

- Microsoft C (version 6.0)
- Microsoft MASM (version 5.10)
- Microsoft Program Maintenance Utility, NMAKE.EXE, that is shipped with MASM 6.0 and Microsoft C 6.0.

Installation procedure

After you've familiarized yourself with the contents of the Developer's Toolkit diskette, install the software:

1. **Check the \PAS directory of the toolkit floppy to see if there is a READ.ME file with last minute additions to the documentation.**
2. **Use the DOS XCOPY command to move everything on the two diskettes to a directory on your machine.**

   You may use any drive. This example uses drive C:

   ```
   xcopy a:\*.* c:\ /s
   ```
3. **Modify the INCLUDE and LIB environment variables so that Microsoft C knows where to find the Pro AudioSpectrum include and library files.**

   Add the following to your `SET` statements for the `INCLUDE` and `LIB` environment variables:

   ```
   \inc;c:\pas\inc;
   ```
4. **Make sure that you have the proper version of NMAKE.EXE in the compiler directory and there are no similarly named NMAKE files in the path.**
5. **Load MVSOUND.SYS before you begin testing the utilities provided on the diskette.**

Experimenting with your Pro AudioSpectrum

After you've installed the Developer's Toolkit package, you can test out your Pro AudioSpectrum using one of the ready-to-run utility programs provided on the Developer's Toolkit diskette. Both the executable file and the source code are provided.

The sample programs listed below are the ideal starting point for learning how to do PCM programming. To learn more about what these utilities do, see Appendix E, "Pro AudioSpectrum Utility Programs."

Category	Program File
Tools	playfile.exe
	recfile.exe
	waveit.exe
	merge.exe
Example Source	playfile.c
	recfile.c
	blockin.c
	blockout.c

3 Common Function Calls and Hardware Registers

This chapter describes the function call and hardware register functions that are used with more than one feature of the Pro AudioSpectrum. Read this chapter before starting to program any Pro AudioSpectrum feature.

Common software API function call

Each of the software API's described in this book use the `mvGetHWVersion` function call.

mvGetHWVersion

`mvGetHWVersion` is an assembler routine that determines which Media Vision product and product revision level is installed.

This routine returns the product identifier and the hardware version number in a 16- bit field. Because the products contain multiple features, the values returned are a combination of the bit field settings shown below.

Use `USE_ACTIVE_ADDR` as defined in the `COMMON.INC` file to find the active base address. You can find a prototype usage of this function in the `GetHW.ASM` file.

Calling Convention

```
C    long MVGetHWVersion (intl baseaddr);

ASM  call MVGetHWVersion

     mov version, AX

     mov ProductID, DX

     mov featurebit, CX
```

Input Parameters

Parameter	Type	Value	Description
intl baseaddr	integer	0	Search for the board at one of four addresses.
		address	Search for the board at this specific address.

Return Values

Parameter	Type	Value	Description
long	32-bit signed integer	DX:AX = -1	Hardware not installed.
		DX:AX = -2	Hardware installed, can't identify it.
		DX = 0	Pro AudioSpectrum installed.
		DX = 1	Pro AudioSpectrum Plus installed.
		DX = 2	Pro AudioSpectrum 16 installed.
		DX = 3	CDPC installed.
		CX, D0 = 1	MVA 508 mixer chip installed.
		CX, D0 = 0	National mixer chip installed.
		CX, D1 = 1	PS/2 interface installed.
		CX, D2 = 1	CDPC slave device installed.
		CX, D3 = 1	SCSI interface installed.
		CX, D4 = 1	Enhanced SCSI interface installed.
		CX, D5 = 1	Sony 535 interface installed.
		CX, D6 = 1	16-bit DAC installed.
		CX, D7 = 1	Sound Blaster H/W emulation installed.
		CX, D8 = 1	MPU (Roland MIDI) H/W emulation installed.
		CX, D9 = 0	3812 installed.
		CX, D9 = 1	OPL3 installed.
		CX, DA = 1	MV101 MIDI interface installed.
		CX, DB = 1	Bit 0 of MV101 revision level.
		CX, DC = 1	Bit 1 of MV101 revision level.
		CX, DD = 1	Bit 2 of MV101 revision level.
		CX, DE = 1	Bit 3 of MV101 revision level.
		CX, DF = 1	Reserved.
		BX = 0	Always 0, reserved
		AH = 0	Pro AudioSpectrum installed
		AH = 2	Pro AudioSpectrum Plus or Pro AudioSpectrum 16 installed
		AH = 7	CDPC installed
Default			N/A

Related topics

None.

Common hardware API register functions

Three register functions are used by more than one of the Pro AudioSpectrum hardware API's:

- Audio Filter Control, Register B8Ah. Use this register to mute the Pro AudioSpectrum and control PCM sampling filters.
- Interrupt Control, Register B8Bh. Use this register to enable Pro AudioSpectrum feature interrupts.
- Interrupt Status, Register B89h. Use this register to determine that status of interrupts you enabled using the Interrupt Control Register.

The following is a description of these three register functions.

Audio Filter Control Register B8Ah

Use the Audio Filter Control Register to set bit masks that enable and disable counters and select filtering.

D7	D6	D5	D4	D3	D2	D1	D0
Sample Buffer Counter Mask	Sample Rate Timer Mask	Audio Mute	Filter Select Bit 4	Filter Select Bit 3	Filter Select Bit 2	Filter Select Bit 1	Filter Select Bit 0

Sample Buffer Counter Mask (D7)

Use this bit to enable the Sample Buffer Count Register.

For information on the Sample Buffer Count Register, see "Sample Buffer Count Register 1389h" on page 7-7.

	D7	Description
Settings:	1	Enable Sample Buffer Count Register.
	0	Disable Sample Buffer Count Register.
Default:	0	

Sample Rate Timer Mask (D6)

Use this bit to enable the Sample Rate Timer Register.

For information on the Sample Rate Timer Register, see "Sample Rate Timer Register 1388h" on page 7-6.

	D6	Description
Settings:	1	Enable Sample Rate Timer Register.
	0	Disable Sample Rate Timer Register.
Default:	0	

Audio Mute(D5)

Use this bit to enable and disable the Pro AudioSpectrum audio output. When the Pro AudioSpectrum is enabled, all audio sources (including the PC speaker) can be mixed and output to speakers or headphones.

When the Pro AudioSpectrum is disabled, only the PC speaker can output sound.

	D5	Description
Settings:	1	Enable Pro AudioSpectrum.
	0	Disable Pro AudioSpectrum.
Default:	0	

Filter Select (D4 though D0)

Use these bits to select a filter to eliminate unwanted high-frequency harmonics. For proper filtering and playback, select a filter with a limiting frequency equal to half the sample rate. If more than one audio signal source is combined in the input mixer, the lower quality audio signal should dictate the filter selection. All filter settings below have a low end threshold of 20 Hz.

	D4	D3	D2	D1	D0	Description
Settings:	0	0	0	0	1	Select filter rate of 17.897 kHz.
	0	0	0	1	0	Select filter rate of 15.909 kHz.
	0	1	0	0	1	Select filter rate of 11.931 kHz.
	1	0	0	0	1	Select filter rate of 8.948 kHz.
	1	1	0	0	1	Select filter rate of 5.965 kHz.
	0	0	1	0	0	Select filter rate of 2.982 kHz
Default:	0	0	0	0	0	

Interrupt Control Register B8Bh

Use the Interrupt Control Register to enable Pro AudioSpectrum feature interrupts and to determine the revision of the Pro AudioSpectrum hardware.

D7	D6	D5	D4	D3	D2	D1	D0
Board Revision Bit 2	Board Revision Bit 1	Board Revision Bit 0	MIDI Interrupt Enable	Sample Buffer Count Interrupt Enable	Sample Rate Timer Interrupt Enable	Right FM Interrupt Enable	Left FM Interrupt Enable

Board Revision (D7 though D5)

Use these bits to determine the current hardware revision level of the Pro AudioSpectrum. This is a read-only field.

	D7	D6	D5	Description
Settings:	X	X	X	Pro AudioSpectrum hardware revision level.
Default:				N/A

MIDI Interrupt Enable (D4)

Use this bit to enable the MIDI controller interrupt.

	D4	Description
Settings:	0	Disable MIDI controller interrupt.
	1	Enable MIDI controller interrupt.
Default:	0	

Sample Buffer Count Interrupt Enable (D3)

Use this bit to enable the sample buffer count interrupt. For more information on the sample buffer count register, see "Sample Buffer Count Register 1389h" on page 7-7.

	D3	Description
Settings:	0	Disable sample buffer count interrupt.
	1	Enable sample buffer count interrupt.
Default:	0	

Sample Rate Timer Interrupt Enable (D2)

Use this bit to enable the sample rate timer interrupt. For more information on the sample rate timer register, see "Sample Rate Timer Register 1388h" on page 7-6.

	D2	Description
Settings:	0	Disable sample rate timer interrupt.
	1	Enable sample rate timer interrupt.
Default:	0	

Right FM Interrupt Enable (D1)

Use this bit to enable the right channel FM synthesizer interrupt on older Pro AudioSpectrum systems. This interrupt has been redefined for the Pro AudioSpectrum Plus. Pro AudioSpectrum 16, and the CDPC.

	D1	Description
Settings:	0	Disable right channel FM interrupt.
	1	Enable right channel FM interrupt.
Default:	0	

Left FM Interrupt Enable (D1)

Use this bit to enable the left channel FM synthesizer interrupt.

	D0	Description
Settings:	0	Disable left channel FM interrupt.
	1	Enable left channel FM interrupt.
Default:	0	

Interrupt Status Register B89h

Use the Interrupt Status Register to determine if any of the interrupts you enabled need servicing. If you have initialized the Pro AudioSpectrum, and an interrupt occurs on the Pro AudioSpectrum, a PC hardware interrupt will occur. The PC then calls an interrupt service routine (ISR). The ISR must then read the Interrupt Status Register to determine the cause of the interrupt.

D7	D6	D5	D4	D3	D2	D1	D0
Sample Clipping Occurred	Reset	PCM Left/Right Flag	MIDI Interrupt Active	Sample Buffer Count Interrupt Active	Sample Rate Timer Interrupt Active	Right FM Interrupt Active	Left FM Interrupt Active

Sample Clipping Occurred (D7)

Read this bit to determine if the audio recording signal is too loud. When clipping occurs, this bit is automatically set to 1. Reduce the input mixer volume for the selected input sources to minimize clipping errors. This bit is reset to 0 after each use.

	D7	Description
Settings:	0	Volume OK.
	1	Clipping occurred.
Default:	0	

Reset (D6)

Read this bit to determine if the Pro AudioSpectrum is in reset mode. This status bit is automatically set to 0 if the Pro AudioSpectrum system is in the reset state. Writing to the Audio Filter Control Register brings the system out of reset.

For information on the Audio Filter Register, see "Audio Filter Control Register B8Ah" on page 3-4.

	D6	Description
Settings:	0	Pro AudioSpectrum is in reset state.
	1	Pro AudioSpectrum active.
Default:	0	

Note: This bit is only valid on the older Pro AudioSpectrum systems. This bit is reserved on the Pro AudioSpectrum Plus, Pro AudioSpectrum 16, and the CDPC.

PCM Left/Right Flag (D5)

Use this bit to determine if the left or right channel is active during PCM operations. In stereo mode, this bit is set to 0 when the left channel is active and set to 1 when the right channel is active. In mono mode, only the left channel is used, so this bit is always set to 0.

To learn how to set stereo or mono mode, see "Cross Channel Control Register F8Ah" on page 7-4.

	D5	Description
Settings:	0	Left channel active.
	1	Right channel active.
Default:	0	

MIDI Interrupt Active (D4)

Use this bit to determine if the MIDI controller is active. When this bit is set to 1, a MIDI interrupt has occurred indicating that an interrupt is waiting to be processed by the MIDI controller.

	D4	Description
Settings:	0	MIDI interrupt not in use.
	1	MIDI interrupt active.
Default:	0	

Sample Buffer Count Interrupt Active (D3)

Use this bit to determine if the sample buffer count interrupt is active, indicating that the sample buffer count is equal to 0. This interrupt is commonly used for PCM DMA buffer management. You may clear it by writing a 0 to this register.

	D3	Description
Settings:	0	Sample buffer count interrupt not in use.
	1	Sample buffer count interrupt active.
Default:	0	

Note: To perform PCM in polled mode, set bits D3 and D2 to 1.

Sample Rate Timer Interrupt Active (D2)

Use this bit to determine if the rate timer interrupt is active, indicating that the sample rate timer is equal to 0. This interrupt is not very useful since it causes interrupts coincident with the sample rate. Clear the interrupt by writing a 0 to this register.

	D2	Description
Settings:	0	Sample rate timer interrupt not in use.
	1	Sample rate timer interrupt active.
Default:	0	

Note: To perform PCM in polled mode, disable bits D3 and D2 by setting them equal to 1.

Right FM Interrupt Active (D1)

Use this bit to determine if the right FM synthesizer interrupt is active, indicating that one or both FM timers have reached a 0 count. Read the Right FM Synthesizer Address and Status port (38Ah) to determine which timer caused the interrupt. Reset this bit by acknowledging the interrupt.

	D1	Description
Settings:	0	Right FM interrupt not in use.
	1	Right FM interrupt active.
Default:	0	

Note: This interrupt has been redefined on the Pro AudioSpectrum Plus, Pro AudioSpectrum 16, and the CDPC.

Left FM Interrupt Active (D1)

Use this bit to determine if the left FM synthesizer interrupt is active, indicating that one or both FM timers have reached a 0 count. Read the Left FM Synthesizer Address and Status port (388h) to determine which timer caused the interrupt. Reset this bit by acknowledging the interrupt.

	D2	Description
Settings:	0	Left FM interrupt not in use.
	1	Left FM interrupt active.
Default:	0	

PCM Programming Section

4 PCM Programming Essentials

This chapter describes:

- How to access PCM function calls
- The theory of operation of the Pro AudioSpectrum PCM system
- General use of the High-Level PCM API
- General use of the Low-Level PCM API

Three distinct PCM programming interfaces are available for the PCM system: the high level software API, described in Chapter 5, "High-Level PCM Function Call Reference," the low level software API, described in Chapter 6, "Low-Level PCM Function Call Reference," and the direct hardware programming, described in Chapter 7, "PCM Hardware Register Functions."

Accessing PCM function calls

You make PCM function calls through statically linked subroutines. Pro AudioSpectrum PCM routines are linked through the usual compile/link process.

The PCM software library is provided as source files on your Pro AudioSpectrum Developer's Toolkit diskette. You can find the high-level routines in the `PCMIO.H` file and the low-level calls in the `MVSOUND.ASM` file.

With the exception of the routines listed below, you should call only high-level block/file routines or low-level routines - not a mix of both.

You can use the following routines in both high-level and low-level derived programs:

- PausePCM
- ResumePCM

For more information on these function calls, see Chapter 6, "Low-Level PCM Function Call Reference."

Theory of Operation

This section describes the operation of DMA-controlled PCM from a high-level, software-oriented perspective. This discussion provides the context for using the three distinct programming interfaces for the PCM.

The Pro AudioSpectrum PCM circuitry must process audio data in real time to ensure an unbroken stream of audio information. Failure to keep pace with input (recording) or output (playback) leads to serious defects in sound quality. The Pro AudioSpectrum uses the PC's DMA channel to perform rapid, direct-to-memory transfers of audio data.

To ensure continuous playback and recording, the Pro AudioSpectrum uses the auto-initialize mode of the DMA controller and multiple buffers. Since MS-DOS is neither a real-time nor re-entrant operating system, the software library uses a buffer management system with distinct foreground and background tasks.

The DMA buffer is split into an even number of divisions. The Sample Buffer counter on the Pro AudioSpectrum interrupts the CPU when one buffer has filled (or emptied) a buffer division.

The buffer divisions can be 1/2, 1/4, 1/8, or a smaller fraction of the overall buffer. A buffer division that is too small results in higher CPU overhead while the system processes interrupts. An excessively large buffer takes valuable system memory without yielding a gain in performance.

PCM Output

Use two levels of buffering to perform PCM output (playback). The top level is a linked list of buffers, which is loaded by the foreground task and unloaded by the background task. The lower buffer is loaded by the background task and unloaded by the DMA controller. The DMA controller transfers this data to the PCM circuitry for playback.

To start PCM output, the foreground task loads all the buffers in the linked list, then calls the background task to begin playing. The background task loads the DMA output buffer every time it receives a sample buffer counter interrupt. This mechanism lets the background and foreground tasks operate asynchronously, keeping the DMA buffer and the linked list of buffers filled.

If the foreground task keeps the linked list of buffers full, there should be continuous PCM output. If the foreground task cannot keep up, the background task is forced to stop the DMA, causing an audible break in the output. The foreground task is then obligated to restart the background task.

PCM Input

Use two levels of buffering to perform PCM input (recording). The bottom level is the DMA input buffer. This buffer is filled with sample data from the PCM circuitry and is unloaded by the background tasks into an empty buffer in a circular linked list of buffers.

This linked list of buffers is the top level of buffering. The foreground tasks unload each top level buffer by writing the contents of the buffer to disk.

To start PCM input, the foreground task calls low-level code that begins the background task. The linked list of buffers is initialized and DMA is enabled.

The background task unloads the DMA input buffer to a top level buffer in the linked list. The foreground task is responsible for writing each filled buffer to disk.

If the foreground task keeps the linked list of buffers empty, there should be continuous PCM input. If the foreground task cannot keep up, the background task is forced to stop the DMA, causing an audible break in the input. The foreground task is then obligated to restart the background task.

Using the High-Level PCM API

You can use the high-level PCM function calls for either file or block programming. Both types of I/O have advantages. File I/O is simpler and requires less memory. Block I/O uses more memory (because you must maintain your own buffers in addition to the internal PCM buffer created by the call to OpenPCMBuffering), but provides far greater control. It is important to understand the differences between these styles of I/O before beginning work.

High-level PCM function calls are organized according to how they are used: file and block, file only, and block only.

The first group of functions apply to both file and block programming:

- ClosePCMBuffering
- OpenPCMBuffering
- PCMState
- StopDMAIO

The second group is segregated according to the file/block programming distinction and performs *start, continue input,* and *output* procedures.

The five file-only routines are:

- StartFileInput
- StartFileOutput
- ContinueFileInput
- ContinueFileOutput
- ASpecialContinueFileInput

The four block-only routines are:

- StartBlockInput
- StartBlockOutput
- ContinueBlockInput
- ContinueBlockOutput

Global Variables

The following list shows global counters and flags used in the high-level PCM API processing:

Global Variable	Description
int BufferDataCount	Number of full buffers.
int DMARunning	DMA processing status. 0=off,1=running.
int ProcessedBlockCount	Number of blocks handled by DMA.

BufferDataCount

`BufferDataCount` is a key handshaking variable between processes. It holds a count of foreground buffers containing data.

For output, `BufferDataCount` is incremented each time a buffer is loaded by the foreground task, and is then decremented when a buffer is emptied by the background task.

For input, `BufferDataCount` is incremented by the background task and decremented by the foreground task.

DMARunning

`DMARunning` indicates the status of the DMA channel. It is set to 1 when DMA is running (either playback or record). It is set to 0 when DMA is turned off (disabled).

ProcessedBlockCount

`ProcessBlockCount` is the total count of blocks (a block is a buffer division) processed since the beginning of I/O.

High-level PCM API programming steps

The following procedure shows the order in which you should use high level API function calls. The function calls are documented in Chapter 5, "High-Level PCM Function Call Reference."

1. **Initialize PCM**

   Function call: `OpenPCMBuffering`

2. **Set PCM parameters**

   Function call: `PCMState`

3. **Start input I/O**

   Function calls: `StartBlockInput`
   `StartFileInput`

4. **Start output I/O**

   Function calls: `StartBlockOutput`
   `StartFileOutput`

5. **Continue input I/O**

   Function calls: `ContinueBlockInput`
   `ContinueFileInput`
   `ASpecialContinueFileInput`

6. **Continue output I/O**

   Function calls: `ContinueBlockOutput`
   `ContinueFileOutput`

7. **Pause I/O**

   Function call: `PausePCM`

8. **Resume I/O**

   Function call: `ResumePCM`

9. **Shut down PCM**

   Function call: ClosePCMBuffering

Return codes

The following is a list of codes returned by the high-level block/file routines. Many PCM routines return a 0 value if the routine was successful, or a non-zero value if an error occurred.

Error Code	Definition
PCMIOERR_SAMPLERATE	Invalid sample rate requested.
PCMIOERR_OPENFILE	Error opening the output file.
PCMIOERR_OPENPCM	Error starting the PCM code.
PCMIOERR_NOMEM	Error starting the PCM code.
PCMIOERR_BADDMA	Invalid DMA number requested.
PCMIOERR_BADIRQ	Invalid IRQ number requested.
PCMIOERR_FILEFULL	Cannot write data to the file.

Using the Low-Level PCM API

The Pro AudioSpectrum's low-level PCM function calls provide maximum control and minimum overhead. The high-level PCM API was built using the low-level PCM routines.

Low-level PCM API programming steps

The following procedure shows the order in which you should use low-level PCM function calls. The function calls are documented in Chapter 6, "Low-Level PCM Function Call Reference."

1. **Initialize PCM routines**

   Function call: `InitMVSound`

2. **Initialize PCM**

   Function call: `InitPCM`

3. **Set PCM Parameters**

   Function calls: `PCMinfo`

   `SetDMA`

   `SetIRQ`

4. **Set up buffering**

   `FindDMABuffer`

   `DMABuffer`

5. **Set up call back routine**

   `UserFunc`

6. **Start input I/O**

   `PCMRecord`

7. **Start output I/O**

   `PCMPlay`

8. **Pause I/O**

   `PausePCM`

9. **Resume I/O**

   `ResumePCM`

10. **Stop PCM**

    `StopPCM`

11. **Shut Down PCM**

    `RemovePCM`

5 High-Level PCM Function Call Reference

This chapter describes the highest level PCM software interface, which is appropriate for file and block-level programming. For information on PCM programming basics, see Chapter 4, "PCM Programming Essentials." For information on the Pro AudioSpectrum's low-level software API, see Chapter 6, "Low-Level PCM Function Call Reference." For information on hardware level programming, see Chapter 7, "PCM Hardware Register Functions."

You can find prototypes of the high level routines in the `PCMIO.H` file.

ASpecialContinueFileInput

Use `ASpecialContinueFileInput` to perform voice-activated recording. As each block of data is recorded, it is scanned for changes in the audio signal. If changes exceed the threshold, the block is written to disk; otherwise, the block is discarded.

This routine is similar to `ContinueFileInput` except that it records input only while the signal level exceeds the noise threshold. You must poll the `ASpecialContinueFileInput` routine continuously in order to sustain recording to disk.

The `noise` parameter sets the threshold level to begin recording. It is specified as a delta value from silence. Sample data has a dynamic range of 128 steps. The center point is 80h, which is the value for silence.

The `rectype` parameter controls whether recording starts and continues once the signal-level threshold is reached, or whether only those samples that exceed the signal-level threshold are recorded.

Calling Convention

```
int ASpecialContinueFileInput(int noise, int rectype);
```

Input Parameters

Parameter	Type	Value	Description
noise	integer	0h to 7Fh	Specifies the signal level that is considered noise; only signals with a magnitude greater than this level will trigger recording. The noise level is specified as one of 128 steps within the dynamic range of PCM data.
rectype	integer	0	Enable latch recording. Record everything when first sample exceeds noise threshold.
		1	Enable threshold recording. Record only buffers that contain samples that exceed noise threshold.

Return Values

Parameter	Type	Value	Description
int	integer	0	DMA recording complete.
		non-zero	DMA recording still in process.

Related Topics

To begin the recording process, see"StartFileInput" on page 5-10.

To capture and record all input, see "ContinueFileInput" on page 5-5.

ClosePCMBuffering

Use `ClosePCMBuffering` to close down the PCM I/O system.

You must call this routine before your program terminates to ensure file buffers are closed, memory is released properly, and interrupt vectors are restored.

Calling Convention

```
void ClosePCMBuffering();
```

Input Parameters

None.

Return Values

None.

Related Topics

To stop DMA transfers, see "StopDMAIO" on page 5-12.

To set up the PCM I/O system, see "OpenPCMBuffering" on page 5-7.

ContinueBlockInput

Use `ContinueBlockInput` to sustain PCM block recording.

This routine is a similar to `ContinueFileInput` except that it records input to a memory block rather than writing it directly to a file. If PCM data has been loaded into the buffers created by the `OpenPCMBuffering` call, this function will cause it to be transferred to your buffer.

You must poll the global variable *DMARunning* to make sure that DMA is still running during this function call.

Note: Your application must call `ContinueBlockInput` frequently to avoid losing PCM data.

Calling Convention

```
int ContinueBlockInput (char far *block);
```

Input Parameters

Parameter	Type	Value	Description
*block	far pointer		Far pointer to the next block in the linked list of buffers you have reserved for storing PCM data. The block must match the buffer division in size.

Return Values

Parameter	Type	Value	Description
int	integer	0	PCM buffer empty.
		non-zero	DMA has loaded PCM buffer.

Related Topics

To begin recording to memory, see"StartBlockInput" on page 5-8.

To stop DMA transfers, see "StopDMAIO" on page 5-12.

To pause DMA transfers, see "PausePCM" on page 6-5.

To resume DMA transfers, see "ResumePCM" on page 6-9.

ContinueBlockOutput

Use `ContinueBlockOutput` to sustain PCM block playback.

This routine is similar to `ContinueFileOutput`. You must call `ContinueBlockOutput` repeatedly to keep the PCM buffer (created by the `OpenPCMBuffering` call) full of new data. When the `ContinueBlockOutput` function returns a non-zero value, you must pass a pointer to the next block in your linked list.

Your application must call this routine frequently to keep PCM output flowing.

Calling Convention

```
int ContinueBlockOutput (char far *block);
```

Input Parameters

Parameter	Type	Value	Description
*block	far pointer		Far pointer to the next block in the linked list of buffers you have reserved for storing PCM data. The block must match the buffer division in size.

Return Values

Parameter	Type	Value	Description
int	integer	0	PCM buffer empty.
		non-zero	DMA has loaded PCM buffer.

Related Topics

To begin playing from memory, see "StartBlockOutput" on page 5-9.

To stop DMA transfers, see "StopDMAIO" on page 5-12.

To pause DMA transfers, see "PausePCM" on page 6-5.

To resume DMA transfers, see "ResumePCM" on page 6-9.

ContinueFileInput

Use `ContinueFileInput` to sustain PCM recording to disk.

This routine writes all new PCM blocks to disk. Your application should call this routine repeatedly until you are ready to stop PCM recording.

Calling Convention

```
int ContinueFileInput();
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer	0	PCM buffer loading complete.
		non-zero	PCM buffer loading in process.

Related Topics

To start recording to disk, see "StartFileInput" on page 5-10.

To stop DMA transfers, see "StopDMAIO" on page 5-12.

ContinueFileOutput

Use `ContinueFileOutput` to sustain PCM playback.

This routine keeps data moving from disk to the PCM buffers. Your application must call this routine frequently to ensure continuous sound output.

Calling Convention

```
int ContinueFileOutput();
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer	0	PCM output has stopped.
		non-zero	PCM output in process.

Related Topics

To start playing from disk, see "StartFileOutput" on page 5-11.

To pause DMA transfers, see "PausePCM" on page 6-5.

To resume DMA transfers, see "ResumePCM" on page 6-9.

To stop DMA transfers, see "StopDMAIO" on page 5-12.

OpenPCMBuffering

Use `OpenPCMBuffering` to initialize the PCM environment for input and output.

This function tells the low-level routines which DMA channel, IRQ level, buffer size, and number of divisions to use. `OpenPCMBuffering` allocates the appropriate buffers using `halloc()`, the huge version of `malloc()`. Using `halloc()` avoids consuming critical near-data segment memory.

Note: 20 bytes per division are allocated from the near-data segment for buffer management control.

Calling Convention

```
int OpenPCMBuffering (int channel, int level, int size, int
divisions);
```

Input Parameters

Parameter	Type	Value	Description
channel	integer	1, 2, 3, 5, 6, or 7	Specifies the DMA channel to use for feeding the PCM circuitry.
		-1	Use the default DMA channel.
level	integer	1, 2, 3, 5, 7, 10, 11, 12, or 15	Specify the IRQ level to use to control the DMA channel that feeds the PCM circuitry.
		-1	Use the default IRQ level.
size	integer	4, 8, 16, 32, or 64	Specifies total size of the DMA buffer in kilobytes.
divisions	integer	2, 4, 8, or 16	Specifies the number of buffer partitions.

Return Values

Parameter	Type	Value	Description
int	integer	0	Initialization succeeded.
		non-zero	Initialization failed

Related Topics

To close down PCM I/O, see "ClosePCMBuffering" on page 5-3.

PCMState

Use `PCMState` to set parameters for PCM.

This routine configures the low-level calls to operate at the desired sampling rate, specifies whether to interpret data in stereo or mono mode, and selects 8- or 16-bit PCM mode.

`PCMState` is identical to the low-level `PCMInfo` function.

Calling Convention

```
int PCMState (long rate, int stereo, int compression, int psize);
```

Input Parameters

Parameter	Type	Value	Description
rate		8,000 to 88,200	Specifies sample rate in cycles per second.
stereo	integer	0	Set sound mode to mono.
		1	Set sound mode to stereo.
compression	integer	0	Default setting. Compression parameter not currently defined.
psize	integer	8	Set PCM mode to 8-bit.
		16	Set PCM mode to 16-bit.

Return Values

Parameter	Type	Value	Description
int	integer	0	Initialization succeeded.
		non-zero	Initialization failed. See error codes to determine specific reason for failure.

Related Topics

None.

StartBlockInput

Use `StartBlockInput` to begin PCM recording to memory.

This routine is similar to `StartFileInput` except that it records input to memory rather than writing it directly to a file. Use `ContinueBlockInput` to fill the caller's memory with all subsequent blocks.

Calling Convention

```
int StartBlockInput();
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer	0	Operation succeeded.
		non-zero	Operation failed. DMA is inactive; an error occurred, possibly because a failure to call PCMState() or incorrect parameters passed by PCMState() or DMABuffer().

Related Topics

To continue recording to memory, see "ContinueBlockInput" on page 5-3.

To set up the PCM I/O system, see "OpenPCMBuffering" on page 5-7.

To set up the PCM environment, see "PCMState" on page 5-8.

StartBlockOutput

Use `StartBlockOutput` to initiate PCM block playback.

This routine initiates playback of PCM data stored in your buffer. The DMA copies this data to the PCM buffer created by the call to `OpenPCMBuffering`. PCM output begins to play this data immediately. Once started, you must call `ContinueBlockOutput` repeatedly to sustain output of all subsequent blocks.

Calling Convention

```
int StartBlockOutput (char far *block);
```

Input Parameters

Parameter	Type	Value	Description
*block	far pointer		Far pointer to your first block reserved for holding PCM data ready for PCM output.

Return Values

Parameter	Type	Value	Description
int	integer	0	DMA started.
		non-zero	DMA initiation failed.

Related Topics

To sustain PCM block playback, see "StartFileInput" on page 5-10.

To stop DMA transfers, see "StopDMAIO" on page 5-12.

To pause DMA transfers, see "PausePCM" on page 6-5.

To resume DMA transfers, see "ResumePCM" on page 6-9.

To set up the PCM environment, see "PCMState" on page 5-8.

StartFileInput

Use `StartFileInput` to initiate PCM data recording.

This routine starts PCM digitized audio data recording. All blocks will be read from the disk file specified by `FILE *fstream` parameter. After initiating the recording process, you must call `ContinueFileInput` repeatedly to sustain file input processing.

Note: This call does not create header information associated with a .WAV or .VOC file.

Calling Convention

```
int StartFileInput (FILE *fstream);
```

Input Parameters

Parameter	Type	Value	Description
FILE *fstream			Pointer to the PCM input file.

Return Values

Parameter	Type	Value	Description
int	integer	0	DMA started.
		non-zero	DMA initiation failed.

Related Topics

To capture and record all input, see "ContinueFileInput" on page 5-5.

To stop DMA transfers, see "StopDMAIO" on page 5-12.

To pause DMA transfers, see "PausePCM" on page 6-5.

To resume DMA transfers, see "ResumePCM" on page 6-9.

To set up the PCM environment, see "PCMState" on page 5-8.

StartFileOutput

Use `StartFileOutput` to begin playing PCM data from disk.

This routine initiates playback of PCM data stored in the disk file specified by `FILE *fstream` parameter. The `long count` parameter specifies the number of bytes to read and play. All bytes will be played until the count is exhausted and the file pointer will be positioned to the next byte in the file.

After initiating the playback process, you must call `ContinueFileOutput` repeatedly to sustain file output processing.

Note: This call does not recognize the header information associated with a .WAV or .VOC file. StartFileOutput plays data beginning with the first byte of the file. You must process the header before calling this routine.

Calling Convention

```
int StartFileOutput (FILE *fstream, long count);
```

Input Parameters

Parameter	Type	Value	Description
FILE *fstream			Pointer to the PCM output file.
count	signed integer		Specifies the number of bytes to read and play.
		-1	Read and play all available data.

Return Values

Parameter	Type	Value	Description
int	integer	0	DMA started.
		non-zero	DMA initiation failed.

Related Topics

To continue playing PCM data from disk, see "ContinueFileOutput" on page 5-6.

To stop DMA transfers, see "StopDMAIO" on page 5-12.

To pause DMA transfers, see "PausePCM" on page 6-5.

To resume DMA transfers, see "ResumePCM" on page 6-9.

To set up the PCM environment, see "PCMState" on page 5-8.

StopDMAIO

Use `StopDMAIO` to terminate PCM I/O.

This routine terminates the PCM I/O immediately, even if the PCM buffers have more PCM data or sound input to process. `StopDMAIO` flushes all buffers to prepare for a new start command. PCM can be restarted by calling one of the "start" functions.

The `StopDMAIO` call frees the DMA channel. This allows the Pro AudioSpectrum to share a DMA channel with another device without conflict.

Note: `StopDMAIO` terminates processing, it does not pause recording or playback.

Calling Convention

```
void StopDMAIO ();
```

Input Parameters
None.

Return Values
None.

Related Topics
To close down PCM I/O, see "ClosePCMBuffering" on page 5-3.

To pause DMA transfers, see "PausePCM" on page 6-5.

To resume DMA transfers, see "ResumePCM" on page 6-9.

6 Low-Level PCM Function Call Reference

This chapter describes low-level PCM function calls. These routines are the lowest-level PCM routines provided in the Pro AudioSpectrum Developer's Toolkit. For background information on PCM programming, see Chapter 4, "PCM Programming Essentials." For simpler programming or for rapid prototyping, use the block or file-level routines described in Chapter 5, "High-Level PCM Function Call Reference."

Unless otherwise specified, you can find prototypes of each of the functions documented here in the `MVSOUND.H` file.

DMABuffer

Use DMABuffer to pass a pointer to the DMA buffer.

This routine receives the physical DMA buffer, size, and number of divisions. The minimum recommended buffer size for low sampling rates is 4 K; the minimum number of divisions is 2.

The DMA buffer is a circular buffer. By using the auto-initialize mode of the DMA controller, the CPU can keep an uninterrupted stream of data moving to or from the Pro AudioSpectrum.

Each time a buffer division is processed, an interrupt is generated. The interrupt, in turn, triggers the CPU to process the next buffer division.

Note: You must call InitMVSound() and then InitPCM() before using this function. The PCM library routines require the DMA buffer to be a contiguous block of memory that doesn't span a 64 K boundary. The function FindDMABuffer() helps you to allocate such a block.

Calling Convention

```
void far * DMABuffer (char far *buffer, int size, int divisions);
```

Input Parameters

Parameter	Type	Value	Description
*buffer		far pointer	Pointer to the buffer buff that you have already allocated.
size	integer	4, 8, 16, 32, or 64	Total size of the buffer, in kilobytes.
divisions	integer	2, 4, 8, or 16	The number of partitions that buffer buff will be divided into. A value of 16 gives the smallest recommended division size; any smaller size results in excessive interrupt overhead.

Return Values

Parameter	Type	Value	Description
void far *		non zero pointer	Call was successful.
		null pointer	Call failed.

Related Topics

To determine the hardware features, IRQ settings, and DMA settings of the Pro AudioSpectrum, see "InitMVSound" on page 6-4.

To find an appropriate DMA buffer, see "FindDMABuffer" on page 6-3.

To hook the interrupt, see "InitPCM" on page 6-4.

To register a call back at interrupt time, see "UserFunc" on page 6-11.

FindDMABuffer

Use FindDMABuffer to find a DMA buffer that lies within the 64K boundary.

Calling Convention

```
char far * FindDMABuffer(char huge *block, int size);
```

Input Parameters

Parameter	Type	Value	Description
*block	pointer		Pointer to a block of memory that is at least twice as large as the required buffer size.
size	integer	4, 8, 16, 32, or 64	DMA buffer size in kilobytes (1024 bytes).

Return Values

Parameter	Type	Value	Description
char far *	pointer	non-zero pointer	Far pointer to a usable DMA buffer. The huge pointer passed in should point to a memory block twice the size of the required DMA buffer size. This huge pointer is converted to a far pointer.
		null pointer	Indicates that a valid far pointer cannot be obtained.

Related Topics

To use the DMA buffer returned by this call, see "DMABuffer" on page 6-2.

InitMVSound

Use `InitMVSound` to initialize the low-level code and link to the hardware state table.

This function establishes a link with the `MVSOUND.SYS` driver that controls the Pro AudioSpectrum. If the `MVSOUND.SYS` driver is found, a far pointer is returned to the MVState structure within the driver's data segment that contains hardware state information. This enables various programs to share the current hardware state. Your application calls `InitMVSound` just once, before any other library calls are made.

Calling Convention

```
void MVState far * InitMVSound();
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
MVState far *	far pointer		Pointer to the hardware state table.

Related Topics

"InitPCM" on page 6-4

InitPCM

Use InitPCM to enable the Pro AudioSpectrum to use the IRQ and DMA channels.

This function initializes the library routines that handle PCM input and output. Your application calls this routine once when it loads. This code initializes the PCM state machine with the following settings:

- Default IRQ and DMA selections
- Default sample rate
- Mono recording mode

Call `InitPCM` a second time only if you called `RemovePCM` to terminate PCM and wish to do further PCM.

Calling Convention

```
int InitPCM();
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer		Version of the library code.

Related Topics

To initialize the PCM library, see "InitMVSound" on page 6-4.

To release the IRQ and DMA, see "RemovePCM" on page 6-8.

PausePCM

Use `PausePCM` to temporarily stop PCM I/O.

This function temporarily stops PCM input/output by disabling the Sample Rate Timer. The PCM hardware and software state is frozen until `ResumePCM` is called.

Calling Convention

```
void PausePCM();
```

Input Parameters

None.

Return Values

None.

Related Topics

To resume DMA transfers, see "ResumePCM" on page 6-9.

To stop DMA transfers, see "StopPCM" on page 6-10.

PCMInfo

Use `PCMInfo` to set parameters for PCM.

This function configures the low-level code to operate at the desired sampling rate and indicates whether to interpret data as stereo or mono.

The sample rate is set between 4,000 and 44,100 kHz for both mono and stereo mode, which is set with the `stereo` parameter.

Compression is not currently supported.

The PCMsize parameter determines whether 8- or 16-bit data is used.

Calling Convention

```
int PCMInfo (long rate, int stereo, int compress, int PCMsize);
```

Input Parameters

Parameter	Type	Value	Description
rate	32-bit signed integer	4,000 to 44,100	Set rate in samples per seconds.
stereo	integer	1	Interpret data as stereo.
		0	Interpret data as mono.
compress	integer		Currently unused, set to 0.
PCMsize	integer	8	Use 8-bit PCM data.
		16	Use 16-bit PCM data.

Return Values

Parameter	Type	Value	Description
int	integer	0	Indicates that rate entered was within range.
		1	Indicates that rate entered was not within range.

Related Topics

None.

PCMPlay

Use `PCMPlay` to start playing PCM.

This function call causes the PCM circuitry to convert PCM digitized audio data into audio sound using DMA.

The function takes advantage of the auto-initialization mode of the DMA controller to maintain a flow of data. Once DMA is started, an interrupt is generated at each buffer division.

Your application must keep the DMA buffers loaded; the callback routine must continually reload the DMA buffer division to ensure an uninterrupted flow of data.

Calling Convention

```
int PCMPlay();
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer	0	Playing began successfully.
		non-zero	Error. A failure occurred.

Related Topics

For a discussion of PCM buffering and a description of relevant global variables, see Chapter 4, "PCM Programming Essentials."

To temporarily pause PCM output, see "PausePCM" on page 6-5.

To stop the PCM output process completely, see "StopPCM" on page 6-10.

To record PCM data, see "PCMRecord" on page 6-8.

To receive a call back at interrupt time, see "UserFunc" on page 6-11.

PCMRecord

Use `PCMRecord` to start recording PCM.

This function causes the PCM circuitry to begin recording digitized audio data to the DMA buffer. This function turns on the PCM state machine, enables the timers and enables the interrupt that is generated for each buffer division.

Calling Convention

```
int PCMRecord();
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer	0	Recording began successfully.
		non-zero	Error. A failure occurred.

Related Topics

For a discussion of PCM buffering and a description of relevant global variables, see Chapter 4, "PCM Programming Essentials."

To temporarily pause PCM recording, see"PausePCM" on page 6-5.

To stop the PCM output process completely, see "StopPCM" on page 6-10.

To play back PCM data, see "PCMPlay" on page 6-7.

To receive a call back at interrupt time, see "UserFunc" on page 6-11.

RemovePCM

Use `RemovePCM` to restore the interrupt.

This function frees the DMA and IRQ channels. If you elect to do further PCM, you must re-initialize the PCM library by calling `InitPCM`; you need not call `InitMVSound` again.

Calling Convention

```
void RemovePCM();
```

Input Parameters
None.

Return Values
None.

Related Topics
To stop the PCM output process completely, see "StopPCM" on page 6-10.

To set up the interrupt chain, see "InitPCM" on page 6-4.

To play back PCM data, see "PCMPlay" on page 6-7.

ResumePCM

Use `ResumePCM` to restart PCM I/O.

This function restarts the PCM circuitry from its state prior to a `PausePCM` call and re-enables the PCM timers.

The Sample Rate Timer is reloaded, so changes to the rate can occur between pause and resume statements.

Calling Convention

```
void ResumePCM();
```

Input Parameters
None.

Return Values
None.

Related Topics
To temporarily pause PCM output, see "PausePCM" on page 6-5.

To stop the PCM output process completely, see "StopPCM" on page 6-10.

StopPCM

Use StopPCM to halt PCM processing.

This function terminates PCM input/output processing by turning off the PCM timers, disabling the interrupts, resetting the PCM state machine, and freeing up the DMA channel. This function allows the Pro AudioSpectrum to share a DMA channel with another device without conflict.

Use `RemovePCM` before terminating your program.

Calling Convention

```
void StopPCM();
```

Input Parameters

None.

Return Values

None.

Related Topics

To temporarily pause PCM output, see "PausePCM" on page 6-5.

To resume DMA transfers, see "ResumePCM" on page 6-9

To play back PCM data, see "PCMPlay" on page 6-7.

To record PCM data, see "PCMRecord" on page 6-8.

UserFunc

Use `UserFunc` to pass a call- back routine to the low-level library.

This function registers your routine to be called back at interrupt time. The objective is to call your routine to keep a continuous flow of data moving to and from the DMA buffer. While one part of the buffer is depleted (or replenished) by the PCM circuitry, the other part is refilled (or emptied). When an interrupt occurs, your function is called. UserFunc must then load (or unload) the DMA buffer.

Caution: You should not attempt to use the Microsoft C interrupt function declaration because it generates an IRET that will crash your application. Your UserFunc() routines must be an ASM module that saves and restores registers.

Note: This code must not execute DOS calls unless it can predetermine the state of DOS.

Note: Your routine should save all registers when called.

Calling Convention

```
void UserFunc (long far *routine);
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
*routine	32-bit signed integer		Far pointer to a routine that will be called at interrupt time.

Related Topics

None.

7 PCM Hardware Register Functions

This chapter is a reference to the Pro AudioSpectrum's PCM hardware register functions. Both the high- and low-level function call interfaces are built upon this interface. PCM hardware register functions provide you with ultimate control over PCM programming.

The Pro AudioSpectrum's PCM circuitry, aided by DMA, samples and plays back PCM waveforms. With DMA, the Pro AudioSpectrum sends or receives data samples without direct program involvement. You must ensure that the DMA is set up with a dedicated memory area to process DMA transfers. Using double -buffering techniques, you can create continuous sound waveforms by simply filling one buffer area while the other is being played. For more information on using DMA, see Appendix C, "Programming the PC's Interrupt Controller and DMA Channels."

Audio Filter Control Register B8Ah

Use the Audio Filter Control Register to set bit masks that enable and disable counters and allow the system to generate interrupts; mute and reset the board; and, select filtering for PCM playback.

D7	D6	D5	D4	D3	D2	D1	D0
Sample Buffer Counter Gate	Sample Rate Timer Gate	Audio Mute	Filter Select Bit 4	Filter Select Bit 3	Filter Select Bit 2	Filter Select Bit 1	Filter Select Bit 0

Sample Buffer Counter Gate (D7)

Use this bit to enable the Sample Buffer Count register.

For information on the Sample Buffer Counter, see "Sample Buffer Count Register 1389h" on page 7-7.

	D7	Description
Settings:	1	Enable Sample Buffer Counter.
	0	Disable Sample Buffer Counter.
Default:	0	

Sample Rate Timer Gate (D6)

Use this bit to enable the Sample Rate Timer Register.

For information on the Sample Rate Timer, see "Sample Rate Timer Register 1388h" on page 7-6.

	D6	Description
Settings:	1	Enable Sample Rate Timer.
	0	Disable Sample Rate Timer.
Default:	0	

Audio Mute(D5)

Use this bit to enable and disable the Pro AudioSpectrum audio output. When the Pro AudioSpectrum is enabled, all audio sources (including the PC speaker) can be mixed and output to speakers or headphones.

When the Pro AudioSpectrum is disabled, only the PC speaker can output sound.

	D5	Description
Settings:	1	Enable Pro AudioSpectrum.
	0	Disable Pro AudioSpectrum.
Default:	0	

Filter Select (D4 though D0)

Use these bits to select a filter to eliminate unwanted high-frequency harmonics. For proper filtering and playback, select a filter with a limiting frequency equal to half the sample rate. If more than one audio signal source is combined in the input mixer, the lower quality audio signal should dictate the filter selection. All filter settings below have a low -end threshold of 20 Hz.

	D4	D3	D2	D1	D0	Description
Settings:	0	0	0	0	1	Select filter rate of 17.897 kHz.
	0	0	0	1	0	Select filter rate of 15.909 kHz.
	0	1	0	0	1	Select filter rate of 11.931 kHz.
	1	0	0	0	1	Select filter rate of 8.948 kHz.
	1	1	0	0	1	Select filter rate of 5.965 kHz.
	0	0	1	0	0	Select filter rate of 2.982 kHz
Default:	0	0	0	0	0	

PCM Data Register F88h

Use the PCM Data register to read samples from the Analog-to-Digital Converter (ADC) and write data to the Digital-to-Analog Converter (DAC). All eight bits of this register are programmed as a group.

To set the sample rate, see "Sample Rate Timer Register 1388h" on page 7-6. Use the PCM Data Register only for direct mode transfers, not for non-DMA PCM.

D7	D6	D5	D4	D3	D2	D1	D0
PCM Data Bit 7	PCM Data Bit 6	PCM Data Bit 5	PCM Data Bit 4	PCM Data Bit 3	PCM Data Bit 2	PCM Data Bit 1	PCM Data Bit 0

Cross Channel Control Register F8Ah

Use Cross Channel Control register to manage the PCM hardware and DMA interface; and to configure the channel connections between mixers. When the Pro AudioSpectrum is reset, all bits in this register are cleared (set to 0).

D7	D6	D5	D4	D3	D2	D1	D0
DMA Enable	PCM Enable	Mono/Stereo Mode	DAC/ADC Mode	Left to Left	Right to Left	Left to Right	Right to Right

DMA Enable(D7)

Use this bit to turn DMA on between the Pro AudioSpectrum DMA circuitry and the PC's motherboard. When D7 is set to 1, DMA can occur. When D7 is set to 0, the Pro AudioSpectrum's DMA circuitry is logically removed from the PC bus and other boards can use a shared DMA channel without conflict. You should set this bit only when using PCM I/O.

You must set DMA Enable to 1 before programming the PC's DMA controller.

	D7	Description
Settings:	1	Enable Pro AudioSpectrum DMA.
	0	Disable Pro AudioSpectrum DMA.
Default:	0	

PCM Enable(D6)

Use this bit to enable the PCM state machine.

You must initialize the sample timer before enabling the PCM state machine. To enable the sample timer, see "Local Timer Control Register 138Bh" on page 7-8.

	D6	Description
Settings:	1	Enable PCM state machine.
	0	Disable PCM state machine.
Default:	0	

Mono/Stereo Mode (D5)

Use this bit to set the PCM circuitry to either mono or stereo mode.

In stereo mode, the first byte is left channel data and the second byte is right channel data. You can determine whether the next byte is from the left or right channel by reading the interrupt status register at address B89h.

	D5	Description
Settings:	1	Set PCM to stereo mode.
	0	Set PCM to mono mode.
Default:	0	

DAC/ADC Mode (D4)

Use this bit to set the PCM circuitry to either output (DAC) or input (ADC) mode.

	D4	Description
Settings:	1	Enable output (DAC) mode.
	0	Enable input (ADC) mode.
Default:	0	

Left to Left (D3)

Use this bit to connect the left output channel of the Input Mixer to the left input channel of the Output Mixer. For a complete description of Input and Output Mixer interactions, see Chapter 20, "Mixer Programming Essentials."

	D3	Description
Settings:	1	Connect left output to left input.
	0	Disabled.
Default:	0	

Right to Left (D2)

Use this bit to connect the right output channel of the Input Mixer to the left input channel of the Output Mixer. For a complete description of Input and Output Mixer interactions, see Chapter 20, "Mixer Programming Essentials."

	D2	Description
Settings:	1	Connect right output to left input.
	0	Disabled.
Default:	0	

Left to Right (D1)

Use this bit to connect the left output channel of the Input Mixer to the right input channel of the Output Mixer. For a complete description of Input and Output Mixer interactions, see Chapter 20, "Mixer Programming Essentials."

	D1	Description
Settings:	1	Connect left output to right input.
	0	Disabled.
Default:	0	

Right to Right (D0)

Use this bit to connect the right output channel of the Input Mixer to the right input channel of the Output Mixer. For a complete description of Input and Output Mixer interactions, see Chapter 20, "Mixer Programming Essentials."

	D0	Description
Settings:	1	Connect right output to right input.
	0	Disabled.
Default:	0	

Sample Rate Timer Register 1388h

Use the Sample Rate Timer to set the interval for processing PCM samples. The Pro AudioSpectrum processes a sample and then waits a specific period of time before processing the next one. You use the Sample Rate Timer Register to specify this period of time.

Program all 16 bits of this register by writing a value representing the interval between samples. To determine the proper programming sequence, see "Local Timer Control Register 138Bh" on page 7-8.

For stereo sampling, load the timer with an interval value half that of the mono sample rate, since twice the number of samples are required per sound.

Calculate interval values as follows:

```
Interval = 1,193,180 / sample rate
```

For example, a 22 kHz sample rate would calculate as follows:

```
Interval = 1,193,180/22,050 or

Interval = 54
```

Note: Before setting the sample rate interval, be sure to select the Sample Rate Timer using Local Timer Control Register 138Bh. Also remember to set the Sample Rate Timer Gate of the Audio Filter Control Register B8Ah to 0 before programming the timer.

D7	D6	D5	D4	D3	D2	D1	D0
Interval Bit 7	Interval Bit 6	Interval Bit 5	Interval Bit 4	Interval Bit 3	Interval Bit 2	Interval Bit 1	Interval Bit 0

Sample Buffer Count Register 1389h

Use the Sample Buffer Count register to set number of bytes in the DMA buffer division. This register holds a 16-bit value. When using a 16-bit DMA channel, the Sample Buffer Count must be divided by two. For example, to set up a 2K DMA buffer division, you must program a count of 1K into the counter. Once the counter decrements to zero, a Sample Buffer Count Interrupt is generated and enabled.

D7	D6	D5	D4	D3	D2	D1	D0
Count Bit 7	Count Bit 6	Count Bit 5	Count Bit 4	Count Bit 3	Count Bit 2	Count Bit 1	Count Bit 0

Note: To determine the proper programming sequence for this register, see "Local Timer Control Register 138Bh" on page 7-8. The Sample Buffer Counter Gate in the filter register (B8Ah) must be set to zero before programming the counter.

Local Speaker Timer Count Register 138Ah

Use the Local Speaker Timer Count Register to control the speaker timer on the Pro AudioSpectrum that shadows the PC speaker timer. When the Pro AudioSpectrum is installed, PC speaker sounds are generated from the Local Speaker Timer Count Register rather than from the PC.

Program all eight bits of this register at once by writing a value representing the speaker count value. This timer is a PC-compatible 8253 timer that can be programmed in all six modes. For more information, on programming the 8253, contact Intel for product literature.

D7	D6	D5	D4	D3	D2	D1	D0
Speaker Count Bit 7	Speaker Count Bit 6	Speaker Count Bit 5	Speaker Count Bit 4	Speaker Count Bit 3	Speaker Count Bit 2	Speaker Count Bit 1	Speaker Count Bit 0

Local Timer Control Register 138Bh

Use the Local Timer Control Register to enable and disable timers; set read and write modes; set timer generator modes; and select binary or BCD counting modes.

D7	D6	D5	D4	D3	D2	D1	D0
Timer Select Bit 1	Timer Select Bit 0	Read/Write Bit 1	Read/Write Bit 0	Timer Mode Bit 2	Timer Mode Bit 1	Timer Mode Bit 0	Binary/BCD

Timer Select (D7 and D6)

Use these two bits to select the Sample Rate Counter, Sample Buffer Counter, or the Local Speaker Timer Counter. You must enable these functions before programming other PCM registers.

	D7	D6	Description
Settings:	0	0	Select Sample Rate Timer.
	0	1	Select Sample Buffer Counter.
	1	0	Select Local Speaker Timer Counter.
Default:	N/A	N/A	

Read/Write (D5 and D4)

Use these two bits to enable the 16-bit sample timer. In 16-bit mode, read or write the least significant byte first, then the most significant byte.

	D5	D4	Description
Settings:	1	1	Enable 16-bit timer.
Default:	0	0	

Timer Mode(D3 though D1)

Use these three bits to select a timer mode. Timer Mode must be matched to the timer you have enabled:

- For Sample Rate Timer, set Timer Mode *to Square Wave Generator*
- For Sample Buffer Count, set Timer Mode to *Rate Generator*

	D3	D2	D1	Description
Settings:	0	1	0	Select Rate Generator (Sample Rate Timer).
	0	1	1	Select Square Wave Generator (Sample Buffer Count).
Default:	0		0	

Binary/BCD (D0)

Use this bit to set the timer counting mode to binary or BCD. This bit should normally be set to binary mode.

	D0	Description
Settings:	0	Set counting mode to binary.
		Set counting mode to BCD.
Default:	0	

Sample Size Configuration Register 8389h

Use the Sample Size Configuration Register to select output sample compression ratios and 8- or 16-bit audio.

D7	D6	D5	D4	D3	D2	D1	D0
Reserved	Reserved	Reserved	Reserved	Reserved	8/16-bit Audio	Over Sampling Bit 1	Over Sampling Bit 0

Reserved (D7 through D3)

These bits should not be programmed by application developers. Modifying any of these bits will result in unpredictable behavior by the Pro AudioSpectrum.

8/16-bit Audio (D2)

Use this bit to select either 8- or 16-bit audio. Choosing 16-bit audio results in better sound quality.

	D0	Description
Settings:	0	Set audio byte size to 8-bit.
	1	Set audio byte size to 16-bit.
Default:	0	

Note: Not all Pro AudioSpectrum products support 16-bit audio. To determine if your system supports 16-bit audio, use the `MVGetHWVersion` function call described in "MVGetHWVersion" on page ~~2-12~~. 3-2

Over Sampling(D1 and D0)

Use these bits to select over -sampling rates of 1, 2, or 4 times.

	D1	D0	Description
Settings:	0	0	Set over sampling rate to 1X.
	0	1	Set over sampling rate to 2X.
	1	1	Set over sampling rate to 4X.
Default:	0	0	

FM Programming Section

8 FM Synthesizer Programming Essentials

This chapter describes:

- The standard features and capabilities of the Pro AudioSpectrum's stereo FM synthesizer
- Pro AudioSpectrum channel mode configurations
- Operator connection modes including *FM synthesis* (serial connection) and *additive synthesis* (parallel connection)
- Understanding operator cell and channel number
- Selecting synthesizer access mode (stereo or mono)
- Reading and writing to hardware registers
- Programming strategy

For information on FM synthesizer function calls, see Chapter 9, "Low-Level FM Synthesizer Function Call Reference." For information on programming the standard FM synthesizer hardware registers, see Chapter 11, "Standard FM Synthesizer Register Functions." For information on the enhanced FM synthesizer hardware registers, see Chapter 12, "Enhanced FM Synthesizer Register Functions."

Pro AudioSpectrum FM synthesizer capabilities

The Pro AudioSpectrum's stereo FM synthesizer can produce complex musical waveforms while introducing very little PC processor overhead. Unlike PCM waveform generation, in which you must simulate a complex waveform, FM synthesis uses pre-programmed counters and waveforms to generate sound.

The Pro AudioSpectrum stereo FM synthesizer is based on Yamaha chips (either dual 3812's or a single OPL3 with stereo synthesizer capability) to simulate the timbre of musical instruments. Timbre, or sound quality, results from the complexity of sound harmonics. The Pro AudioSpectrum's frequency modulation technique lets you create rich harmonics and musical sounds by programmatically controlling a few simple parameters.

The Pro AudioSpectrum can be programmed to produce up to 20 voices. Both FM synthesizers on the Pro AudioSpectrum generate music as a monaural audio source. You produce stereo music by setting the board to stereo synthesizer access mode and programming the two synthesizers independently.

The key features of the FM synthesizers include:

- Programmatic control over individual operators, operator pairs, and all operators at once
- Control over a wide range of parameters for creating sound with rich texture:

  Frequency

  Frequency multiplier

  Envelope type (percussive or non-percussive)

  Envelope amplitude (total level)

  Envelope key scaling level (KSL)

  ADSR (attack/decay/sustain/release) rate

  Key scaling rate (KSR)

  Sustain amplitude level

  Feedback (modulating operator only)

  Tremolo depth and enable

  Vibrato depth and enable

  Waveform selection (eight different waveforms)

Low-level FM synthesizer software API programming steps

The following procedure shows the order in which you should use low- level API function calls. The function calls are documented in Chapter 9, "Low-Level FM Synthesizer Function Call Reference."

1. **Initialize the function call library and the Pro AudioSpectrum hardware.**

Function call:	`mvInitFMMode`

2. **Write FM data**

Function calls:	`mvOutDual3812` or
	`mvOutLeft3812` or
	`mvOutRight3812`

Note: Although you may use the New bit of the Select OPL3 hardware register to split the FM synthesizer, in order to ensure compatibility with all Pro AudioSpectrum models, Media Vision recommends that you use the low-level software API to perform this function.

FM synthesizer channel modes

The stereo FM synthesizer can be configured in two different ways:

- Two- operator mode

  This mode provides either 18 melody voices or 12 melody and six rhythm voices.

- Four- operator mode

  This mode provides: six four-operator melody voices and six two-operator melody voices; or six four-operator melody voices, three two-operator voices, and five rhythm voices.

For information on programming the FM synthesizer's the different channel modes, see the following section and Chapter 12, "Enhanced FM Synthesizer Register Functions."

Operator connection modes

The FM synthesizer operators are combined to create channels (also known as voices). The FM synthesizer of the Pro AudioSpectrum Plus, Pro AudioSpectrum 16, and the CDPC support eight different connection modes. Earlier Pro AudioSpectrum systems support two connection modes: FM synthesis (serial connection) and additive synthesis (parallel connection). These two connection modes are described here.

FM synthesis (serial connection)

In FM synthesis (serial connection), a sine wave carrier frequency is modulated by a second sine wave signal at the same or closely related frequency. Both of these frequencies, as well as many of the partial frequencies (harmonics) that are created, are within the audible range. Since it contains many harmonics, the resulting sound spectrum is complex and creates notes that are rich in timbre.

Harmonics are predictable because they appear in accordance with well-known formulae, called Bessel functions. Bessel functions describe the combination of two sine waves by modulation.

The FM synthesis technique requires that two cells be connected in series, as shown in the figure below:

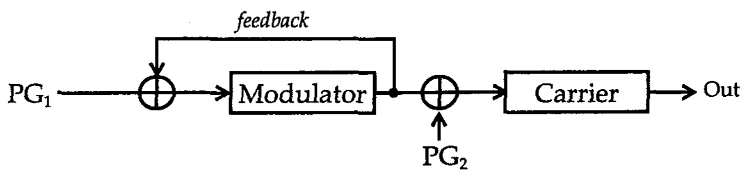


Figure 1 FM Synthesis (Serial Connection)

The output of the carrier cell (which generates the base frequency) is *modulated*, or altered, by the modulator cell. The modulator cell imparts a richness in timbre when it modulates the steady frequency produced by the carrier cell.

Each operator cell has three components:

- Phase generator
- Envelope generator
- Sine table

The diagram below depicts the characteristics of operator cells.

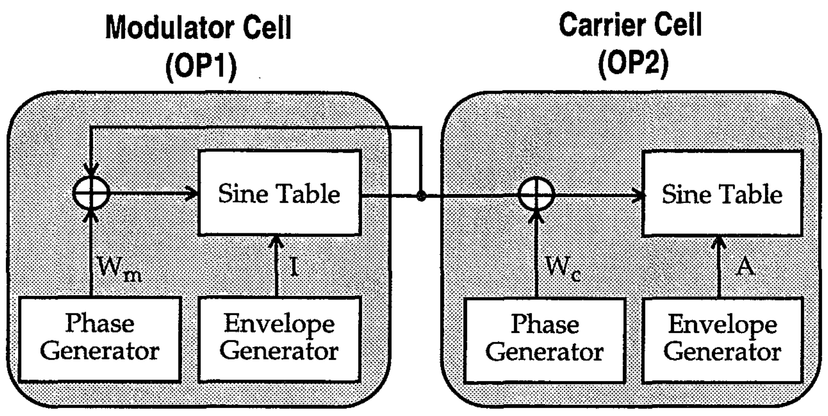


Figure 2 Operator Cell Components

Each operator creates sine waves independently. The frequency of each sine wave is controlled by the phase generator; the amplitude is controlled by the envelope generator.

Operators are combined in series. The output of the modulator cell alters the carrier cell. The waveform that results from the output of the carrier cell contains the fundamental frequency of the carrier cell, plus harmonics that result from the interaction of the two cells. The harmonics are at frequencies equal to the carrier frequency, plus and minus integer multiples of the modulator frequency.

The relative strength of the harmonics depends on the amplitude of the modulator cell output. By changing the frequency and amplitude of the modulator cell, while keeping the frequency of the carrier cell constant, you can dramatically change the timbre of the FM synthesized sound created by these two operators.

The formula below explains the interaction of the four parameters, which you control programmatically, that control FM sound production:

$$F(t) = A \sin(w_c t + I \sin(w_m t))$$

A	output amplitude
I	modulation index (modulator cell amplitude)
w_c	carrier cell frequency
w_m	modulator cell frequency

See Figure 2 for a visual representation of this formula.

Additive synthesis (parallel connection)

With additive synthesis (parallel connection), the output equals the sum of the two operators. Operator 1 optionally has feedback, which is useful for creating interesting harmonics. The figure below illustrates the interaction of cells operating in parallel.

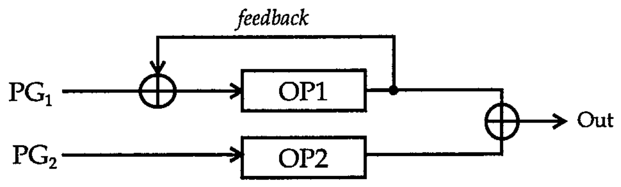


Figure 3 Additive Synthesis

The formula below describes composite sine wave synthesis:

$F(t) = A_1 \sin(w_1 t) + A_2 \sin(w_2 t)$

A_1	cell 1 amplitude
w_1	cell 1 frequency
A_2	cell 2 amplitude
w_2	cell 2 frequency

Understanding operator cell number and channel number

The FM synthesizer of early Pro AudioSpectrum systems can be configured for up to nine FM channels (also known as *voices*). Each channel is produced by combining a pair of operator cells.

When the FM synthesizer is set to melody mode, all 18 operators are dedicated to the nine FM channels. To set the FM synthesizer to melody mode, set CSM Mode/Keyboard Split Register (08h) bit D7 = 0 and Depth/Percussion/ Instruments Register (0BDh) bit D5 = 0.

In percussion mode, operators 1 through 12 are paired to create six FM channels; operators 13 through 18 produce five percussion sounds, including bass drum and high hat. For more information on enabling percussion mode and percussion instruments, see "Depth / Percussion / Instruments Register BDh" on page 11-20.

The table below shows how operator cells are paired to create channels when the synthesizer is in melody mode. Remember, in percussion mode, operator cells 13 through 18 are reserved for percussion instruments.

Operator	Channel	Offset	Function
1	1	00	modulator
2	2	01	modulator
3	3	02	modulator
4	1	03	carrier
5	2	04	carrier
6	3	05	carrier
--	--	06	scratch register
--	--	07	scratch register
7	4	08	modulator
8	5	09	modulator
9	6	0A	modulator
10	4	0B	carrier
11	5	0C	carrier
12	6	0D	carrier
--	--	0E	scratch register
--	--	0F	scratch register
13	7	10	modulator*
14	8	11	modulator*
15	9	12	modulator*
16	7	13	carrier*
17	8	14	carrier*
18	9	15	carrier*

Table 1 Operator Cell Pairings To Create Channels

* Used for either FM synthesis (in melody mode) or percussion sound production (in percussion mode)

Caution: Notice the gaps in the address map at 06h and 07h, and at 0Eh and 0Fh.

Note: The Key On bits (D5) of Block and F-Number Registers B6h, B7h, and B8h must be set to 0 before enabling percussion instruments.

Synthesizer sound modes

The Pro AudioSpectrum's stereo FM synthesizers support two sound modes:

- Monaural
- Stereo

When powered on, the Pro AudioSpectrum comes up in monaural mode. In monaural mode, both FM synthesizers are addressed at ports 388h and 389h.

In stereo mode, the left FM synthesizer is addressed at 388h and 389h. The right FM synthesizer is addressed at 38Ah and 38Bh.

Changing sound modes

Use the `mvInitFMMode` function to change sound modes between monaural and stereo. For information on this function call, see "mvInitFMMode" on page 9-2.

After writing to an *FM synthesizer address and status* port, wait 3.35 microseconds before writing to the corresponding *FM synthesizer data* port. Execute five IN instructions to read the FM Status port following your OUT instruction.

After writing to an *FM synthesizer data* port, wait 23.75 microseconds before writing to the corresponding *FM synthesizer address and status* port. Execute 33 IN instructions to read the FM Status port following your OUT instruction.

These sequences provide the correct delays for the older Pro AudioSpectrum systems. The Pro AudioSpectrum Plus, Pro AudioSpectrum 16, and the CDPC require only 2 IN operations after each write to the address and status or data ports.

Programming strategy

The following procedure integrates the concepts presented in this chapter. Follow the steps below to produce computer synthesized music.

1. **Identify the channel and operators you want to program.**
2. **Initialize.**

   Select FM synthesis mode (FM synthesis or additive synthesis) and melody or percussion mode.
3. **Determine the note you want to produce.**

   Load the frequency and octave information into the appropriate registers.

4. **Start playing notes.**

   Toggle the *Key On* bit for the channel and operators you've chosen.

5. **Wait until all notes have been played and start again.**

Note: If you follow the procedure to produce a note, but don't produce the one you want, check the ADSR setting, frequency, octave scale, connection bit, frequency multiplier, and CSM mode/keyboard split, in that order.

Note: The design of the the early Pro AudioSpectrum systems requires that you wait 3.3 microseconds after writing to register 38Ah and 23 microseconds after writing to register 38Bh before issuing a second command to the FM synthesizer. Newer Pro AudioSpectrum systems (Plus, 16, or CDPC) require a 2 microsecond delay.

9 Low-Level FM Synthesizer Function Call Reference

This chapter describes the low-level function calls you can use to program the Pro AudioSpectrum's FM synthesizer. To understand the fundamentals of programming the FM synthesizers, see Chapter 10, "FM Synthesizer Hardware I/O Ports." For a reference to the basic FM synthesizer register functions, see Chapter 11, "Standard FM Synthesizer Register Functions." For a reference to more advanced FM synthesizer register functions, see Chapter 12, "Enhanced FM Synthesizer Register Functions."

You must start your FM synthesizer programs by calling the `mvInitFMMode` routine. You can find prototypes of each of the function calls listed in this chapter in the `3812A.ASM` file.

mvInitFMMode

Use `mvInitFMMode` to set the synthesizer to stereo or mono mode.

You must call this routine before using any of the other FM synthesizer function calls.

Calling Convention

```
int mvInitFMMode (int mode);
```

Input Parameters

Parameter	Type	Value	Description
mode	integer	0	Set synthesizer mode to mono.
		1	Set synthesizer mode to stereo.

Return Values

Parameter	Type	Value	Description
int	integer	0	Initialization succeeded.
		-1	Initialization failed. Hardware not found.

Related topics

None.

mvOutDual3812

Use `mvOutDual3812` to write FM synthesizer register commands and data both the left and right FM synthesizers.

Calling Convention

```
void mvOutDual3812 (int addr, int leftdata, int rightdata);
```

Input Parameters

Parameter	Type	Value	Description
addr	integer		Specifies the address of an FM register function.
leftdata	integer		Data to be sent to a left FM synthesizer device.
rightdata	integer		Data to be sent to a right FM synthesizer device.

Return Values

None.

Related topics

To initialize the FM synthesizer environment, see "mvInitFMMode" on page 9-2.

For a reference to the basic FM synthesizer register functions, see Chapter 11, "Standard FM Synthesizer Register Functions."

For a reference to more advanced FM synthesizer register functions, see Chapter 12, "Enhanced FM Synthesizer Register Functions."

mvOutLeft3812

Use `mvOutLeft3812` to write FM synthesizer register commands and data to the left FM synthesizer (port addresses 388h and 389h). If the Pro AudioSpectrum is in FM mono mode, `mvOutLeft3812` will write data to the left and right FM synthesizers simultaneously.

Calling Convention

```
  void mvOutLeft3812 (int addr, int leftdata);
```

Input Parameters

Parameter	Type	Value	Description
addr	integer		Specifies the address of an FM register function.
leftdata	integer		Specifies data to be sent to a left FM synthesizer device.

Return Values

None.

Related topics

To initialize the FM synthesizer environment, see "mvInitFMMode" on page 9-2.

For a reference to the basic FM synthesizer register functions, see Chapter 11, "Standard FM Synthesizer Register Functions."

For a reference to more advanced FM synthesizer register functions, see Chapter 12, "Enhanced FM Synthesizer Register Functions."

mvOutRight3812

Use `mvOutRight3812` to write FM synthesizer register commands and data to the right FM synthesizer (port addresses 38Ah and 38Bh). This function always sends data to the right FM synthesizer, regardless of the FM -synthesizer -split -mode state.

Calling Convention

```
void mvOutRight3812 (int addr, int rightdata);
```

Input Parameters

Parameter	Type	Value	Description
addr	integer		Specifies the address of an FM register function.
rightdata	integer		Specifies data to be sent to a right FM synthesizer device.

Return Values

None.

Related topics

To initialize the FM synthesizer environment, see "mvInitFMMode" on page 9-2.

For a reference to the basic FM synthesizer register functions, see Chapter 11, "Standard FM Synthesizer Register Functions."

For a reference to more advanced FM synthesizer register functions, see Chapter 12, "Enhanced FM Synthesizer Register Functions."

10 *FM Synthesizer Hardware I/O Ports*

This chapter describes the hardware programming interface to the stereo FM synthesizer.

Unlike the other hardware programming interfaces of the Pro AudioSpectrum where you read and write data to many individual ports, the FM synthesizer uses the concept of an index. In this scheme, only four port addresses handle all interactions with the FM synthesizer chip. The following section describes this concept in detail.

For information on standard FM registers and register values, see Chapter 11, "Standard FM Synthesizer Register Functions." For information on enhanced FM synthesizer registers and register values, see Chapter 12, "Enhanced FM Synthesizer Register Functions."

FM synthesizer I/O addresses

The following table describes the channel I/O addresses (ports) used to program the FM synthesizer.

Channel	I/O address(es)	Description
Left	388h	*Left FM synthesizer address and status* port. Used for two purposes: To select the register address and to read the FM synthesizer status register.
Left	389h	*Left FM synthesizer data* port. Used to write data to the selected register.
Right	38Ah	*Right FM synthesizer address and status* port. Used for two purposes: To select the register address and to read the FM synthesizer status register.
Right	38Bh	*Right FM synthesizer data* port. Used to write data to the selected register.

In monaural mode, both sides of the stereo synthesizer is addressed at 388h and 389h and function like as a one channel device. Monaural mode ensures compatibility with AdLib and Sound Blaster, and with older sound boards that have a single FM synthesizer. The FM synthesizer also supports an alternative set of left channel addresses at addresses 2x8h and 2x9h for Sound Blaster compatibility.

Reading and writing to FM synthesizer ports

The bits of the FM synthesizer address and status port contains different types of data depending upon whether the port is being read or written to.

When reading this register, it returns the following data:

D7	D6	D5	D4	D3	D2	D1	D0
Interrupt Request	Timer 1 Flag	Timer 2 Flag	Not Used	Not Used	Not Used	Not Used	Not Used

Figure 4 Data Returned From Reading The FM Synthesizer Address and Status Port (388h or 2x8h and 38Ah)

Use this port to determine the current state of the FM synthesizer Interrupt line and Timers. When either Timer overflows, the corresponding Timer Flag (D5 or D6) and the Interrupt Request (D7) is set to 1. For more information on handling interrupts, see Appendix E, "Programming the PC's Interrupt Controller.".

The FM synthesizer address and status port accepts the following data:

D7	D6	D5	D4	D3	D2	D1	D0
FM Address Select	FM Address Select	FM Address Select	FM Address Select	FM Address Select	FM Address Select	FM Address Select	FM Address Select

Figure 5 Data to Write to the FM Synthesizer Address and Status Port (388h or 2x8h and 38Ah)

Use this port to specify FM registers to load when your program writes to the FM synthesizer data port.

The FM synthesizer data port is used to load (write) sound generation parameters into the FM synthesizer. The table below shows a bit map of FM synthesizer data port.

D7	D6	D5	D4	D3	D2	D1	D0
FM Register Data	FM Register Data	FM Register Data	FM Register Data	FM Register Data	FM Register Data	FM Register Data	FM Register Data

Figure 6 FM Synthesizer Data Port (389h or 2x9h and 38Bh)

Use this port to load FM registers with register values.

11 *Standard FM Synthesizer Register Functions*

This chapter provides a complete description of the standard FM synthesizer registers and register functions. All Pro AudioSpectrum systems support the registers and functions described here. The newest Pro AudioSpectrum models like the Pro AudioSpectrum Plus, Pro AudioSpectrum 16, and the CDPC also support enhanced FM synthesizer functions. For information on enhanced FM synthesizer registers and functions, see Chapter 12, "Enhanced FM Synthesizer Register Functions.".

Test Register 01h

This register enables the Wave Select command. The other bits of this register are reserved by Yamaha to test the FM synthesizer chip. With the exception of the Enable Wave Select bit (D5), this register should always be cleared (initialized to zero).

D7	D6	D5	D4	D3	D2	D1	D0
Test	Test	Enable Wave Select	Test	Test	Test	Test	Test

Test (D7, D6 and D4 to D0)

Used by Yamaha to test the synthesizer chip. These bits should be set to 0.

	Dn	Description
Settings:	1	Reserved, do not use.
	0	Clear test bits.
Default:	0	

Enable Wave Select (D5)

Use this bit to enable the Wave Select command (registers E0h to F5h). For information on the Wave Select command, see "Waveform Registers E0h to F5h" on page -24.

	D5	Description
Settings:	1	Turn Wave Select on.
	0	Turn Wave Select off.
Default:	0	

Note: Use this register with 3812-based Pro AudioSpectrum systems only. This bit disrupts the operation of OPL3-based systems (Pro AudioSpectrum Plus, Pro AudioSpectrum 16, and the CDPC).

Timer 1 Register 02h

Use this register to set a value for Timer 1. Timer 1 is an 8-bit, presettable counter with a resolution of 80 microseconds.

D7	D6	D5	D4	D3	D2	D1	D0
Timer 1 Bit 7	Timer 1 Bit 6	Timer 1 Bit 5	Timer 1 Bit 4	Timer 1 Bit 3	Timer 1 Bit 2	Timer 1 Bit 1	Timer 1 Bit 0

The Timer 1 interval is calculated as shown below:

$$T1_{overflow} = (256-N) * 80 \text{ microseconds}$$

where N = Timer 1 register value

	Dn	Description
Settings:	1	Set bits on to indicate greater Timer 1 value.
	0	Set bits off to indicate smaller Timer 1 value.
Default:	N/A	

Timer 2 Register 03h

Use this register to set Timer 2. Timer 2 is identical to Timer 1 except that its resolution is 320 microseconds.

D7	D6	D5	D4	D3	D2	D1	D0
Timer 2 Bit 7	Timer 2 Bit 6	Timer 2 Bit 5	Timer 2 Bit 4	Timer 2 Bit 3	Timer 2 Bit 2	Timer 2 Bit 1	Timer 2 Bit 0

The timer 2 interval is calculated as shown below:

$$T2_{overflow} = (256-N) * 320 \text{ microseconds}$$

where N = Timer 2 register value

	Dn	Description
Settings:	1	Set bits on to indicate greater Timer 2 value.
	0	Set bits off to indicate smaller Timer 2 value.
Default:	N/A	

Control Timer Register 04h

This register starts, stops, and masks the status flags for Timer 1 and Timer 2.

D7	D6	D5	D4	D3	D2	D1	D0
IRQ Reset	Mask T1	Mask T2	--	--	--	ST2	ST1

Reset IRQ (D7)

Use this bit to clear Status Register flags for IRQ, Timer 1, and Timer 2. This bit automatically resets itself to 0 after flags are cleared.

	D7	Description
Settings:	1	Clear status register flags.
	0	Inactive.
Default:	0	

Note: See "Synthesizer sound modes" on page 8-8 for an explanation of how the board responds to the FM synthesizer IRQ and Timer status flags.

Mask Timer 1 (D6)

Use this bit to prevent the Timer 1 Status flag (D5 of Test and Status Register 01h) from being set when Timer 1 overflows.

Because the Media Vision systems has its own interrupt enable bits for FM synthesis, using this mask is not recommended.

	D6	Description
Settings:	1	Mask Timer 1 flag.
	0	Inactive.
Default:	0	

Mask Timer 2 (D5)

Use this bit to mask the Timer 2 flag. This function of this bit is identical to the Mask Timer 1 bit.

	D5	Description
Settings:	1	Mask Timer 1 flag.
	0	Inactive.
Default:	0	

Start or Stop Timer 2 (D1)

Use this bit to load the Timer 2 register value and begin counting. When this bit is set to 0, Timer 2 is stopped and reset.

	D1	Description
Settings:	1	Load register value into Timer 2 and begin counting.
	0	Stop Timer 2.
Default:	0	

Start or Stop Timer 1 (D0)

Use this bit to load the Timer 1 register value and begin counting. When this bit is set to 0, Timer 1 is stopped and reset.

	D0	Description
Settings:	1	Load register value into Timer 1 and begin counting.
	0	Stop Timer 1.
Default:	0	

CSM Mode / Keyboard Split Register 08h

This register controls whether the synthesizer operates in music mode or composite speech mode (CSM), and the location of the keyboard split for keyboard rate scaling.

D7	D6	D5	D4	D3	D2	D1	D0
CSM	Keyboard Split (SEL)	-	-	-	-	-	-

Composite Sine Wave Mode (D7)

Use this bit to toggle the synthesizer between music mode and composite speech mode.

In music mode, the FM synthesizer can be programmed to operate in one of two sub-modes: melody mode (9 FM channels) or percussion mode (6 FM channels and 5 percussion sounds). Melody mode is the default configuration.

All 18 operators must be turned off (Key On = 0) before you switch to the composite speech mode. Composite speech sound is produced by momentarily switching to Key On.

	D7	Description
Settings:	1	Set synthesizer to composite speech mode.
	0	Set synthesizer to music mode.
Default:	0	

Note: There is no functional use for composite speech mode. This function is not supported on the Pro AudioSpectrum Plus, Pro AudioSpectrum 16, or the CDPC.

Keyboard Split Point or SEL (D6)

Use this bit to control the split point of the keyboard.

Note that the FM synthesizer *definition* of keyboard split is different from the standard music definition. In the context of the FM synthesizer, keyboard split refers to the point where keyboard *scaling occurs within each octave.*

The *Split Number* controls the amount of key scaling applied to a note. As outlined in the table below, the *Split Number* is a function of both the octave (block number) and the split point within each octave. For both split point settings (SEL = 0 or 1), key scaling increases by a factor of two as the octave (block number) increases. An exception to this rule occurs at the split point within an octave. Beyond the split point, key scaling increases by 1.

Octave	0		1		2		3		4		5		6		7	
Block Data	0		1		2		3		4		5		6		7	
F-Num MSB	1		1		1		1		1		1		1		1	
F-Num 2nd	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Split Number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Octave	0		1		2		3		4		5		6		7	
Block Data	0		1		2		3		4		5		6		7	
F-Num MSB	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
F-Num 2nd	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Split Number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

For more information on key scaling, see "AM/VIB/EG/KSR/Multiple Registers 20h to 35h" on page 11-7.

For both SEL bit settings, the keyboard split point occurs in the middle of each octave. Note that the most significant bit changes from 0 to 1 in the middle of the octave.

When set to 1, keyboard scaling increases smoothly (in jumps of 2) from octave to octave, and increases by 1 within each octave. Keyboard scaling increases in a roughly linear fashion across the entire keyboard. When set to 0, notes of the lower half of the keyboard have no scaling while notes of the upper half scale smoothly (in jumps of 2).

Since the split number influences the note's attack/decay/release rate, SEL=1 results in a natural sounding scaling effect while SEL=0 scaling provides a special effect.

	D6	Description
Settings:	1	Scale keyboard by jumps of 2 from octave to octave.
	0	Do not scale lower half of keyboard, scale by jumps of 2 for upper half.
Default:	0	

AM/VIB/EG/KSR/Multiple Registers 20h to 35h

This register group controls operator characteristics that produce changes in timbre. One register is dedicated to each of the operators.

D7	D6	D5	D4	D3	D2	D1	D0
Tremolo (AM)	Vibrato	Envelope (EG) Type	Key Scaling Rate (KSR)	Multiple Bit 3	Multiple Bit 2	Multiple Bit 1	Multiple Bit 0

Tremolo Modulation (D7)

Use this bit to control the tremolo (amplitude) modulation of an operator.

The tremolo frequency is 3.7 Hz and the modulation depth is either 7% or 14%, depending on the setting of the Tremolo Depth (bit D7) in register BDh.

	D7	Description
Settings:	1	Apply amplitude modulation.
	0	No amplitude modulation.
Default:	0	

Vibrato Modulation (D6)

Use this bit to control the vibrato modulation of an operator.

The vibrato is a frequency of 6.4 Hz (twice that of the tremolo frequency) and the modulation depth is either 4.8 dB or 1 dB, depending on the setting of the Vibrato Depth (bit D6) in register at BDh.

	D6	Description
Settings:	1	Apply vibrato modulation.
	0	No vibrato modulation.
Default:	0	

Envelope Type (D5)

Use this bit to control whether the sound has the envelope shape of a diminishing sound or a continuing sound.

Diminishing sounds are sometimes referred to as *percussive* sounds while continuing sounds are called *non-percussive*. The figure below shows the two envelope types.

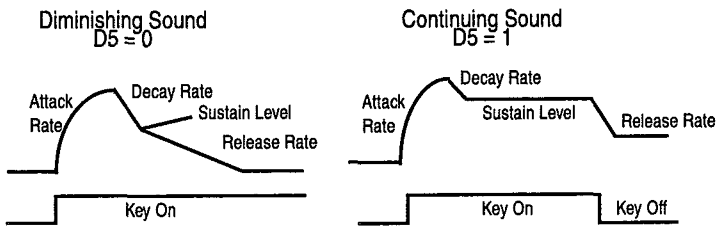


Table 2 Envelope Types

	D5	Description
Settings:	1	Set envelope shape to continuing sound.
	0	Set envelope shape to diminishing sound.
Default:	0	

Key Scaling Rate (D4)

Use this bit to control the rate of key scaling in order to imitate string instruments. Scaling choices are minimal or maximal scaling.

To the human ear, notes of string instruments appear to shorten at higher frequencies. Key scaling lets you set the rate of attack, decay, and release of a note. A higher (more vertical) rate of attack, decay, and release results in a gradual shortening of the envelope length as higher frequency notes are played. The figure below illustrates this phenomenon.

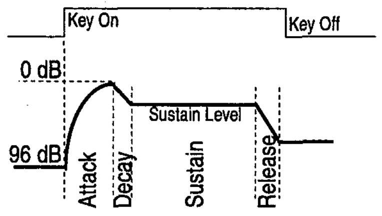


Table 3 Envelope Waveform

Using values from the table below, use this formula to calculate the actual rate:

```
Actual_rate = 4 * Unscaled_rate + KSR_adjustment
```

where			
	Unscaled_rate	=	Unscaled attack/decay/release rate (Keyboard Split Number)
	KSR_adjustment	=	KSR adjustment value from table

Unscaled_rate (keyboard split #)	KSR_adjustment	
	Minimal D4=0	**Maximal D4=1**
0	0	0
1	0	1
2	0	2
3	0	3
4	1	4
5	1	5
6	1	6
7	1	7
8	2	8
9	2	9
10	2	10
11	2	11
12	3	12
13	3	13
14	3	14
15	3	15

Table 4 Key Scaling Rate/Keyboard Split Number Cross Reference

	D4	Description
Settings:	1	Set key scaling rate to maximum.
	0	Set key scaling rate to minimum.
Default:	0	

Multiple Bits or MUL (D3 - D0)

Use the MUL bits to switch from the fundamental frequency of a note to a nearby harmonic frequency.

To understand the effect of the MUL bits, the formula below shows FM synthesis, this time with the multiplication factor explicitly included:

$$F(t) = A \sin (m_c w_c t + I \sin (m_m w_m t))$$

where	A	=	output amplitude
	m_c	=	multiplication factor for carrier
	w_c	=	frequency for carrier cell
	m_m	=	multiplication factor for modulator
	w_m	=	frequency for modulator cell

	D3	D2	D1	D0	Description
Settings:	1	1	1	1	Set multiplication factor to 15.
	1	1	1	0	Set multiplication factor to 15.
	1	1	0	1	Set multiplication factor to 12.
	1	1	0	0	Set multiplication factor to 12.
	1	0	1	1	Set multiplication factor to 10.
	1	0	1	0	Set multiplication factor to 10.
	1	0	0	1	Set multiplication factor to 9.
	1	0	0	0	Set multiplication factor to 8.
	0	1	1	1	Set multiplication factor to 7.
	0	1	1	0	Set multiplication factor to 6.
	0	1	0	1	Set multiplication factor to 5.
	0	1	0	0	Set multiplication factor to 4.
	0	0	1	1	Set multiplication factor to 3.
	0	0	1	0	Set multiplication factor to 2.
	0	0	0	1	Set multiplication factor to 1.
	0	0	0	0	Set multiplication factor to .5.
Default:	0	0	0	1	

KSL / Total Level Registers 40h to 55h

The KSL/Total Level registers control the operator envelope level (amplitude). By changing the operator envelope of the carrier operator, you control the output level (strength) of a note. By changing the modulator operator envelope level, relative to the carrier operator, you control the harmonic richness (timbre) of the sound. One register is dedicated to each of the operators.

D7	D6	D5	D4	D3	D2	D1	D0
KSL Bit 1	KSL Bit 0	Total Level Bit 5	Total Level Bit 4	Total Level Bit 3	Total Level Bit 2	Total Level Bit 1	Total Level Bit 0

Key Scaling Level (D7 and D6)

Use these bits to specify the rate at which the output amplitude declines from the starting level as pitch increases. Key Scaling Level is used to simulate the behavior of acoustic instruments.

	D7	D6	Description
Settings:	1	1	Set degree of attenuation to 6 dB/octave.
	0	1	Set degree of attenuation to 3 dB/octave.
	1	0	Set degree of attenuation to 1.5 dB/octave.
	0	0	Set degree of attenuation to 0 dB/octave.
Default:	0	0	Set degree of attenuation to 0 dB/octave.

Total Level (D5 to D0)

Use the Total Level bits to specify the operator strength. The output amplitude can vary from full strength to 47.25 dB attenuation (reduction). The table below illustrates the amount of attenuation controlled by each bit.

	D5	D4	D3	D2	D1	D0
Degree of Attenuation	24 dB	12 dB	6 dB	3 dB	1.5 dB	.75 dB

To determine the total amount of attenuation, add up the decibel values for each of the bits set to 1.

The carrier cell output level governs how dominant this channel is relative to other FM channels produced by the FM synthesizer. For a given carrier cell output level, varying the modulator cell output level changes the amount of modulation. The greater the modulation, the greater the number and strength of harmonics. Strong harmonics result in rich timbre.

	Dn	Description
Settings:	1	Turn bit on for attenuation.
	0	Turn bit off for attenuation.
Default:	0	

Attack / Decay Rate Registers 60h to 75h

This set of registers sets the rising and decaying times for a sound. One register is dedicated to each of the operators.

D7	D6	D5	D4	D3	D2	D1	D0
Attack Rate Bit 3	Attack Rate Bit 2	Attack Rate Bit 1	Attack Rate Bit 0	Decay Rate Bit 3	Decay Rate Bit 2	Decay Rate Bit 1	Decay Rate Bit 0

Attack Rate (D7 to D4)

Use these bits to control the attack rate for the sound. To determine the actual rate of attack, see Appendix A, "FM Hardware Register Charts and Tables."

	D7	D6	D5	D4	Description
Settings:	1	1	1	1	Set attack rate value to 15.
	1	1	1	0	Set attack rate value to 14.
	1	1	0	1	Set attack rate value to 13.
	1	1	0	0	Set attack rate value to 12.
	1	0	1	1	Set attack rate value to 11.
	1	0	1	0	Set attack rate value to 10.
	1	0	0	1	Set attack rate value to 9.
	1	0	0	0	Set attack rate value to 8.
	0	1	1	1	Set attack rate value to 7.
	0	1	1	0	Set attack rate value to 6.
	0	1	0	1	Set attack rate value to 5.
	0	1	0	0	Set attack rate value to 4.
	0	0	1	1	Set attack rate value to 3.
	0	0	1	0	Set attack rate value to 2.
	0	0	0	1	Set attack rate value to 1.
	0	0	0	0	Set attack rate value to 0.
Default:	0	0	0	0	

Decay Rate (D3 to D0)

Use these bits to control the decay rate for the sound. To determine the actual rate of decay, see Appendix A, "FM Hardware Register Charts and Tables."

	D3	D2	D1	D0	Description
Settings:	1	1	1	1	Set decay rate value to 15.
	1	1	1	0	Set decay rate value to 14.
	1	1	0	1	Set decay rate value to 13.
	1	1	0	0	Set decay rate value to 12.
	1	0	1	1	Set decay rate value to 11.
	1	0	1	0	Set decay rate value to 10.
	1	0	0	1	Set decay rate value to 9.
	1	0	0	0	Set decay rate value to 8.
	0	1	1	1	Set decay rate value to 7.
	0	1	1	0	Set decay rate value to 6.
	0	1	0	1	Set decay rate value to 5.
	0	1	0	0	Set decay rate value to 4.
	0	0	1	1	Set decay rate value to 3.
	0	0	1	0	Set decay rate value to 2.
	0	0	0	1	Set decay rate value to 1.
	0	0	0	0	Set decay rate value to 0.
Default:	0	0	0	0	

Sustain Level / Release Rate Registers 80h to 95h

This set of registers sets the sustain level for a continuing sound and the release rate for both continuing and diminishing sounds. One register is dedicated to each of the operators.

D7	D6	D5	D4	D3	D2	D1	D0
Sustain Level Bit 3	Sustain Level Bit 2	Sustain Level Bit 1	Sustain Level Bit 0	Release Rate Bit 3	Release Rate Bit 2	Release Rate Bit 1	Release Rate Bit 0

Sustain Level (D7 to D4)

Use the Sustain Level bits to specify the amplitude of a continuing sound prior to decay. The Sustain Level is specified relative to peak amplitude of the note; it can vary between 0 dB (same as the peak) and 46 dB attenuation (almost a vertical drop-off from the peak). The table below illustrates the number of sustain decibels controlled by each bit.

	D7	D6	D5	D4
Degree of Attenuation	24 dB	12 dB	6 dB	3 dB

To determine the total sustain level, add up the decibel values for each of the bits set to 1.

	Dn	Description
Settings:	1	Turn a bit on for sustain level.
	0	Turn a bit off for sustain level.
Default:	0	

Release Rate (D3 to D0)

Use the Release Rate bits to specify the the speed at which a sound fades away. Release Rate can be applied to both continuing and diminishing sounds.

For continuing sounds, the Release Rate governs the speed at which sound fades away after the transition from Key On (D5 0f Axh =1) to Key Off (D5 = 0). For diminishing sounds, the Release Rate governs the rate at which sound fades away after the Sustain Level signal level is reached. To see understand the relationship between Release Rate values and actual release times, see Appendix A, "FM Hardware Register Charts and Tables."

	D3	D2	D1	D0	Description
Settings:	1	1	1	1	Set release rate value to 15.
	1	1	1	0	Set release rate value to 14.
	1	1	0	1	Set release rate value to 13.
	1	1	0	0	Set release rate value to 12.
	1	0	1	1	Set release rate value to 11.
	1	0	1	0	Set release rate value to 10.
	1	0	0	1	Set release rate value to 9.
	1	0	0	0	Set release rate value to 8.
	0	1	1	1	Set release rate value to 7.
	0	1	1	0	Set release rate value to 6.
	0	1	0	1	Set release rate value to 5.
	0	1	0	0	Set release rate value to 4.
	0	0	1	1	Set release rate value to 3.
	0	0	1	0	Set release rate value to 2.
	0	0	0	1	Set release rate value to 1.
	0	0	0	0	Set release rate value to 0.
Default:	0	0	0	0	

Block and F-Number Registers A0h to A8h to B0h to B8h

These two groups of nine registers select the octave, and note within the octave, associated with the two operator cells that are paired in FM synthesis to form a channel. The Key On bit of register group B0h to B8h controls when notes are played.

The Block Number specifies the octave for a pair of operator cells. The 10-bit F-Number, which spans Ax and Bx registers, specifies the note within the octave.

The 10 bit F-Number is formed by combining bits D1 and D0 from the B0h to B8h register group with its matching register in the A0h to A8h group. F-Numbers are configured as follows:

- F-Number (8 lower bits) (D7 to D0 bits in Axh)
- F-Number (2 high bits) (D1 and D0 bits in Bxh)

Register Group A0h to A8h

D7	D6	D5	D4	D3	D2	D1	D0
F-Num Bit 7	F-Num Bit 6	F-Num Bit 5	F-Num Bit 4	F-Num Bit 3	F-Num Bit 2	F-Num Bit 1	F-Num Bit 0

Register Group B0h to B8h

D7	D6	D5	D4	D3	D2	D1	D0
-	-	Key On	Block Num Bit 2	Block Num Bit 1	Block Num Bit 0	F-Num Bit 9	F-Num Bit 8

F-Num (D7 to D0 of register Axh plus D1 and D0 of register Bxh)

Use these bits to set the frequency number for a note within an octave. The F-Num for a given note is the same for all octaves.

Ten bit F-Nums map into 1,024 frequency values. Many of these values extend outside an octave or specify frequencies that fall between commonly accepted pitch values. Consequently, only a relatively small number of the F-Num values make sense for a given octave. The figure below shows the F-Num for the fourth octave in both decimal and in binary:

	Freq. Octave 4	F-Number									
		Decimal Value	High		Low						
			D1	D0	D7	D6	D5	D4	D3	D2	D1	D0
C#	277.2	363	0	1	0	1	1	0	1	0	1	1
D	293.7	385	0	1	1	0	0	0	0	0	0	1
D#	311.1	408	0	1	1	0	0	1	1	0	0	0
E	329.6	432	0	1	1	0	1	1	0	0	0	0
F	349.2	458	0	1	1	1	0	0	1	0	1	0
F#	370	485	0	1	1	1	1	0	0	1	0	1
G	392	514	1	0	0	0	0	0	0	0	1	0
G#	415.3	544	1	0	0	0	1	0	0	0	0	0
A	440	577	1	0	0	1	0	0	0	0	0	1
A#	466.2	611	1	0	0	1	1	0	0	0	1	1
B	493.9	647	1	0	1	0	0	0	0	1	1	1
C	523.3	686	1	0	1	0	1	0	1	1	1	0

Table 5 F-Numbers For Octave 4

The operator frequency can be multiplied by the Multiple bits found in register group 20h through 35h to yield a set of frequencies different than those specified by the Block Number and F-Num alone. Note that while the Block Number and F-Num applies to both operators in a pair, the Multiple can be applied to the modulator, the carrier, or to both the modulator and carrier. For a complete list of standard pitch values for all notes within the eight octave range of the FM synthesizer, see Appendix A, "FM Hardware Register Charts and Tables."

	D1	D2	D7	D6	D5	D4	D3	D2	D1	D0	Description
Setting:	0	1	0	1	1	0	1	0	1	1	Set F-Num value to (C#).
	0	1	1	0	0	0	0	0	0	1	Set F-Num value to (D).
	0	1	1	0	0	1	1	0	0	0	Set F-Num value to (D#).
	0	1	1	0	1	1	0	0	0	0	Set F-Num value to (E).
	0	1	1	1	0	0	1	0	1	0	Set F-Num value to (F).
	0	1	1	1	1	0	0	1	0	1	Set F-Num value to (F#).
	1	0	0	0	0	0	0	0	1	0	Set F-Num value to (G).
	1	0	0	0	1	0	0	0	0	0	Set F-Num value to (G#).
	1	0	0	1	0	0	0	0	0	1	Set F-Num value to (A).
	1	0	0	1	1	0	0	0	1	1	Set F-Num value to (A#).
	1	0	1	0	0	0	0	1	1	1	Set F-Num value to (B).
	1	0	1	0	1	0	1	1	1	0	Set F-Num value to (C).
Default:	0	0	0	0	0	0	0	0	0	0	

Key On (D5 of register Bxh)

Use these bits to control the output of a given operator pair. There is one Key On bit for each of the nine operator pairs (in FM synthesis melody mode). For a discussion of operator cell pairs, see "Understanding operator cell number and channel number" on page 8-6.

Note: Key On bits 13 through 18 must be cleared before you switch from melody mode to percussion mode.

	D5	Description
Settings:	1	Turn channel on for operator pair.
	0	Turn channel off for operator pair.
Default:	0	

Block Number (D4 through D2 of register Bxh)

Use the Block Number bits to specify the octave you are programming. The FM synthesizer supports an eight octave range.

	D4	D3	D2	Description
Settings:	1	1	1	Set to octave 8.
	1	1	0	Set to octave 7.
	1	0	1	Set to octave 6.
	1	0	0	Set to octave 5.
	0	1	1	Set to octave 4.
	0	1	0	Set to octave 3.
	0	0	1	Set to octave 2.
	0	0	0	Set to octave 1.
Default:	0	0	0	

Depth / Percussion / Instruments Register BDh

This register group controls tremolo modulation and vibrato depth for melody instruments; toggles the FM synthesizer between melody and percussion mode; and enables percussion instruments of the FM synthesizer.

D7	D6	D5	D4	D3	D2	D1	D0
Tremolo Depth	Vibrato Depth	Percussion Mode Enable	Bass Drum Enable	Snare Drum Enable	Tom Tom Enable	Top Cymbal Enable	Hi-Hat Enable

Tremolo Depth (D7)

Use this bit to set the amount of tremolo modulation (changes to amplitude) for a given operator pair. Tremolo must be enabled (D7 of register group 20h through 35h) for this setting to take effect.

	D7	Description
Settings:	1	Set tremolo modulation to 4.8 decibels.
	0	Set tremolo modulation to 1 decibel.
Default:	0	

Vibrato Depth (D6)

Use the Vibrato Depth bit to set the amount of vibrato modulation (changes to pitch) for a given operator pair. Vibrato must be enabled (D6 or register group 20h through 35h) for this setting to take effect.

	D6	Description
Settings:	1	Set vibrato modulation to 14 percent.
	0	Set vibrato modulation to 7 percent.
Default:	0	

Percussion Mode Enable (D5)

Use this bit to turn percussion instruments on and off. You must set this bit on in order to enable individual percussion instruments.

	D5	Description
Settings:	1	Set FM synthesizer to percussion mode.
	0	Set FM synthesizer to melody mode.
Default:	0	

Percussion Instrument Enable (D4 through D0)

When FM synthesizer mode is set to percussion (D5 = 1), use these bits to enable the bass, snare drum, Tom Tom drums, Top cymbal, and the Hi-Hat cymbal. The essential characteristics of the percussion instruments are pre-programmed into the FM synthesizer, but you must still program other characteristics, such as the F-Number and Attack Rate, to control the frequency and timbre.

The table below shows FM synthesizer percussion instruments, the particular bits of register BDh that control them, and the operator cell numbers associated with them.

Percussion Instrument	Control Bit	Operator Cell Number(s)
Bass Drum	D4	13 and 16
Snare Drum	D3	17
Tom Tom	D2	15
Top Cymbal	D1	18
Hi-Hat Cymbal	D0	14

Before enabling a percussion instrument, you must turn off the corresponding Key On bits (D5 = 0) in registers B6h through B8h. These three registers control channels 7 through 9 (operators 13 through 18), which are now reserved for rhythm instrument sounds.

	Dn	Description
Settings:	1	Enable percussion instrument.
	0	Disable percussion instrument.
Default:	0	

Feedback / Connection Registers C0h - C8h

This group of registers determines the amount of operator cell feedback and whether pairs of operators are connected serially (FM synthesis) or in parallel (additive synthesis).

D7	D6	D5	D4	D3	D2	D1	D0
-	-	-	-	Feedback Bit 2	Feedback Bit 1	Feedback Bit 0	Connection

Feedback (D3 to D1)

Use these bits to set the level of feedback for the first operator cell in a channel. The FM synthesizer supports eight fixed amplitude settings.

When an operator pair is connected serially (FM synthesis), feedback applies to the *modulator* operator; when connected in parallel (additive synthesis), feedback applies to one operator only. For more information on connection modes, see "Operator connection modes" on page 8-3.

	D3	D2	D1	Description
Settings:	1	1	1	Set feedback to 4π.
	1	1	0	Set feedback to 2π.
	1	0	1	Set feedback to π.
	1	0	0	Set feedback to $\pi/2$.
	0	1	1	Set feedback to $\pi/4$.
	0	1	0	Set feedback to $\pi/8$.
	0	0	1	Set feedback to $\pi/16$.
	0	0	0	Set feedback to 0.
Default:	0	0	0	

Connection (D0)

Use this bit to determine how pairs of operator cells are connected. The 18 operator cells provided by the FM synthesizer are organized as nine pairs of operator cells which can be connected serially (for FM synthesis) or in parallel (for additive synthesis). The figures below illustrate the the differences between FM synthesis and additive synthesis.

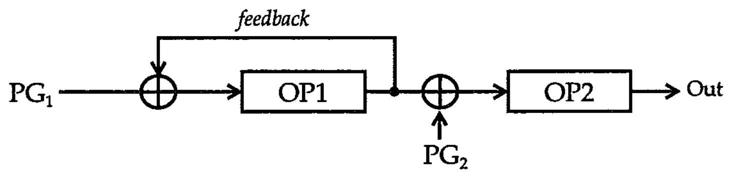


Figure 7 FM Synthesis (Serial) Connection

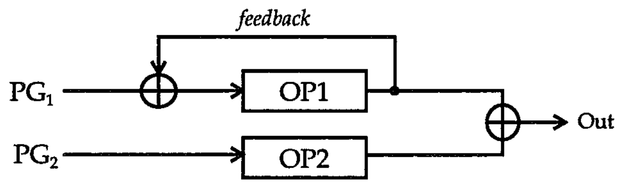


Figure 8 Additive Synthesis (Parallel) Connection

	D0	Description
Settings:	1	Set connection to additive synthesis (parallel).
	0	Set connection to FM synthesis (serial).
Default:	0	

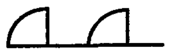
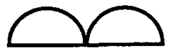
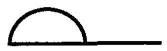
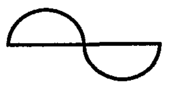
Waveform Registers E0h to F5h

This group of 18 registers is used to select the waveform generated by the operator cells.

D7	D6	D5	D4	D3	D2	D1	D0
!	!	!	!	!	!	Wave Select Bit 1	Wave Select Bit 0

Wave Select (D1 to D0)

Use the Wave Select bit to control the shape of the waveform generated by operator cells.

	D1	D0	Description	
Settings:	1	1	Set waveform to:	
	1	0	Set waveform to:	
	0	1	Set waveform to:	
	0	0	Set waveform to:	
Default	0	0		

12 *Enhanced FM Synthesizer Register Functions*

This chapter describes additional functions available with the newest models of the Pro AudioSpectrum including the Pro AudioSpectrum Plus, Pro AudioSpectrum 16, and the CDPC. All these systems come with Yamaha OPL3 stereo FM synthesizer chip.

With the exception of the additional registers documented here, the OPL3 is register -compatible with systems based on the Yamaha 3812 (such as the Thunder Board and earlier versions of the Pro AudioSpectrum). For information on the compatible registers, see Chapter 11, "Standard FM Synthesizer Register Functions."

Some functions are available only on the left or right side of the OPL3 chip and, therefore, can only be accessed using the left or right side interfaces (388H and 389H or 38AH or 38BH). Functions available only on one side of the chip are noted in the specific reference for that function.

Some pre-existing register functions, such as the connection function, have been enhanced. This section documents only new functions or pre-existing register functions that have been enhanced.

Channel/Connection Select Register 04h (right half of OPL3)

When New (bit D0 of register 05h) is set to OPL3 mode (D0 = 1), this register controls how individual channels are connected together in four operator modes.

D7	D6	D5	D4	D3	D2	D1	D0
¦	¦	Channel/Connection Select 5	Channel/Connection Select 4	Channel/Connection Select 3	Channel/Connection Select 2	Channel/Connection Select 1	Channel/Connection Select 0

Channel/Connection Select (D4 to D1)

Use these bits to specify which channels are used in two- and four-operator connections.

	Dn	Description
Settings:	1	In four-operator channel mode, select channels 6, 5, 4, 3, 2, and 1.
		In two-operator mode, select pairs 12/15, 11/14, 10/13, 3/6, 2/5, and 1/4.
	0	Do not select channels.
Default:	0	

Select OPL3 Register 05h

The Select OPL3 register controls FM synthesizer mode (Yamaha 3812 or Yamaha OPL3 mode). Selecting OPL3 mode activates the additional functions described in this chapter.

D7	D6	D5	D4	D3	D2	D1	D0
¦	¦	¦	¦	¦	¦	¦	New

New (D0)

Use this bit to toggle between 3812 mode and OPL3 mode.

	Dn	Description
Settings:	1	Select OPL3 mode.
	0	Select 3812 mode.
Default:	0	

Feedback/Connection/Stereo Left and Right Register(s) C0h to C8h of left and right half of chip

When the New bit (register 05h, bit DO) is set to OPL3 mode, this set of registers activates a different set of functions than when the FM synthesizer is in 3812 mode. In OPL3 mode, these registers enable left and right stereo channels and add four new four-operator connection modes.

When the New bit is set to 3812 mode (D0 = 0), registers C0h to C8h work as documented in Chapter 11, "Standard FM Synthesizer Register Functions."

D7	D6	D5	D4	D3	D2	D1	D0
--	--	Stereo Left	Stereo Right	--	--	--	Connection Mode

Stereo Left (D5)

When the New bit (register 05h, bit DO) is set to OPL3 mode, use these bits to enable left and right stereo channels and add four new four-operator connection modes.

	D5 (left side)	D5 (right side)	Description
Settings:	1	1	Output data to the left channel from both the left and right hand sides of the chip.
	1	0	Output data to the left channel from only the left hand side of the chip.
	0	1	Output data to the left channel from only the right hand side of the chip.
	0	0	Do not output data to the left channel.
Default:	0		

Stereo Right (D4)

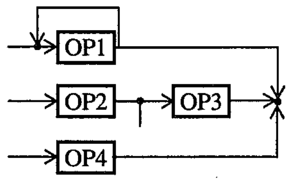
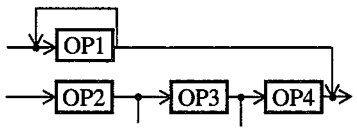
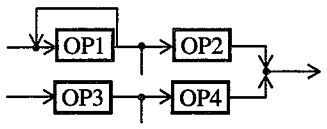
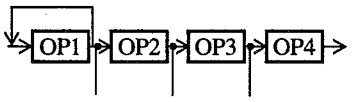
When OPL3 mode is enabled, use this bit to direct data output to the right channel. Stereo Right bits can be controlled individually for both the left and right hand sides of the OPL3 chip.

	D4 (left side)	D4 (right side)	Description
Settings:	1	1	Output data to the right channel from both the left and right hand sides of the chip.
	1	0	Output data to the right channel from only the left hand side of the chip.
	0	1	Output data to the right channel from only the right hand side of the chip.
	0	0	Do not output data to the right channel.
Default:	0		

Connection Mode (D0)

When OPL3 mode is enabled, use bits D0 of the twin set of C0h to C8h registers to select four other connection modes.

See the following diagrams for visual depictions of the new connection modes.

	D0 (left side)	D0 (right side)	Description	
Settings:	1	1	Set connection mode to:	
	1	0	Set connection mode to:	
	0	1	Set connection mode to:	
	0	0	Set connection mode to:	
Default:	0	0		

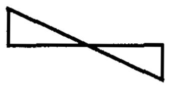
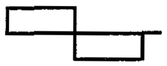
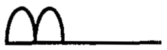
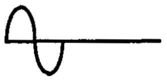
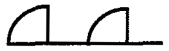
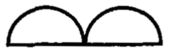
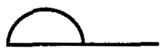
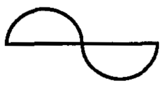
Waveform Registers E0h to F5h

When OPL3 mode is enabled, this group of registers selects four new waveform types.

D7	D6	D5	D4	D3	D2	D1	D0
–	–	–	–	–	Wave Select Bit 2	Wave Select Bit 1	Wave Select Bit 0

Wave Select (D1 to D0)

Use the Wave Select bit to set the shape of the waveform generated by operator cells.

	D2	D1	D0	Description	
Settings:	1	1	1	Set waveform to:	
	1	1	0	Set waveform to:	
	1	0	1	Set waveform to:	
	1	0	0	Set waveform to:	
	0	1	1	Set waveform to:	
	0	1	0	Set waveform to:	
	0	0	1	Set waveform to:	
	0	0	0	Set waveform to:	
Default:		0	0		

MIDI Programming Section

13 MIDI Programming Essentials

This chapter provides the basic information you need to know in order to use the MIDI function calls and hardware programming register functions described in later chapters. For information on the MIDI function calls, see Chapter 14, "Low-level MIDI Function Call Reference." For information on MIDI hardware register functions, see Chapter 15, "MIDI Hardware Register Functions."

MIDI software API information

The information in the following sections is pertinent to both the low-level and high-level function call API's for the MIDI device.

Low-Level MIDI API programming steps

The following procedure shows the order in which you should use high level API function calls. The function calls are documented in Chapter 14, "Low-level MIDI Function Call Reference."

1. **Initialize MIDI hardware and function call library.**

Function call:	`mvMIDIEnable`

2. **Perform MIDI input, output, or both.**

Function calls:	`mvMIDIGetByte`
	`mvMIDIGetBuff`
	`mvMIDISendByte`
	`mvMIDISendBuff`

3. **Clean up and terminate MIDI programming.**

Function call:	`mvMIDIDisable`

Global variable

One global variable, `_MidiInFilter`, is available from the MIDI routines.

The `_MidiInFilter` variable filters out MIDI ACTIVE SENSE bytes. By default, the filter actively removes these bytes. It can be turned off by masking out the appropriate bit(s) in the variable. Future versions of the Pro AudioSpectrum code will provide additional filtering capabilities.

The following table shows `_MidiInFilter` bit assignments according to bit position:

Description	Bit Settings
MF_SYSEX	0000000000000001b
MF_MTCQTR	0000000000000010b
MF_SONGPOS	0000000000000100b
MF_SONGSEL	0000000000001000b
MF_UNDEFINED1	0000000000010000b
MF_UNDEFINED2	0000000000100000b
MF_TUNE	0000000001000000b
MF_ENDEX	0000000010000000b
MF_MIDICLOCK	0000000100000000b
MF_UNDEFINED3	0000001000000000b
MF_START	0000010000000000b
MF_CONTINUE	0000100000000000b
MF_STOP	0001000000000000b
MF_UNDEFINED4	0010000000000000b
MF_ACTSENSE	0100000000000000b
MF_SYSRESET	1000000000000000b

Table 6 _MidiInFilter Bit Settings

Caution: The `_MidiInFilter` variable is an unsigned integer. You should mask out only the active sense filter bit (`MF_ACTSENSE`) to keep from interfering with future functionality.

The code implementing this variable is available in the `MIDIA.ASM` file.

MV101 interrupt control

The MV101 MIDI controller provides software control of I/O interrupts, FIFO control, and other important features. Interrupts from each MIDI device are routed through the Pro Audio Spectrum Interrupt Mask and Status register.

To set up interrupt processing:

1. **Enable the interrupt on the appropriate device.**
2. **Enable the Pro Audio MIDI interrupt.**
3. **Enable the PC system interrupt controller.**

For more information on the Pro AudioSpectrum interrupt control, see Chapter 3, "Common Function Calls and Hardware Registers."

14 Low-level MIDI Function Call Reference

This chapter describes the low-level MIDI function call API. The routines in this section let you read or write MIDI data to any of the Pro Audio Spectrum MIDI devices. The programming interface provides six separate functions: two for reading MIDI data, two for writing MIDI data, and two for setup. You can find source code to these functions in the `MIDIA.ASM` file.

mvMIDIEnable

Use `mvMIDIEnable` to initialize the MIDI hardware interface for bi-directional I/O.

This function resets the MIDI interface and enables interrupts for either direction and sets up the global variable, `_MidiInFilter`, to filter out all incoming ACTIVE SENSE MIDI messages.

The `intl` parameter determines whether I/O is either polled or interrupt driven. Polled I/O uses direct I/O to or from the MIDI interface. Interrupt driven I/O enables the Pro Audio interrupts to allow MIDI data to be sent and received using an internal buffering mechanism. The programming interface is the same in both cases.

Calling Convention

```
int mvMIDIEnable (int enableint);
```

Input Parameters

Parameter	Type	Value	Description
enableint	integer	Bit 0 = 1	Enable input interrupts.
		Bit 1 = 1	Enable output interrupts.

Return Values

Parameter	Type	Value	Description
int	integer	0	MIDI hardware was successfully initialized.
		0xFFFF	MIDI initialization failed.

Related Topics

None.

mvMIDIDisable

Use `mvMIDIDisable` to shut down the MIDI hardware.

`MVMIDIDisable` resets the MIDI hardware interface, and disables and unhooks interrupts. Do not make any calls to the MIDI I/O routines until performing an `mvMIDIEnable` call.

Calling Convention

```
void mvMIDIDisable ();
```

Input Parameters
None.

Return Values
None.

Related Topics
None.

mvMIDIGetBuff

Use `mvMIDIGetBuff` to fetch a buffer of data from the MIDI interface.

This routine loads as many bytes as possible from the MIDI input buffer into the user's buffer, up to the maximum count specified in the count parameter.

The routine returns only currently available data; it does not wait for more to be received. Therefore, a return value of zero is possible, indicating no MIDI data available.

Calling Convention

```
int mvMIDIGetBuff (int limit, char far *buffer);
```

Input Parameters

Parameter	Type	Value	Description
limit	integer		Specifies the maximum number of bytes to load in the buffer
*buffer	far pointer		Pointer to the user's target buffer.

Return Values

Parameter	Type	Value	Description
int	integer	1	The data available from the MIDI FIFO or circular buffer was placed in the user's buffer.
		0	No data available from the MIDI FIFO or circular buffer.

Related Topics
None.

mvMIDIGetByte

Use `mvMIDIGetByte` to fetch the next byte from the MIDI channel.

This function returns the next byte available from the internal buffer of the receiver. If no data is available, the routine returns a value of 0xFFFF (signed integer negative one). If data is available, the return value will be less than 256.

Calling Convention

```
int mvMIDIGetByte ();
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	16-bit signed integer	MIDI data	Next data byte available from the MIDI input buffer.
		-1	No data is available.

Related Topics

None.

mvMIDISendBuff

Use `mvMIDISendBuff` to send a buffer of data out the MIDI device.

This routine passes the entire buffer to the MIDI device. If the buffer is larger than the physical MIDI FIFO, the return will be delayed until the routine finishes sending the buffer.

For polled output, `mvMIDISendBuff` feeds the on-board 16 byte FIFO. When the on-board 16 byte FIFO is filled, the routine polls the MIDI interface until additional buffer space is available.

For interrupt driven output, user data is loaded into a 128 byte circular buffer. The circular buffer sends data to the on-board 16 byte FIFO at interrupt time.

Calling Convention

```
void mvMIDISendBuff (int limit, char far *buffer);
```

Input Parameters

Parameter	Type	Value	Description
limit	integer	Max count	Specifies the maximum number of bytes to load in the buffer.
*buffer	far pointer		Pointer to the user's target buffer.

Return Values

None.

Related Topics

None.

mvMIDISendByte

Use `mvMIDISendByte` to send one data byte out the MIDI interface.

For polled- mode output, the byte is sent when room is available in the outbound FIFO. For interrupt -driven output, the byte is queued until the interrupt sends it out.

Calling Convention

```
void mvMIDISendByte (char databyte);
```

Input Parameters

Parameter	Type	Value	Description
databyte	char	Data byte	The byte to be sent.

Return Values

None.

Related Topics

None.

15 *MIDI Hardware Register Functions*

This chapter describes register functions for Media Vision's state-of-the-art MIDI controller, the MV101, which is used in the Pro AudioSpectrum Plus, Pro AudioSpectrum 16, and the CDPC.

Earlier versions of the Pro AudioSpectrum used the Yamaha 3802 MIDI controller. Yamaha 3802 programming is not covered in this chapter.

MIDI Prescale Register 1788h

Use the MIDI Prescale register to set the time interval for MIDI timer interrupts. The formula for calculating the time is (N+1)/2 in ms, where N is the programmed value.

A MIDI timer interrupt is generated when the value being decreased wraps from zero to 255, and is enabled setting bit D0 of the MIDI Control Register. For more information on this register, see"MIDI Control Register 178Bh" on page 15-3.

D7	D6	D5	D4	D3	D2	D1	D0
Timer Interval Bit 7	Timer Interval Bit 6	Timer Interval Bit 5	Timer Interval Bit 4	Timer Interval Bit 3	Timer Interval Bit 2	Timer Interval Bit 1	Timer Interval Bit 0

Program bits D7 though D0 as a group. The table here shows two of a possible 255 settings.

	D7	D6	D5	D4	D3	D2	D1	D0	Description
Settings:	1	1	1	1	1	1	1	1	Set time interval to 128 ms.
	0	0	0	0	0	0	0	1	Set time interval to 1 ms.
Default:									N/A

Note: Using this register is optional. It is intended to be used to generate special interrupts.

MIDI Timer Register 1789h

Use the MIDI Timer register to determine the contents of the MIDI timer. This register is read-only. The register value is decreased in 2 millisecond increments and resets to zero when the MIDI Prescale register is programmed.

This register is useful for controlling MIDI system interrupts. The MIDI system generates an interrupt under one of the following conditions: when the MIDI Timer register wraps to zero, or, when the MIDI Timer register contains the same value as the MIDI Compare Time register. For information on the MIDI Compare Time register, see "MIDI Compare Time Register 1B8Ah" on page 15-10.

D7	D6	D5	D4	D3	D2	D1	D0
Timer Count Bit 7	Timer Count Bit 6	Timer Count Bit 5	Timer Count Bit 4	Timer Count Bit 3	Timer Count Bit 2	Timer Count Bit 1	Timer Count Bit 0

Set bits D7 through D0 as a group. The table here shows two of a possible 255 settings.

	D7	D6	D5	D4	D3	D2	D1	D0	Description
Settings:	1	1	1	1	1	1	1	1	Current MIDI time = 128 ms.
	0	0	0	0	0	0	0	1	Current MIDI time = 1 ms.
Default:									N/A

MIDI Data Register178Ah

Use the MIDI Data register to send data and receive to and from MIDI FIFO buffers. These FIFOs allow MIDI data to be read or written in bursts, thereby minimizing host processor overhead.

Write to the register to send data to the MIDI transmitter FIFO buffer. Read from the register to retrieve data from the MIDI receiver FIFO.

D7	D6	D5	D4	D3	D2	D1	D0
MIDI Data Bit 7	MIDI Data Bit 6	MIDI Data Bit 5	MIDI Data Bit 4	MIDI Data Bit 3	MIDI Data Bit 2	MIDI Data Bit 1	MIDI Data Bit 0

Set bits D7 through D0 as a group. The table here shows two of a possible 255 settings.

	D7	D6	D5	D4	D3	D2	D1	D0	Description
Settings:	1	1	1	1	1	1	1	1	MIDI data.
	0	0	0	0	0	0	0	1	MIDI data.
Default:									N/A

MIDI Control Register 178Bh

Use the MIDI Control register to perform MIDI management tasks such as: echoing input to output; resetting output and input FIFO pointers; and enabling output FIFO half-empty, output FIFO empty, input data available, compare time, and time stamp interrupts.

D7	D6	D5	D4	D3	D2	D1	D0
Echo input to output	Reset Output FIFO pointer	Reset Input FIFO pointer	Enable Output FIFO Half empty int	Enable Output FIFO Empty interrupt	Enable Input Data Avail. interrupt	Enable Compare Time interrupt	Enable Time stamp interrupt

Echo Input To Output (D7)

Setting this bit allows data received though the MIDI receiver FIFO buffer to be transmitted immediately to the MIDI transmitter FIFO buffer, thus creating a THRU connection on the IN/OUT MIDI connector.

	D7	Description
Settings:	1	Enable MIDI echoing.
	0	Disable MIDI echoing.
Default:	1	Enable MIDI echoing.

Reset Output FIFO Pointer (D6)

Setting the Reset Output FIFO Pointer bit to 1 clears the output FIFO pointer. After clearing the pointer, you must immediately reset the Reset Output FIFO Pointer bit to 0.

	D6	Description
Settings:	1	Clear output FIFO pointer.
	0	Normal state.
Default:	0	

Reset Input FIFO Pointer (D5)

Setting the Reset Input FIFO Pointer bit to 1 clears the input FIFO pointer. After clearing the pointer, you must immediately reset the Reset Input FIFO Pointer bit to 0.

	D6	Description
Settings:	1	Clear input FIFO pointer.
	0	Normal state.
Default:	0	

Enable Output FIFO Half-Empty Interrupt (D4)

Setting this bit to 1 enables the output FIFO half-empty interrupt. This interrupt is generated when half of the data in the output FIFO is transmitted.

	D4	Description
Settings:	1	Enable FIFO half-empty interrupt.
	0	Disable FIFO half-empty interrupt.
Default:	0	

Enable Output FIFO Empty Interrupt (D3)

Setting this bit to 1 enables the output FIFO empty interrupt. This interrupt is generated when the last byte of the output FIFO is transmitted.

	D3	Description
Settings:	1	Enable FIFO empty interrupt.
	0	Disable FIFO empty interrupt.
Default:	0	

Enable Input Data Available Interrupt (D2)

Setting this bit to 1 enables the input data available interrupt. This interrupt is generated when MIDI data is received and is available in the FIFO.

	D2	Description
Settings:	1	Enable input data available interrupt.
	0	Disable input data available interrupt.
Default:	0	

Enable Compare Time Interrupt (D1)

Setting this bit to 1 enables the compare time interrupt. This interrupt is generated whenever the value in the MIDI timer is equal to the MIDI Compare Time register.

	D1	Description
Settings:	1	Enable compare time interrupt.
	0	Disable compare time interrupt.
Default:	0	

Enable Time Stamp Interrupt (D0)

Setting this bit to 1 enables the time stamp interrupt. This interrupt is generated whenever the MIDI timer wraps from FFh to 00h.

	D1	Description
Settings:	1	Enable time stamp interrupt.
	0	Disable time stamp interrupt.
Default:	0	

MIDI Status Register 1B88h

Read the MIDI Status register to detect: frame errors; output and input FIFO overruns; output FIFO half-empty and FIFO empty interrupts; input data available interrupts; and compare time and time stamp interrupts. Clear MIDI Status bits by setting them to 0.

Be aware that overrun and frame error interrupts cannot be disabled. This means your interrupt routine must always process these error conditions.

D7	D6	D5	D4	D3	D2	D1	D0
Frame Error	Output FIFO overrun	Input FIFO overrun	Output FIFO Half Empty interrupt	Output FIFO Empty interrupt	Input Data Available. interrupt	Compare Time interrupt	Time Stamp interrupt

Frame Error (D7)

The Frame Error bit is set to 1 when a frame error occurs.

	D7	Description
Settings:	1	Frame error occurred.
	0	Normal state.
Default:	0	

Output FIFO Overrun (D6)

The Output FIFO Overrun bit is set to 1 when an overrun occurs.

	D6	Description
Settings:	1	Output FIFO overrun occurred.
	0	Normal state.
Default:	0	

Input FIFO Overrun (D5)

The Input FIFO Overrun bit is set to 1 when an overrun occurs.

	D5	Description
Settings:	1	Input FIFO overrun occurred.
	0	Normal state.
Default:	0	

Output FIFO Half-Empty Interrupt (D4)

The Output FIFO Half-Empty Interrupt bit is set to 1 when an interrupt occurs.

	D4	Description
Settings:	1	Output FIFO half-empty interrupt occurred.
	0	Normal state.
Default:	0	

Output FIFO Empty Interrupt (D3)

The Output FIFO Empty Interrupt bit is set to 1 when an interrupt occurs.

	D4	Description
Settings:	1	Output FIFO empty interrupt occurred.
	0	Normal state.
Default:	0	

Input Data Available Interrupt (D2)

The Input Data Available Interrupt bit is set to 1 when an interrupt occurs.

	D4	Description
Settings:	1	Input data available interrupt occurred.
	0	Normal state.
Default:	0	

Compare Time Interrupt (D1)

The Compare Time Interrupt bit is set to 1 when an interrupt occurs indicating that the compare time value is equal to the MIDI timer value.

	D4	Description
Settings:	1	Compare time interrupt occurred.
	0	Normal state.
Default:	0	

Time Stamp Interrupt (D0)

The Time Stamp Interrupt bit is set to 1 when an interrupt occurs.

	D4	Description
Settings:	1	Time stamp interrupt occurred.
	0	Normal state.
Default:	0	

MIDI FIFO Count Register 1B89h

Use the MIDI FIFO Count register to determine the number of bytes that can be sent to the output FIFO buffer and the number of bytes that have been received and stored in the input FIFO buffer.

The upper four bits of this register indicate the number of bytes that can be written to the output FIFO before it becomes full. The lower four bits indicate the number of bytes of data that have been received and stored in the input FIFO.

D7	D6	D5	D4	D3	D2	D1	D0
FIFO Space Bit 3	FIFO Space Bit 2	FIFO Space Bit 1	FIFO Space Bit 0	Received Bytes Bit 3	Received Bytes Bit 2	Received Bytes Bit 1	Received Bytes Bit 0

FIFO Space (D7 through D4)

This group of bits indicates the number of bytes that can be written to the output FIFO before it becomes full. Valid settings range from 0h to Fh. A count of 0h indicates that either zero or 16 bytes are ready to be loaded into the output FIFO.

Determining whether a 0h count indicates zero or sixteen bytes is difficult. The Output FIFO Empty bit only goes high when the FIFO shifts out its last byte to the transmitter. When the Output FIFO Pointer is cleared, the Output FIFO Empty bit remains zero. To handle this situation, assume that a 0h value indicates 16 bytes. Write only 15 MIDI bytes at a time to the output FIFO and ignore the empty bit..

	D7	D6	D5	D4	Description
Settings:	1	1	1	1	Either 0 or 16 bytes available for output FIFO.
	1	1	1	0	15 bytes available for output FIFO.
	1	1	0	1	14 bytes available for output FIFO.
	1	1	0	0	13 bytes available for output FIFO.
	1	0	1	1	12 bytes available for output FIFO.
	1	0	1	0	11 bytes available for output FIFO.
	1	0	0	1	10 bytes available for output FIFO.
	1	0	0	0	9 bytes available for output FIFO.
	0	1	1	1	8 bytes available for output FIFO.
	0	1	1	0	7 bytes available for output FIFO.

	D7	D6	D5	D4	Description
	0	1	0	1	6 bytes available for output FIFO.
	0	1	0	0	5 bytes available for output FIFO.
	0	0	1	1	4 bytes available for output FIFO.
	0	0	1	0	3 bytes available for output FIFO.
	0	0	0	1	2 bytes available for output FIFO.
	0	0	0	0	1 byte available for output FIFO.
Default:					N/A

Received Bytes (D3 though D0)

This group of bits indicates the number of bytes that that have been received and stored in the input FIFO. Valid settings range from 0h to Fh. A count of 0h indicates that either zero or 16 bytes have been received.

To determine whether zero or 16 bytes have been received, determine if data is available by checking bit D2 of the MIDI Status register. If D2 = 0, the FIFO count indicates zero bytes have been received. If D2 = 1, 16 bytes have been received.

	D3	D2	D1	D0	Description
Settings:	1	1	1	1	Either 0 or 16 bytes received from input FIFO.
	1	1	1	0	15 bytes received from input FIFO.
	1	1	0	1	14 bytes received from input FIFO.
	1	1	0	0	13 bytes received from input FIFO.
	1	0	1	1	12 bytes received from input FIFO.
	1	0	1	0	11 bytes received from input FIFO.
	1	0	0	1	10 bytes received from input FIFO.
	1	0	0	0	9 bytes received from input FIFO.
	0	1	1	1	8 bytes received from input FIFO.
	0	1	1	0	7 bytes received from input FIFO.
	0	1	0	1	6 bytes received from input FIFO.
	0	1	0	0	5 bytes received from input FIFO.
	0	0	1	1	4 bytes received from input FIFO.
	0	0	1	0	3 bytes received from input FIFO.
	0	0	0	1	2 bytes received from input FIFO.
	0	0	0	0	1 byte received from input FIFO.
Default:					N/A

MIDI Compare Time Register 1B8Ah

Use the MIDI Compare Time register to set values that will evaluated against MIDI timer values. When the MIDI timer count is equal to the value programmed into this register, an interrupt is generated.

This register can be used to cause events that are triggered by a time stamp.

D7	D6	D5	D4	D3	D2	D1	D0
MIDI Timer Compare Bit 7	MIDI Timer Compare Bit 6	MIDI Timer Compare Bit 5	MIDI Timer Compare Bit 4	MIDI Timer Compare Bit 3	MIDI Timer Compare Bit 2	MIDI Timer Compare Bit 1	MIDI Timer Compare Bit 0

Note: Using this register is optional. It is useful for generating special interrupts.

Program bits D7 though D0 as a group. The table here shows two of a possible 255 settings.

	D7	D6	D5	D4	D3	D2	D1	D0	Description
Settings:	1	1	1	1	1	1	1	1	Set compare time value to 255.
	0	0	0	0	0	0	0	1	Set compare time value to 1.
Default:									N/A

CD-ROM Programming Section

16 *CD-ROM Programming Essentials*

This chapter provides background information and information that is generally applicable to all of the CD-ROM API's including:

- Location of CD-ROM source code on the Pro AudioSpectrum Developer's Toolkit diskette
- CD-ROM function call syntax
- Definitions of CD-ROM units of measure
- Red Book address definitions
- CD-ROM device driver error messages and status codes

The Media Vision CD-ROM programming environment consists of a higher level function call API and a lower level function call API. The lower level API is fully developed, and is the base line upon which all Media Vision software (including more abstract API's) will be developed.

The high-level API set, which is built on the low-level API, is currently being expanded. When it is completed, this API set will make it much easier for novice and intermediate level multi-media programmers to work with the Media Vision product, and for experienced developers to prototype new functions. The most important CD-ROM calls have been implemented in the current release. Calls from both API sets can be inter-mixed.

For more information on the low-level API, see Chapter 18, "Low-Level CD-ROM Function Call Reference." For more information on the high-level API, see Chapter 17, "High-Level CD-ROM Function Call Reference."

CD-ROM function call syntax

Most functions specify the drive number of the CDROM device as their first argument s. Use `getfirstcdrom` to set the default drive or `getcdromunits`, which returns an array of structures containing the drive numbers. For more information on these calls, see "Microsoft CD-ROM Extension Function Call Reference" on page 19-1.

CD-ROM units of measure

CD-ROM technology defines frames, seconds, and minutes as its basic units of measure. You use these units of measure to tell your program where to position the read mechanism of the CD-ROM drive on a given disk. Frames, seconds, and minutes are defined as follows:

Function	Value
Frame	1/75th of a second
Second	One second of time, 1/60th of a minute
Minute	A minute of time, 60 seconds, 4500 frames

Red book address definitions

A *Red Book address* is an addressing unit used to locate a particular position on a CD. It is made up of three portions: minutes (0 through 59), seconds (0 through 59), and frames (0 through 74). Normally, each element is represented by a single value in a 32-bit unsigned long integer with the most significant byte not used. The conversion from Red Book address into physical sector is:

Sector = (minute*60*75)+(second*75)+frame

Red book addresses can also be specified as *structures*. The following table shows Red Book addresses in both formats:

Red Book address value:	Format	Comments

"long" value	0xMMSSFF	MM = minutes,
		SS = seconds,
		FF = frames

"structure"	struct redbookaddr {char frame, sec, min, unknown}	

Use the following values to convert a Red Book value to "number of frames":

Value	Equal to:	Description
frames	(Red Book& 0x00FF0000)>> 16;	Number of minute.
frames*	60;	Number of seconds.
frames+	(Red Book& 0x0000FF00)>> 8;	Number of seconds.
frames*	75;	Number of frames.
frames+	Red Book& 0x000000FF;	Number of frames.

BCD to integer value conversion

Some CD routines return values in BCD instead of binary (*qchaninfo.track*, for example). Use the conversion functions, `inttobcd` and `bcdtoint`, to convert values from BCD to integer and visa versa. For information on these two functions, see "inttobcd" on page 18-15 and "bcdtoint" on page 18-14.

TOC

TOC stands for Table of Contents and is a table that contains addresses of every sound track in a CD. The addresses are specified in 32-bit unsigned long integers.

In order to minimize interrupting the CD-ROM, your application program should read the TOC once and then make it accessible from memory.

CD-ROM device driver status and error codes

The CD ROM driver status is a specific 16-bit value that is generated from the CD ROM driver. Bits 15, 9, and 8 indicate the status of the driver and are defined as follows:

Status Bits and Setting	Description
D15 = 1	Driver error. See four lowest bits for specific error condition.
D9 = 1	Drive busy.
D8 = 1	Command done.

When Bit 15 is set to 1, bits 3, 2, 1, and 0 indicate specific CD-ROM error conditions. These error conditions are defined as follows:

D3	D2	D1	D0	Description
0	0	0	0	Write-protect violation
0	0	0	1	Unknown unit
0	0	1	0	Drive not ready
0	0	1	1	Unknown command
0	1	0	0	CRC error
0	1	0	1	Bad request header length
0	1	1	0	Seek error
0	1	1	1	Unknown media
1	0	0	0	Sector not found
1	0	0	1	Printer out of paper
1	0	1	0	Write fault
1	0	1	1	Read fault
1	1	0	0	General failure
1	1	0	1	Reserved
1	1	1	0	Reserved
1	1	1	1	Invalid disk change

Table 7 CD-ROM Driver Error Codes

17 High-Level CD-ROM Function Call Reference

This chapter is a reference to high-level CD-ROM calls. Over time, this set of calls will be expanded and enhanced significantly. For general information on how to program the CD-ROM, see Chapter 16, "CD-ROM Programming Essentials."

All functions described here require that the internal data structures be initialized using `buildaudiotoc`. Unless otherwise indicated, you can find prototypes of the function calls in the `CDMASTER.H` file.

buildaudiotoc

Use `buildaudiotoc` to initialize CD-ROM data structures and prepare to program the CD-ROM. This function fills the internal *discinfo* structure using `cddiscinfo`, allocates an array of *trackinfo* structures, fills *trackinfo* using `cdtrackinfo`, makes an after-the-last track min:sec:frame, and returns the address of the internal *cdtable* structure.

Note: This function destroys the previous table of contents for the drive if it existed.

Calling Convention

```
struct cdtable * buildaudiotoc (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.

Return Values

Parameter	Type	Value	Description
struct cdtable *	data structure		Pointer to the internal *cdtable* structure.
		null	Null if an error occurred with cddiscinfo() or calloc().

Related topics

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

To free allocated memory and clear the table entry for the drive, see "destroyaudiotoc" on page 17-3.

createaudiotoc

Use `createaudiotoc` to allocate memory and prepare to program the CD-ROM. This function is similar to the `buildaudiotoc` function except that it returns without filling the *trackinfo* structures, enabling disc initialization to occur quickly.

All internal routines verify that the trackinfo structures have been properly initialized, so you can choose between `buildaudiotoc` and `createaudiotoc` functions.

Calling Convention

```
struct cdtable * createaudiotoc (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.

Return Values

Parameter	Type	Value	Description
struct cdtable*	data structure		Pointer to the internal *cdtable* structure or a null if an error occurred with cddiscinfo() or calloc().

Related topics

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

To free allocated memory and clear the table entry for the drive, see "destroyaudiotoc" on page 17-3.

To initialize CD-ROM data structures and prepare for CD-ROM programming, see "buildaudiotoc" on page 17-2.

destroyaudiotoc

Use `destroyaudiotoc` to free memory allocated by `buildaudiotoc` and perform housekeeping on internal variables.

Calling Convention

```
void destroyaudiotoc (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.

Return Values

Parameter	Type	Value	Description
void		0	Internal address was null.
		-1	Internal address was not null.

Related topics

To initialize the routine internal tables and variables, see "buildaudiotoc" on page 17-2.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

getcdtable

Use `getcdtable` to returns the address of the internal *cdtable* structure. The *cdtable* structure holds pointers to the internal *discinfo* structure and to the internal array of *trackinfo* table structures.

Calling Convention

```
struct cdtable * getcdtable (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.

Return Values

Parameter	Type	Value	Description
struct cdtable *	data structure		Pointer to internal *cdtable* structure.
		Null	Indicates buildaudiotoc was not called or failed.

Related topics

To initialize the routine internal tables and variables, see "buildaudiotoc" on page 17-2.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

To determine the address of *discinfo* table for the specified drive, see "getdiscinfotable" on page 17-5.

To determine the address of an internal array of *trackinfo* structures for the specified drive, see "gettrackinfotable" on page 17-6.

For more information on the *cdtable* data structure, see "struct cdtable" on page B-3.

getdiscinfotable

Use `getdiscinfotable` to determine the address of the internal *discinfo* table for the specified drive.

Calling Convention

```
struct discinfo * getdiscinfotable (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.

Return Values

Parameter	Type	Value	Description
struct discinfo *	data structure		Pointer to the internal discinfo table for the specified drive.
		null	Indicates buildaudiotoc was not called or failed.

Related topics

To initialize the routine internal tables and variables, see "buildaudiotoc" on page 17-2.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

To determine discinfo table address, see "getcdtable" on page 17-4.

To determine the address of an internal array of *trackinfo* structures for the specified drive, see "gettrackinfotable" on page 17-6.

For more information on the *discinfo* data structure, see "struct discinfo" on page B-4.

gettrackinfotable

Use `gettrackinfotable` to determine the address of the internal array of *trackinfo* structures.

Calling Convention

```
struct trackinfo * gettrackinfotable (int);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.

Return Values

Parameter	Type	Value	Description
trackinfo*	data structure		Pointer to internal array of trackinfo structures.
		null	Indicates buildaudiotoc was not called or failed.

Related topics

To initialize the routine internal tables and variables, see "buildaudiotoc" on page 17-2.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

gettrackframes

Use `gettrackframes` to retrieve the number of frames specified in the track parameter. You can derive the track number from the audio table of contents.

Calling Convention

```
long gettrackframes (int cddrive, int tracknum);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.
tracknum	integer		Specifies the track number.

Return Values

Parameter	Type	Value	Description
long	32-bit signed integer		Length (frames) of the specified track.
		0	Error - buildaudiotoc() was not called first or the track number is out of range.
		-1	Error - the track calculation produced a negative length.

Related topics

To initialize the routine internal tables and variables, see "buildaudiotoc" on page 17-2.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

playcdtrack

Use `playcdtrack` to play a track starting from an offset (specified in seconds) for a period of time (also specified in seconds). If length equals -1, playing will continue till the end of the disc. If length equals -2, playing will continue till the end of the track.

Calling Convention

```
int playcdtrack (int cddrive, int track, int offset, int length);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.
track	integer		Specifies track number.
offset	integer		Specifies offset (in seconds) from the start of the track.
length	integer		Specifies length (in seconds) from the starting offset.

Return Values

Parameter	Type	Value	Description
int	integer		CD-ROM driver status.

Related topics

To initialize the routine internal tables and variables, see "buildaudiotoc" on page 17-2.

To get CD-ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

seektotrack

Use seektotrack to move the CD drive head to the specified track. You can determine track number by looking at the audio table of contents.

Calling Convention

```
int seektotrack (int cddrive, int track);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.
track	integer		Specifies a track number.

Return Values

Parameter	Type	Value	Description
int	integer	0	Error - buildaudiotoc() was not called first, or the track number is out of range.
		Non-zero	Indicates CD-ROM driver status.

Related topics

To initialize the routine internal tables and variables, see "buildaudiotoc" on page 17-2.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

18 Low-Level CD-ROM Function Call Reference

This chapter is a complete reference to Media Vision's low level function calls to CD-ROM.

Unless otherwise indicated, you can find prototypes of the function calls described in this chapter in the `CDROM.H` file.

cdplay

Use `cdplay` to begin playing CD audio from a starting frame for a specified number of frames. Calling cdplay a second time will stop the current playback, advance to the new starting frame, then begin playing again.

Calling Convention

```
int cdplay (int cddrive, long startframe, long framecount);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD drive number.
startframe	32-bit signed integer		Specifies the starting frame number.
framecount	32-bit signed integer		Specifies the count of frames to play.

Return Values

Parameter	Type	Value	Description
int	integer		CD ROM driver status.

Related topics

To resume from a paused status, see "cdresume" on page 18-4.

To stop from a playing state, see "cdstop" on page 18-2.

To pause, see "cdpause" on page 18-3.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

To calculate the starting frame and length frame, see Chapter 16, "CD-ROM Programming Essentials."

To determine CD ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3, "cdstatus" on page 18-6, and "cdaudiostatus" on page 18-7.

cdstop

Use `cdstop` to stop a playing or paused song.

Calling Convention

```
int cdstop (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD drive number.

Return Values

Parameter	Type	Value	Description
int	integer		CD ROM driver status.

Related topics

To start the CD playing audio, see "cdplay" on page 18-1

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

To determine CD ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3.

cdpause

Use `cdpause` to pause the CD audio output and leave the CD spinning. Use `cdresume` to resume playing.

Calling Convention

```
int cdpause (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD drive number.

Return Values

Parameter	Type	Value	Description
int	integer		CD-ROM driver status.
		0	Audio output has already paused or stopped.

Related topics

To resume from a paused status, see "cdresume" on page 18-4.

To stop from a playing or paused state, see "cdstop" on page 18-2.

To start the CD playing audio, see "cdplay" on page 18-1.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

To determine CD ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3.

cdresume

Use `cdresume` to resume playing audio from CD's in a paused state.

Calling Convention

```
int cdresume (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.

Return Values

Parameter	Type	Value	Description
int	integer		CD-ROM driver status.

Related topics

To stop from a playing or paused state, see "cdstop" on page 18-2.

To start the CD playing audio, see "cdplay" on page 18-1.

To pause the audio play, see "cdpause" on page 18-3.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

To determine CD ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3.

cdseek

Use `cdseek` to move the drive head to location specified by the frame number parameter.

Calling Convention

```
int cdseek (int cddrive, long framenumber);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD drive number.
long1 framenumber	32-bit signed integer		Specifies a valid starting frame number.

Return Values

Parameter	Type	Value	Description
int	integer		CD-ROM driver status

Related topics

To calculate the starting frame, see "CD-ROM function call syntax" on page 16-1.

To determine CD ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

cdreset

Use `cdreset` to force the driver to clear all internal buffers and re-initialize.

Calling Convention

```
int cdreset (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD drive number.

Return Values

Parameter	Type	Value	Description
int	integer		CD-ROM driver status.

Related topics

To determine CD ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

cdeject

Use cdeject to send a signal to drive to eject the disc.

Note: Some drives do not support this function.

Calling Convention

```
int cdeject (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.

Return Values

Parameter	Type	Value	Description
int	integer		CD-ROM driver status.

Related topics

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

To determine CD ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3.

cdstatus

Use `cdstatus` to determine the CD-ROM drive status. `cdstatus` calls `cdaudiostatus` and returns a bit field. This status is maintained locally and is not associated with the CD-ROM Driver Status derived from the CD-ROM driver.

Each of the return values is represented by one of the bits in the bit field. Certain values will never be returned, such as CD is playing and CD is paused.

Calling Convention

```
int cdstatus (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive.

Return Values

Parameter	Type	Value	Description
int	Bit field integer	0x00	No CD is present in CD-ROM drive.
		0x01	CD is present in CD-ROM drive.
		0x02	CD is playing.
		0x04	CD is paused.

Related topics

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

cdaudiostatus

Use `cdaudiostatus` to determine the status of the CD-ROM driver. `cdaudiostatus` returns the status of the CD-ROM driver along with the next start and end frame values.

Calling Convention

```
int cdaudiostatus (int cddrive, long *beginfm, long *endfm);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD drive number.
beginfm	32-bit signed integer		Pointer to a 32-bit integer to receive the next start frame address.
endfm	32-bit signed integer		Pointer to a 32-bit integer to receive the next end frame address.

Return Values

Parameter	Type	Value	Description
int	integer		CD-ROM driver status.

Related topics

To determine CD ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

cdmediachanged

Use `cdmediachanged` to see if the CD ROM has changed.

Calling Convention

```
int cdmediachanged (int cddrive, int *result);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.
result	integer		Pointer to an integer address to receive a one word result.

Return Values

Parameter	Type	Value	Description
int	integer	0	Function call succeeded.
		-1	Function call failed.
*result	integer	-1	If int = 0,indicates CD-ROM media has changed.
		1	If int = 0, indicates CD-ROM media has not changed.
		0	If int = -1, unknown if CD-ROM media has changed.

Note: If the function return value is -1, then an error occurred in the CD ROM driver. To determine the failure, call `getlasterror` to get the CD-ROM driver status word.

Related topics

To determine the last CD ROM driver status, see "getlasterror" on page 19-12.

To determine CD ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

cddiscinfo

Use `cddiscinfo` to fill the *discinfo* structure with first and last track numbers, and the end of disc address.

Calling Convention

```
int cddiscinfo (int cddrive, struct discinfo*);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.
struct discinfo*	pointer		Pointer to structure to receive discinfo data.

Return Values

Parameter	Type	Value	Description
int	integer		CD-ROM driver status.

Related topics

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

To determine CD ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3.

cdtrackinfo

Use `cdtrackinfo` to fill the *trackinfo* structure with the min:sec:frame address.

Calling Convention

```
int cdtrackinfo (int cddrive, int track, struct trackinfo*);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.
track	integer		Specifies a CD track number. Track number can be derived from the audio table of contents.
trackinfo*	pointer		Pointer to the address of the buffer to receive start and ending track numbers and the last red book address.

Return Values

Parameter	Type	Value	Description
trackinfo*	pointer		Address of start and ending track numbers and the last red book address.

Related topics

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

For information on the *trackinfo* data structure, see "struct trackinfo" on page B-4.

cdqchaninfo

Use `cdqchaninfo` to fill the *qchaninfo* structure with the current location of the drive head within the current track on the disc (in min:sec:frame format).

Calling Convention

```
int cdqchaninfo (int cddrive, struct qchaninfo*);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.
qchaninfo*	pointer		Pointer to the address of the buffer to receive the current location of the CD-drive head.

Return Values

Parameter	Type	Value	Description
int	integer		CD-ROM driver status.

Related topics

To determine CD ROM driver status, see "CD-ROM device driver status and error codes" on page 16-3.

For more information on the *qchaninfo* data structure, see "struct qchaninfo" on page B-4.

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

isanaudiocd

Use `isanaudio` to determine the type of CD inserted in the CD-ROM drive.

Calling Convention

```
int isanaudiocd (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.

Return Values

Parameter	Type	Value	Description
int	Bit field integer	0	Unknown compact disk type detected.
		1	Audio compact disk detected.

Related topics

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

cdseekmsf

Use `cdseekmsf` to calculate the frame offset for the `cdseek` routine.

Calling Convention

```
cdseekmsf (int cddrive, int min, int sec, int frame);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.
min	integer		Specifies the number of minutes to seek forward.
sec	integer		Specifies the number of seconds to seek forward.
frame	integer		Specifies the number of frames to seek forward.

Return Values

This function calculates the frame value, then returns `cdseek`.

Related topics

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

cdplaymsf

Use `cdplaymsf` to calculate the frame offset and length for `cdplay`.

Calling Convention

```
cdplaymsf (int cddrive, int minfwd, int secfwd, int frmfwd, int
minplay, int secplay, int frmplay);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive number.
minfwd	integer		Specifies the number of minutes to move forward before playing.
secfwd	integer		Specifies the number of seconds to move forward before playing.
frmfwd	integer		Specifies the number of frames to move forward before playing.
minplay	integer		Specifies the number of minutes to play.
secplay	integer		Specifies the number of seconds to play.
frmplay	integer		Specifies the number of frames to play.

Return Values

This function calculates the frame values and returns `cdplay`.

Related topics

To get the CD drive number, see "getnumcdroms" on page 19-2 and "getfirstcdrom" on page 19-3.

fixmsf

Use `fixmsf` after performing math operations that use minutes, seconds, and frame values to ensure that minute, second, and frames values are within legal bounds. This function makes each value non-negative; checks that the frame value is between 0 and 74, inclusive; and checks that minutes and seconds are between 0 and 59, inclusive.

Calling Convention

```
int fix msf (int min, int sec, int frame);
```

Input Parameters

Parameter	Type	Value	Description
min	integer		Specifies minutes value.
sec	integer		Specifies seconds value.
frame	integer		Specifies frames value.

Return Values

Parameter	Type	Value	Description
int	integer	-1	Minutes value is negative, other values are not updated.
		0	Minutes value is not negative, other values are updated.

Related topics

None.

bcdtoint

Use `bcdtoint` to convert BCD values to signed, 16-bit integers.

Calling Convention

```
int bcdtoint (int BCD);
```

Input Parameters

Parameter	Type	Value	Description
BCD	integer		Specifies BCD number to convert to integer.

Return Values

Parameter	Type	Value	Description
int	integer		A signed 16 -bit binary quantity representing the16-bit BCD input value.

Related topics

To see a structure expressed in BCD, see "struct qchaninfo" on page B-4.

inttobcd

Use `inttobcd` to convert integers to BCD values.

Calling Convention

```
int inttobcd (int intval);
```

Input Parameters

Parameter	Type	Value	Description
intval	integer		Specifies the integer to convert to BCD.

Return Values

Parameter	Type	Value	Description
int	integer		An unsigned 16-bit value in BCD representing the binary input value.

Related topics

None.

redtolong

Use `redtolong` to convert a Red Book address in the format 0x00MMSSFF into a long value representing the frame address.

Calling Convention

```
long redtolong (struct redaddress);
```

Input Parameters

Parameter	Type	Value	Description
redaddress	data structure		Four-byte structure of the Red Book address.

Return Values

Parameter	Type	Value	Description
long	32-bit signed integer		Frame address corresponding to the Red Book address.

Related topics

To determine the format of Red book addresses and how convert them to frame addresses, see "Red book address definitions" on page 16-2.

msftolong

Use `msftolong` to convert an MSF number to a frame address. The MSF number returned by this call is similar to the Red Book address except that it is presented in byte-reversed order.

Calling Convention

```
longR msftolong (long msfvalue);
```

Input Parameters

Parameter	Type	Value	Description
msfvalue	32-bit signed integer		Four-byte value in 0xFFSSMM00 order.

Return Values

Parameter	Type	Value	Description
longR	32-bit signed integer		Frame address corresponding to the MSF number.

Related topics

None.

longtored

Use `longtored` to convert frame value to a Red Book address. The Red Book address is returned as a *long* value.

Calling Convention

```
longR longtored (long redvalue);
```

Input Parameters

Parameter	Type	Value	Description
redvalue	32-bit signed integer		A frame value.

Return Values

Parameter	Type	Value	Description
longR	32-bit signed integer		Red Book address corresponding to the input frame value.

Related topics

None.

19 Microsoft CD-ROM Extension Function Call Reference

This chapter describes the function calls specified in under Microsoft CD-ROM Extensions. CD-ROM Extensions allow MS-DOS applications to access CD-ROM's in the High Sierra or ISO-9660 formats as if they were standard MS-DOS block devices. An entire CD-ROM is represented as a single logical unit with up to 660MB of storage. Using this interface, files can be opened and read with the normal Int 21H function calls. The CD-ROM extensions also support a set of special function calls that provide CD-ROM-specific information.

Most Microsoft CD-ROM Extensions function calls return Microsoft CD-ROM Extension error values. To interpret the return values, use the `getlasterror` function.

This set of function calls uses one static variable, defined as `int lasterror`. This variable holds the last error value when a flag is set upon return from the int 2F call.

You can find assembly language prototypes of the CD-ROM functions and variables in `MSCDEX.ASM` file. You can find "C" language prototypes in the `MSCDEX.PRO` file. To accommodate all memory models, all pointers are prototyped as *far.*

Note: Make sure to compile modules with byte-aligned structure elements.

ismscdex

Use `ismscdex` to determine if the Microsoft CD-ROM Extensions are installed on your system.

Calling Convention

```
int ismscdex (void);
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer	0	MSCDEX is installed.
		non-zero	MSCDEX is not installed.

Related topics

None.

getnumcdroms

Use `getnumcdroms` to determine the number of CD-ROM drives installed in the PC.

Calling Convention

```
int getnumcdroms (void);
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer		Indicates the total number of CD-ROM drives installed in the PC.

Related topics

None.

getfirstcdrom

Use `getfirstcdrom` to determine the drive number of the first CD-ROM drive installed in the PC.

Calling Convention

```
int getfirstcdrom (void);
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer		Indicates the number of the first CD-ROM drive installed in the PC.

Related topics

None.

getcdromlist

Use `getcdromlist` to fill a buffer with drive identifiers and addresses.

Calling Convention

```
int getcdromlist (struct far *cdromdrives);
```

Input Parameters

Parameter	Type	Value	Description
struct far *cdrom-drives	pointer		Pointer to list of CD-ROM drive identifiers and addresses.

Return Values

Parameter	Type	Value	Description
int	integer	0	Function call was successful.
		-1	Function call failed.

Related topics

None.

getcopyrightfname

Use `getcopyrightfname` to fill a buffer with the name of the copyright file for the specified drive.

Calling Convention

```
int getcopyrightfname (int cddrive, char far *copyrightfname);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive.
*copyrightfname	pointer		Pointer to the copy right name file on the CD.

Return Values

Parameter	Type	Value	Description
int	integer	0	Function call was successful.
		-1	Function call failed.

Related topics

None.

getabstractfname

Use `getabstractfname` to fill a buffer with the name of the abstract file for the specified drive.

Calling Convention

```
int getabstractfname (int cddrive, char far *abstractfname);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive.
*abstractfname	pointer		Pointer to the abstract file on the CD.

Return Values

Parameter	Type	Value	Description
int	integer	0	Function call was successful.
		-1	Function call failed.

Related topics

None.

getbibliofname

Use `getbibliofname` to fill a buffer with the name of the bibliography file for the specified drive.

Calling Convention

```
int getbibliofname (int1 cddrive, char far *bibliofname);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive.
*bibliofname	pointer		Pointer to the bibliography file on the CD.

Return Values

Parameter	Type	Value	Description
int	integer	0	Function call was successful.
		-1	Function call failed.

Related topics

None.

readvtoc

Use `readvtoc` to read an entry from the volume table of contents for the specified drive.

Calling Convention

```
int readvtoc (int cddrive, int index, char far *dscbuf);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive.
index	integer		Specifies index number.
*dscbuf	pointer		Pointer to the CD buffer.

Return Values

Parameter	Type	Value	Description
int	integer		Indicates type of descriptor.
		-2	Function call failed.

Related topics

None.

absdiscread

Use `absdiscread` to read one or more logical sectors from the specified drive.

Calling Convention

```
int absdiscread (int cddrive, int count, long sector, char far
*buffer);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive.
count	integer		Specifies the number of sectors to read.
sector			Specifies the starting sector address.
*buffer	pointer		Pointer to the buffer to receive sector data.

Return Values

Parameter	Type	Value	Description
int	integer		Indicates number of sectors that were read.
		-1	Function call failed.

Related topics

None.

absdiscwrite

Use `absdiscwrite` to write one or more logical sectors to the specified drive.

Calling Convention

```
int absdiscwrite (int drive, int count, long sector, char far
*buffer);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive.
count	integer		Specifies number of sectors to write.
sector			Specifies the number of sectors to write.
*buffer	pointer		Pointer to the buffer to write.

Return Values

Parameter	Type	Value	Description
int	integer		Indicates number of sectors that were written.
		-1	Function call failed.

Related topics

None.

chkdrive

Use `chkdrive` to determine that the specified drive supported by MSCDEX.

Calling Convention

```
int chkdrive (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive.

Return Values

Parameter	Type	Value	Description
int	integer	0	MSCDEX is installed and the drive is supported.
		1	MSCDEX is installed and the drive is not supported.
		-1	MSCDEX is not installed.

Related topics

None.

getmscdexversion

Use `getmscdexversion` to determine the version of MSCDEX installed.

Calling Convention

```
int getmscdexversion (void);
```

Input Parameters
None.

Return Values

Parameter	Type	Value	Description
int	integer		Indicates version number of MSCDEX installed.

Related topics
None.

getcdromunits

Use `getcdromunits` to fill a buffer with a list of CD-ROM drives.

Calling Convention

```
int getcdromunits (char far *cdromunits);
```

Input Parameters

Parameter	Type	Value	Description
* cdromunits	far pointer		Pointer to a buffer to receive a list of CD-ROM drive numbers.

Return Values

Parameter	Type	Value	Description
int	integer		Function call was successful.
			Function call failed.

Related topics
None.

getvdescpref

Use getvdescpref to get preference for primary or supplementary descriptors.

Calling Convention

```
int getvdescpref (int cddrive);
```

Input Parameters

Parameter	Type	Value	Description
int cddrive	integer		Specifies the target CD-ROM drive.

Return Values

Parameter	Type	Value	Description
int	integer	0	Indicates preference for primary descriptors.
		1	Indicates preference for supplementary descriptors.
		-1	Function call failed.

Related topics

None.

setvdescpref

Use `setvdescpref` to set preference for primary or supplementary descriptors.

Calling Convention

```
int setvdescpref (int cddrive, int pref);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive.
pref	integer		Specifies preference for version description.

Return Values

Parameter	Type	Value	Description
int	integer	0	Function call was successful.
		-1	Function call failed.

Related topics

None.

getdirentry

Use `getdirentry` to search directory for entry, fill a buffer if the directory is found.

Calling Convention

```
int getdirentry (int drive, char far *name, char far *buffer);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive.
*name	far pointer		Pointer to the name of the entry to search for.
*buffer	far pointer		Pointer to the buffer to fill.

Return Values

Parameter	Type	Value	Description
int	integer		Format of the target volume.
		-1	Function call failed.

Related topics
None.

senddevreq

Use `senddevreq` to send a device request.

Calling Convention

```
int senddevreq (intl cddrive, struct far *cdreqheader);
```

Input Parameters

Parameter	Type	Value	Description
cddrive	integer		Specifies the target CD-ROM drive.
*cdreqheader	far pointer		Pointer to the cdreqheader data structure.

Return Values

Parameter	Type	Value	Description
int	integer	0	Function call was successful.
		-1	Function call failed.

Related topics

None.

getlasterror

Use `getlasterror` to retrieve the value of the last MSCDEX error.

Calling Convention

```
int getlasterror (void);
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer		Value of last MSCDEX error.
		-1	Function call failed.

Related topics

None.

clearlasterror

Use `clearlasterror` to clear the error variable prior to executing any CD-ROM function call.

Calling Convention

```
int clearlasterror (void);
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer		Value of last error value.
		-1	Function call failed.

Related topics

None.

Mixer Programming Section

20 Mixer Programming Essentials

This chapter describes:

- The MVPROAS mixer device and the MVSOUND.SYS device driver
- Mixer channel and device connections
- Low-level Mixer software API programming steps

The Pro AudioSpectrum mixer blends multiple audio channels from the FM synthesizer, CD-ROM, PC speaker, microphone, digital audio controller, and an external stereo line-in port.

You can program the mixer using either the text string, command line interface or the low-level software API. For information on the text string, command line interface to the mixer, see Chapter 21, "Command Line Mixer Interface." For information on the low-level mixer function calls, see Chapter 22, "Low-Level Mixer Function Call Reference." A direct hardware API is not provided.

MVPROAS Device Driver Overview

MVPROAS is an MS-DOS device that provides easy access to the Pro AudioSpectrum's mixer and volume controls through both a text string, MS-DOS command line interface, and a binary programming interface. The unique design of this driver makes it easy for both end-users and developers to control the Pro AudioSpectrum devices.

MVPROAS text interface provides a natural language interface that lets you issue English-like sentences to control the mixer and volume levels of all the audio devices. End users can use this interface from the DOS prompt, from a DOS application program, or from a Windows program.

The binary programming interface is a set of low-level function calls that are linked with the MVSOUND.SYS device driver. These function calls are similar to the other low-level function calls supporting other features of the Pro AudioSpectrum.

MVPROAS has at least three "behind-the-scenes" duties:

- It initializes the hardware at boot time

  Every time the computer boots, the Pro AudioSpectrum performs an internal reset. Sounds generated prior to booting are silenced.

- After a reset, MVPROAS sets the mixers, volume controls, and other devices to their default states
- It serves as a single, central resource for all applications to share hardware-dependant information like mixer and volume settings, DMA settings, and IRQ settings

Loading and Customizing MVPROAS

The installation procedure described in the Pro AudioSpectrum User's Guide copies the device driver file, `MVSOUND.SYS`, to your PC system disk and sets up a default configuration profile.

`MVSOUND.SYS` is automatically loaded by the MS-DOS device driver loading facility, `CONFIG.SYS`. You access the `MVSOUND.SYS` device driver using the unique device name of *MVPROAS*. MVPROAS is similar to other MS-DOS devices like LPT1 and COM1, and can be accessed in the same way you would access them.

MVSOUND.SYS Command Line Switches

Command line switches tell `MVSOUND.SYS` about the configuration of the PC and specify default volume settings for Pro AudioSpectrum devices. Command line switches are processed once at boot time.

To load the `MVSOUND.SYS` device driver with command line switches, modify your PC's `CONFIG.SYS` file so that the MVSOUND device line looks similar to the one below:

```
DEVICE=\MVSOUND.SYS D:1 Q:7 V:20
```

In order for `MVSOUND.SYS` to process command line switches correctly, be sure to put at least one space between each of the arguments.

`MVSOUND.SYS` command line switches are defined as follows:

Switch	Description
D:xx	Identifies the DMA selection. This switch tells the driver which DMA channel the hardware is set to use. The actual selection is made by a jumper on the Pro AudioSpectrum system. Valid DMA channel numbers are 1, 3, 5, 6, or 7.
Q:xx	Identifies the IRQ selection. Like the DMA selection, this switch only informs the driver which IRQ channel the system has been jumpered to use. Valid IRQ channel numbers are 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, or 15.
V:xx	Identifies the total volume control setting. A numeric value ranging from 0 to 100% sets the volume level, where 0 is the lowest volume (off) and 100% is the highest volume (full on).

Controlling Total Volume From The Keyboard

Once loaded, MVPROAS provides direct keyboard control over the total volume settings. Raise, lower, or mute the total volume control by typing one of the following special key sequences:

Key Sequence	Description
[Ctrl] + [Alt] + [U]	Increases volume
[Ctrl] + [Alt] + [D]	Decreases volume
[Ctrl] + [Alt] + [M]	Toggles the volume mute

Table 8 Total Volume Control Key Sequences

Mixer block diagram

The following diagram shows the logical audio path from the audio source, through the input and output mixers, volume control device, and out the speaker. Note that each audio source can be directed to the input or output mixer.

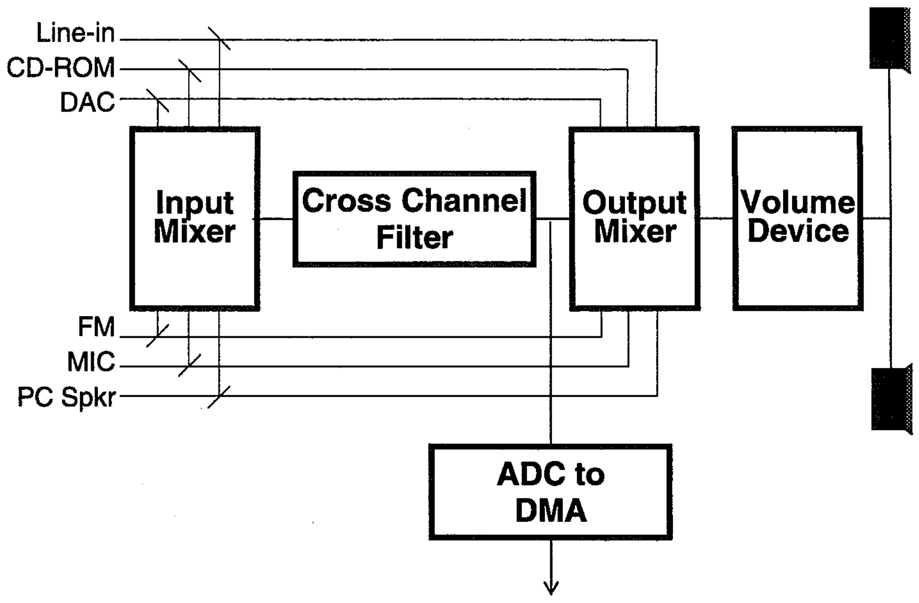


Figure 9 Mixer Block Diagram

Low-level Mixer API programming steps

The following procedure shows the order in which you should use low-level Mixer function calls. The function calls are documented in MIXERS.ASM.

1. **Initialize mixer hardware and software function call library.**

Function call:	`cMVInitMixerCode`

2. **Read from or write to the mixer channel.**

Function calls:	`cMVSetMixerFunction`
	`cMVGetMixerFunction`
	`cMVSetVolumeFunction`
	`cMVGetVolumeFunction`

3. **Perform other mixer functions (can be intermixed with read and write calls).**

Function calls:	`cMVGetFilterFunction`
	`cMVSetFilterFunction`
	`cMVRealSoundSwitch`

21 *Command Line Mixer Interface*

This chapter describes the command line interface of the Pro AudioSpectrum's mixer device, MVPROAS. MVPROAS controls the Pro AudioSpectrum's mixer and total volume control devices.

Topics covered in this chapter include:

- MVPROAS MS-DOS command line syntax
- A reference to the MVPROAS command set
- Controlling MVPROAS devices
- Using the command set interactively and with batch files
- Using the command set from "C" programs

For more general information on using the mixer, see Chapter 20, "Mixer Programming Essentials." For information on the low-level mixer function calls, see Chapter 22, "Low-Level Mixer Function Call Reference."

Command Line Syntax

MVPROAS provides a natural, English-like language interface to the various devices on the Pro AudioSpectrum system. The interface uses a subject/verb/predicate structure where periods and commas are optional. Every command must be entered on a single command line.

The following examples demonstrate the flexibility and intuitiveness of the MVPROAS interface.

```
TURN VOLUME LEFT LEVEL TO 5 PERCENT.

FADE [THE] VOLUME [FOR THE] RIGHT LEVEL TO 100%

SET OUTPUT MIXER [FOR THE] LEFT FM TO 50%

TURN INPUT MIXER FM FROM 0, TO 50%

FADE INPUT MIXER FM FROM 50, TO 0%, IN 3 SECONDS.

SET INPUT MIXER LEFT MIC IN 3 SECONDS, TO 0,FROM 50 PERCENT.

TURN [THE] INPUT MIXER FM UP 50%, IN 3 SECONDS.

TURN INPUT MIXER FM FROM 0, UP 100%, IN 3 SECONDS.

TURN INPUT [THE] MIXER FM FROM 50%, DOWN 50%, IN 3 SECONDS.
```

Some of the keywords are surrounded with brackets (e.g., [THE]). This indicates the keyword is optional and is ignored by MVPROAS.

As you can see from the examples, there are several ways to set device levels. Each volume setting can be changed using up to three arguments:

- *FROM* a starting point

  The FROM starting point is optional. If the starting point is not specified, the current setting is used.

- *TO* a new setting

  The TO setting must be explicitly stated.

- *IN* x amount of time

  The IN command, also optional, tells MVPROAS to adjust the setting over a period of time. MVPROAS takes the difference between the FROM setting and the TO setting, and determines the number of steps needed to complete the task over the time period.

The FROM, TO, and IN portions of the command can be listed in any order. In the next set of examples shows three different ways to adjust the volume level from 0 to 100 percent over a period of three seconds.

```
SET VOLUME LEVEL FROM 0% TO 100% IN 3 SECONDS.

SET VOLUME LEVEL TO 100%, FROM 0%, IN 3 SECONDS.

SET VOLUME LEVEL IN 3 SECONDS, FROM 0%, TO 100%.
```

The following is a formal definition of the MVPROAS language syntax:

Syntax Element	Definition
<statement>	[TURN I SET I FADE] <devcontext>
<devcontext>	INPUT MIXER <term list1> I OUTPUT MIXER <term list1> I VOLUME <term list2> I FILTER <term list3> I CROSS <term list4>
<term list1>	<channels> <setting>
<term list2>	<levelterm> I <loudnessterm> I <enhancedterm>
<term list3>	<MUTE> I <ON I OFF>
<term list4>	<opt> TO <opt> <ON I OFF>
<channels>	[LEFT I RIGHT] <device list>
<device list>	FM I SPEAKER I MIC I EXT I INT I
<levelterm>	[BASS I TREBLE I LEVEL] <setting>
<setting>	[FROM <term>] TOIUPIDOWN <term> [IN <expression> SECONDS]
<loudnessterm>	LOUDNESS <ON I OFF>
<enhancedterm>	ENHANCED <ON I OFF>
<term>	<expression> [PERCENT I%]
<opt>	LEFT I RIGHT
<expression>	<number> I <operators>,...
<number>	'0'.'9'
<operators>	'(' I ')' I '*' I '/' I '+' I '-'

MVPROAS Verbs

The MVPROAS verb set defines the actions you can perform to the Pro AudioSpectrum from the device driver. Some actions require additional information while others are single words.

Fade/Set/Turn

Use these `FADE`, `SET`, and `TURN` to set volume levels or switch settings. These verbs are interchangeable, even when the syntax seems unnatural.

The following examples show MVPROAS commands using the `FADE`, `SET`, and `TURN` verbs:

```
FADE [THE] INPUT MIXER [FOR THE] FM TO 5%

SET [THE] INPUT MIXER [FOR THE] LEFT FM TO 5%

TURN [THE] INPUT MIXER [FOR THE] RIGHT FM TO 5%
```

FADE, SET, and TURN operate on the following devices:

- Input Mixer
- Output Mixer
- Volume
- Mute
- Cross Channel
- Real Sound

Get

GET can only effectively be used within a program, not from the MS-DOS command line. The GET command tells MVPROAS to prepare to fetch the current setting of a device. Issuing a subsequent *read* command to MVPROAS returns the settings as a text message.

The GET command syntax is identical to the FADE/SET/TURN set command, except no FROM/TO/IN operation needs to be specified.

The following examples show MVPROAS commands using the GET verb:

```
GET [THE] INPUT MIXER [FOR THE] FM

GET [THE] OUTPUT MIXER [FOR THE] RIGHT FM

GET [THE] VOLUME [FOR THE] LEFT LEVEL

GET [THE] MUTE

GET [THE] CROSS [FOR THE] LEFT TO LEFT

GET [THE] REALSOUND
```

The GET command operates on all Pro AudioSpectrum devices accessible to the mixer.

Hold

HOLD tells MVPROAS to queue up the next series of commands. You may queue and then execute up to sixteen (16) commands at a time. The HOLD verb is useful for creating cross-channel fades, from one channel to the next or from one device to another.

The following example shows MVPROAS commands using the HOLD verb:

```
HOLD
```

Release

RELEASE tells MVPROAS to start executing the queued commands. If MVPROAS is read any time while processing a queue, it returns the text message "BUSY" until all command entries have been executed.

The following example shows MVPROAS commands using the RELEASE verb:

```
RELEASE
```

Reset

RESET initializes MVPROAS settings to their boot up states.

The following example shows an MVPROAS command using the RESET verb:

```
RESET
```

Controlling MVPROAS devices

The following is a description of the devices and device components you can control using MVPROAS.

Input Mixer

You can control six different stereo channels of the input mixer using MVPROAS:

Input Mixer Channel	**Description**
FM	FM synthesizer.
PCM	PCM playback.
INT	Internal CD connection.
EXT	External stereo jack.
SPEAKER	PC speakers.
MIC	Microphone jack.
SB	Sound Blaster emulation.

You can specify LEFT channel or RIGHT channel of the device. By not indicating LEFT or RIGHT, you implicitly specify both channels.

The following list of examples shows all the simple input mixer commands (with 5% shown as a representative value) you can issue to MVPROAS:

```
SET [THE] INPUT MIXER [FOR THE] FM TO 5%

SET [THE] INPUT MIXER [FOR THE] LEFT FM TO 5%

SET [THE] INPUT MIXER [FOR THE] RIGHT FM TO 5%

SET [THE] INPUT MIXER [FOR THE] PCM TO 5%
SET [THE] INPUT MIXER [FOR THE] LEFT PCM TO 5%
SET [THE] INPUT MIXER [FOR THE] RIGHT PCM TO 5%
SET [THE] INPUT MIXER [FOR THE] INT TO 5%
SET [THE] INPUT MIXER [FOR THE] LEFT INT TO 5%
SET [THE] INPUT MIXER [FOR THE] RIGHT INT TO 5%
SET [THE] INPUT MIXER [FOR THE] EXT TO 5%
SET [THE] INPUT MIXER [FOR THE] LEFT EXT TO 5%
SET [THE] INPUT MIXER [FOR THE] RIGHT EXT TO 5%
SET [THE] INPUT MIXER [FOR THE] SPEAKER TO 5%
SET [THE] INPUT MIXER [FOR THE] LEFT SPEAKER TO 5%
SET [THE] INPUT MIXER [FOR THE] RIGHT SPEAKER TO 5%
SET [THE] INPUT MIXER [FOR THE] MIC TO 5%
SET [THE] INPUT MIXER [FOR THE] LEFT MIC TO 5%
SET [THE] INPUT MIXER [FOR THE] RIGHT MIC TO 5%
```

Output Mixer

You can control six different stereo channels of the output mixer using MVPROAS:

Output Mixer Channel	Description
FM	FM synthesizer.
INPUT	Input mixers connection to the output mixer.
INT	Internal CD connection.
EXT	External stereo jack.
SPEAKER	PC speakers.
MIC	Microphone jack.
SB	Sound Blaster emulation.

With the exception of the PCM input channel and the INPUT output channel, every channel is duplicated on the input and output mixers. Any given input channel can be routed to either the input or output mixer, but not both. The last mixer programmed for any given channel sets the active channel.

For example, programming the FM channels of the output mixer turns off the FM channels on the input mixer. As a result, there is only one stereo input/ output control active per device at a time.

As with the input mixer, you can address each channel on the output mixer as LEFT or RIGHT to specify left channel or right channel. If you don't specify LEFT or RIGHT, you implicitly specify both channels to be modified.

The following is a complete list of all the simple output mixer commands (with 5% shown as a representative value) that you can issue to MVPROAS:

```
SET [THE] OUTPUT MIXER [FOR THE] FM TO 5%

SET [THE] OUTPUT MIXER [FOR THE] LEFT FM TO 5%

SET [THE] OUTPUT MIXER [FOR THE] RIGHT FM TO 5%

SET [THE] OUTPUT MIXER [FOR THE] INPUT TO 5%

SET [THE] OUTPUT MIXER [FOR THE] LEFT INPUT TO 5%

SET [THE] OUTPUT MIXER [FOR THE] RIGHT INPUT TO 5%

SET [THE] OUTPUT MIXER [FOR THE] INT TO 5%

SET [THE] OUTPUT MIXER [FOR THE] LEFT INT TO 5%

SET [THE] OUTPUT MIXER [FOR THE] RIGHT INT TO 5%

SET [THE] OUTPUT MIXER [FOR THE] EXT TO 5%

SET [THE] OUTPUT MIXER [FOR THE] LEFT EXT TO 5%

SET [THE] OUTPUT MIXER [FOR THE] RIGHT EXT TO 5%

SET [THE] OUTPUT MIXER [FOR THE] SPEAKER TO 5%

SET [THE] OUTPUT MIXER [FOR THE] LEFT SPEAKER TO 5%

SET [THE] OUTPUT MIXER [FOR THE] RIGHT SPEAKER TO 5%

SET [THE] OUTPUT MIXER [FOR THE] MIC TO 5%

SET [THE] OUTPUT MIXER [FOR THE] LEFT MIC TO 5%

SET [THE] OUTPUT MIXER [FOR THE] RIGHT MIC TO 5%
```

Volume

The Volume control device on the Pro AudioSpectrum is a combination of on/off switches and volume level controls.

You can control five different device elements using the volume control device of MVPROAS:

Device Element	Description
Enhanced Bass and Treble	On/off switches that provide an absolute boost when turned on.
Bass and Treble	The equalizer bands on the volume control device. Both Bass and Treble has its own level setting. You set each device with a percentage: from 0 percent (-12 dB) to 100 percent (+12 dB).
Volume	Sets left and right channel, either programmed independently or together as a single device

The following is a list of example volume commands that you can issue to MVPROAS:

```
SET [THE] VOLUME ENHANCED BASS [TO] ON

SET [THE] VOLUME ENHANCED BASS [TO] OFF

SET [THE] VOLUME ENHANCED TREBLE [TO] ON

SET [THE] VOLUME ENHANCED TREBLE [TO] OFF

SET [THE] VOLUME BASS TO 5%

SET [THE] VOLUME TREBLE TO 100%

SET [THE] VOLUME LEVEL TO 0%

SET [THE] VOLUME LEFT LEVEL TO 100%

SET [THE] VOLUME RIGHT LEVEL TO 50%
```

Mute

The Mute device lets you completely mute the external output jack. When Mute is set to ON, the PC speaker is the only active sound device. All other audio devices are silent.

Program Mute using one of the following commands:

```
SET [THE] MUTE [TO] ON

SET [THE] MUTE [TO] OFF
```

Cross Channel

Residing between the input mixer and the filter, the Cross Channel device lets you mix and match stereo input and output channels in interesting combinations. The following diagram shows that there are four possible inputs and four possible outputs for a total 16 possible combinations. Each of the inputs is either ON or OFF.

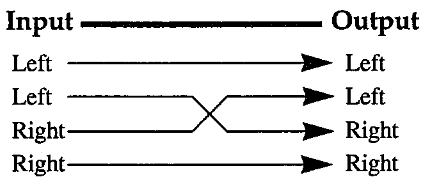


Figure 10 Cross Channel Connections

You can set the Cross Channel device in any of the possible combinations by issuing up to four separate commands. Following is a complete list of all the possible Cross Channel commands:

```
SET [THE] CROSS [CHANNEL] LEFT TO LEFT [TO] OFF

SET [THE] CROSS [CHANNEL] LEFT TO LEFT [TO] ON

SET [THE] CROSS [CHANNEL] LEFT TO RIGHT [TO] OFF

SET [THE] CROSS [CHANNEL] LEFT TO RIGHT [TO] ON

SET [THE] CROSS [CHANNEL] RIGHT TO LEFT [TO] OFF

SET [THE] CROSS [CHANNEL] RIGHT TO LEFT [TO] ON

SET [THE] CROSS [CHANNEL] RIGHT TO RIGHT [TO] OFF

SET [THE] CROSS [CHANNEL] RIGHT TO RIGHT [TO] ON

SET [THE] CROSS [CHANNEL] [TO] ON

SET [THE] CROSS [CHANNEL] [TO] OFF
```

Note: The command `SET CROSS CHANNEL ON` enables all connections: LEFT to LEFT, LEFT to RIGHT, RIGHT to LEFT, and RIGHT to RIGHT. This creates a monaural composite of both inputs.

The command `SET CROSS CHANNEL OFF` disables connections and effectively mutes the system.

Real Sound

The Real Sound device lets you switch the improved "Real Sound" hardware on or off to enhance a unique technology employed in certain video games called "Real Sound." This technology drives the PC speaker in a way that results in amazingly clear pre-recorded audio playback.

You control the Real Sound device using the following commands:

```
SET [THE] REALSOUND [TO] OFF

SET [THE] REALSOUND [TO] ON
```

Using MS-DOS Commands With MVPROAS

You can communicate with MVPROAS from either the MS-DOS command line or from batch programs by using the MS-DOS ECHO and COPY commands.

ECHO Command

The easiest way to pass a command to MVPROAS is to use the MS-DOS ECHO command. This command takes whatever data you type and writes it to the device. You can redirect the write to any MS-DOS device by specifying an appropriate device name. The following example shows a RESET command being sent to MVPROAS:

```
ECHO >MVPROAS RESET
```

The greater than symbol (>) precedes the device name (MVPROAS). This is the standard MS-DOS method for redirecting output to the designated device. Since the ECHO command is a standard MS-DOS command, it can be placed in batch files.

The following example, when entered in a batch file, fades in the PC Speaker while fading out the microphone:

```
ECHO >MVPROAS HOLD

ECHO >MVPROAS SET OUTPUT MIXER MIC TO 0 IN 2 SECONDS

ECHO >MVPROAS SET OUTPUT MIXER SPEAKER TO 75 PERCENT IN 2 SECONDS

ECHO >MVPROAS RELEASE
```

COPY Command

The COPY command is similar to the ECHO command, but allows multiple lines of text to be sent to the device at one time.

In the following example, the four lines of text below are contained in a file called FADE:

```
HOLD

SET OUTPUT MIXER MIC TO 0 IN 2 SECONDS

SET OUTPUT MIXER SPEAKER TO 75% IN 2 SECONDS

RELEASE
```

To execute the fade-in, fade-out procedure described in the under the ECHO command, type this command at the DOS prompt:

```
COPY FADE MVPROAS
```

Controlling MVPROAS From Programs

You can send text to MVPROAS with a simple C program. The following code sample opens the MVPROAS device, sends a command, closes the device, and exits to DOS. For this example, assume the program sets the total volume by accepting a percentage from the command line. Error checking code is not included.

```
main(argc,argv)

     int argc;

     char *argv[];

{

FILE *proas;

     /* open the device */

     if ((proas = fopen ("MVPROAS", "w")) == 0) {

         printf ("\cannot open the device!\n");

         exit(1);}

     /* send out the text, then we can exit */

     fprintf (proselyte VOLUME LEVEL TO %S%%\n",argv[1]);

     fclose (proas);

     exit(0);

}
```

22 *Low-Level Mixer Function Call Reference*

This section provides a complete reference to the low-level mixer function calls. Use these calls to modify the mixer and volume settings. Unless otherwise specified, you can find prototypes of each of the calls described here in the `MIXERS.H` file.

cMVInitMixerCode

Use `cMVInitMixerCode` to link software library with `MVSOUND.SYS` to give programs access to the mixers and other related functions. This function is called only once, and must be called before any of the mixer routines are called. It does not initialize the Pro AudioSpectrum board.

Calling Convention

```
int MVInitMixerCode (void);
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer	0	No driver found.
		1	MVSOUND.SYS driver was found.

Related Topics

None.

cMVGetFilterFunction

Use `cMVGetFilterFunction` to determine the current setting of the PCM (digital audio) low-pass filter. This function returns a value, stated as a percentage of the total data frequency range of data that will pass through the Pro AudioSpectrum board, that indicates the highest allowable frequency for recording data.

Calling Convention

```
int cMVGetFilterFunction ();
```

Input Parameters

None.

Return Values

Parameter	Type	Value	Description
int	integer	100	Limiting frequency set to 18 kHz (near CD quality).
		84	Limiting frequency set to 16 kHz (cassette quality).
		67	Limiting frequency set to 12 kHz (FM radio quality).
		50	Limiting frequency set to 9 kHz (AM radio quality).
		34	Limiting frequency set to 6 kHz (telephone quality).
		17	Limiting frequency set to 3 kHz (male voice quality).
		0	Limiting frequency set to 0 kHz (mute).

Related Topics

To program settings for the low-pass filter, see"cMVSetFilterFunction" on page 22-6.

cMVGetMixerFunction

Use `cMVGetMixerFunction` to determine the signal level setting for the specified channel. This function returns a two-byte value. The low byte indicates the source channel's level of attenuation. The high byte indicates whether the mixer is on or off.

Calling Convention

```
int cMVGetMixerFunction (int mixer, int channel);
```

Input Parameters

Parameter	Type	Value	Description
mixer	integer	BI_OUTPUTMIXER	Select output mixer.
		BI_INPUTMIXER	Select input mixer.
channel	integer	DX = BI_L_FM	Select left FM synthesizer.
		BI_R_FM	Select right FM synthesizer.
		BI_L_IMIXER	Select left input mixer.
		BI_R_IMIXER	Select right input mixer.
		BI_L_EXT	Select left line in.
		BI_R_EXT	Select right line in.
		BI_L_INT	Select left internal (CD) audio.
		BI_R_INT	Select right internal (CD) audio.
		BI_L_MIC	Select left microphone.
		BI_R_MIC	Select right microphone.
		BI_L_PCM	Select left PCM.
		BI_R_PCM	Select right PCM.
		BI_L_SPEAKER	Select left PC speaker.
		BI_R_SPEAKER	Select right PC speaker.
		BI_L_SBDAC	Select left Sound Blaster channel.
		BI_L_SBDAC	Select left Sound Blaster channel.

Return Values

Parameter	Type	Value	Description
int	integer	Low byte: 0 to 100	Indicates level of attenuation. 0 indicates almost complete attenuation (-80 db), 100 indicates no attenuation (0 db).
		High byte: 0	Mixer off.
		High byte: 1	Mixer on.

Related Topics

To set the signal level for a specified channel, see "cMVSetMixerFunction" on page 22-7.

To determine the current settings for the total volume control device, see "cMVGetVolumeFunction" on page 22-4.

cMVGetVolumeFunction

Use `cMVGetVolumeFunction` to determine the current settings for the total volume control device.

Calling Convention

```
int cMVGetVolumeFunction (int level, int device);
```

Input Parameters

Parameter	Type	Value	Description
level	integer	0	If device = BI_VOLLOUD, loudness off.
			If device = BI_VOLENHANCE, enhanced stereo off.
		100	If device = BI_VOLLOUD, loudness on.
			If device = BI_VOLENHANCE, enhanced stereo on.
		0 to 100	If device = BI_VOLBASS or BI_VOLTRE-BLE, bass or treble equalizer band set to cut (0), no change (50), or maximum gain (100).
			If device = BI_VOLLEFT or BI_VOL-RIGHT, left or right channel volume set to nearly complete attenuation (0) or no atten-uation (100). Nearly complete attenuation is 80dB. No attenuation is 0dB.
device	integer	BI_VOLLOUD	Select loudness device.
		BI_VOLENHANCE	Select enhanced stereo device.
		BI_VOLBASS	Select bass equalizer band device.
		BI_VOLTREBLE	Select treble equalizer band device.
		BI_VOLLEFT	Select left channel volume device.
		BI_VOLRIGHT	Select right channel volume device.

Return Values

Parameter	Type	Value	Description
int	integer	0	Function call failed.
		1	Function call succeeded.

Related Topics

To set the total volume control device levels, see "cMVSetVolumeFunction" on page 22-9.

To determine the signal level setting for the specified channel, see "cMVGetMixerFunction" on page 22-3.

cMVRealSoundSwitch

Use `cMVRealSoundSwitch` to set or read the status of the Pro AudioSpectrum's real sound circuitry. Valid settings are on or off.

The Pro AudioSpectrum's real sound device the PC's native sound device. To hear *normal* PC speaker sound, disable real sound. Real sound circuitry should be turned on only for video games.

You can find a prototype of this function in the `BINARY.H` file.

Calling Convention

```
int cMVRealSoundSwitch (int state, int mode);
```

Input Parameters

Parameter	Type	Value	Description
state	integer	0	For read mode, real sound disabled.
			For write mode, disable real sound.
		100	For read mode, real sound enabled.
			For write mode, enable real sound.
mode	integer	0	Set mode to read settings.
		1	Set mode to write settings.

Return Values

Parameter	Type	Value	Description
int	integer	0	Function call failed.
		1	Function call succeeded.

Related Topics

None.

cMVSetFilterFunction

Use `cMVSetFilterFunction` to set the frequency limit of the digital audio filter. The filter causes frequencies above the limit to be removed while leaving frequencies below the limit un-modified. Filter settings are specified as a percent of the total frequency range.

The digital audio filter resides between the input and output mixer and is useful for recording and playing back PCM sounds. When recording, it eliminates frequencies that cause aliasing errors. When playing back, it removes higher harmonics generated as a by-product of digital to analog conversion.

For playback or recording, set the filter to a frequency no greater than half the sample rate.

Calling Convention

```
void cMVSetFilterFunction (int filter);
```

Input Parameters

Parameter	Type	Value	Description
filter	integer	100	Set limiting frequency to 18 kHz (near CD quality).
		84	Set limiting frequency to 16 kHz (cassette quality).
		67	Set limiting frequency to 12 kHz (FM radio quality).
		50	Set limiting frequency to9 kHz (AM radio quality).
		34	Set limiting frequency to 6 kHz (telephone quality).
		17	Set limiting frequency to 3 kHz (male voice quality).
		0	Set limiting frequency to 0 kHz (mute).

Return Values

None.

Related Topics

To determine the current setting of the PCM low-pass filter, see "cMVGetFilterFunction" on page 22-2.

cMVSetMixerFunction

Use `cMVSetMixerFunction` to configure channels and mixers and set the signal level for a specified channel. The level can vary between 0 (off) and 100 (maximum).

The power-on default mixer settings specify all sources off. MVSOUND.SYS will route all the channels to the input mixer. Since any given source can be routed to either the input or output mixer, but not both simultaneously, a call that 'connects' a source to one mixer will automatically cut-off the signal from reaching the other mixer.

The mixers are high fidelity devices designed to handle the entire audio spectrum from 20 Hz to 20 kHz with low distortion. You route a signal source to the input mixer to convert an audio input signal to digital PCM data, or to convert digital PCM data to audio with subsequent mixing with other audio sources in the output mixer. See "Mixer block diagram" on page 20-4 for a depiction of mixer configurations.

Note: Make sure to turn off the input mixer PCM channels before recording to prevent feedback.

Note: The channels `BI_L_IMIXER` and `BI_R_IMIXER` must only be sent to the output mixer.

Calling Convention

```
void cMVSetMixerFunction (int level, int mixer, int channel);
```

Input Parameters

Parameter	Type	Value	Description
level	integer	0 to 100	Set channel attenuation level. 0 is almost complete attenuation (~80dB); 100 is no attenuation (0 dB).
mixer	integer	BI_OUTPUTMIXER	Select output mixer.
		BI_INPUTMIXER	Select input mixer.
channel	integer	BI_L_FM	Select left FM synthesizer.
		BI_R_FM	Select right FM synthesizer.
		BI_L_IMIXER	Select left input mixer.
		BI_R_IMIXER	Select right input mixer.
		BI_L_EXT	Select left line in.
		BI_R_EXT	Select right line in.
		BI_L_INT	Select left internal (CD) audio.
		BI_R_INT	Select right internal (CD) audio.
		BI_L_MIC	Select left microphone.
		BI_R_MIC	Select right microphone.
		BI_L_PCM	Select left PCM.
		BI_R_PCM	Select right PCM.
		BI_L_SPEAKER	Select left PC speaker.
		BI_R_SPEAKER	Select right PC speaker.
		BI_L_SBDAC	Select left Sound Blaster DAC.
		BI_R_SBDAC	Select right Sound Blaster DAC.

Return Values

None.

Related Topics

To determine the signal level setting for the specified channel, see "cMVGetMixerFunction" on page 22-3.

To set the total volume control device levels, see "cMVSetVolumeFunction" on page 22-9.

cMVSetVolumeFunction

Use `cMVSetVolumeFunction` to set the total volume control device levels. Volume control device levels shape the Pro AudioSpectrum sound by controlling the relative strength of signal sources flowing into the mixer.

Together with the total volume control circuitry, this function provides controls similar to those found on a speaker amplifier:

- Loudness on/off
- Bass boost/cut
- Treble boost/cut
- Left and right channel volume

An additional switch, enhanced stereo, is also provided to force increased stereo channel separation, enhancing the stereo effect, or give a slight stereo effect to what would otherwise be normal mono sound.

Calling Convention

```
void cMVSetVolumeFunction (int level, int device);
```

Input Parameters

Parameter	Type	Value	Description
level	integer	0	If device = BI_VOLLOUD, set loudness off.
			If device = BI_VOLENHANCE, set enhanced stereo off.
		100	If device = BI_VOLLOUD, set loudness on.
			If device = BI_VOLENHANCE, set enhanced stereo on.
device		BI_VOLLOUD	Select loudness device.
		BI_VOLENHANCE	Select enhanced stereo device.
		BI_VOLBASS	Select bass equalizer band device.
		BI_VOLTREBLE	Select treble equalizer band device.
		BI_VOLLEFT	Select left channel volume device.
		BI_VOLRIGHT	Select right channel volume device.

Return Values

None.

Related Topics

To determine the current settings for the total volume control device, see "cMVGetVolumeFunction" on page 22-4.

To set the signal level for a specified channel, see "cMVSetMixerFunction" on page 22-7.

Appendices

A *FM Hardware Register Charts and Tables*

This appendix provides a variety of information including:

- Rate-to-time conversion tables
- Key scaling level tables
- Standard Pitch Values

Rate to Time Conversion Tables

Use the the following tables to compute attack, decay, release values for Attack/Decay Rate (60h to 75h) and Sustain Level/Release Rate Registers (80h to 95h).

These tables list time periods (in milliseconds) for the full spread of rate values (63 to 0) and show two durations for each rate value:

- 10% to 90% (or 90% to 10%)

  Indicates the time period over which the envelope makes 80% of the attack, decay, or release transition.

- 0% to 100% (or 100% to 0%)

  Indicates the total duration for the attack, decay, or release.

Note that a sustain column is not provided; the sustain bits specify a level, in decibels below the peak of envelope, for the sustain period. The sustain period is maintained as long as the note is played (Key On is set to 1).

Rate	Attack Time (in milliseconds) (10%-90%)	(0%-100%)	Decay/Release Time (in milliseconds) (90%-10%)	(100%-0%)
63	0.00	0.00	0.51	2.40
62	0.00	0.00	0.51	2.40
61	0.00	0.00	0.51	2.40
60	0.00	0.00	0.51	2.40
59	0.11	0.20	0.58	2.74
58	0.11	0.24	0.63	3.20
57	0.14	0.30	0.81	3.84
56	0.19	0.38	1.01	4.80
55	0.22	0.42	1.15	5.48
54	0.26	0.46	1.35	6.40
53	0.31	0.56	1.62	7.68
52	0.37	0.70	2.02	9.60
51	0.43	0.80	2.32	10.96
50	0.49	0.92	2.68	12.80
49	0.61	1.12	3.22	15.36
48	0.73	1.40	4.02	19.20
47	0.85	1.56	4.62	21.92
46	0.97	1.84	5.38	25.56
45	1.13	2.20	6.42	30.68

Table 9 Rate Table For Rates 63 to 45

*10%-90% is equivalent to -86.4 dB to -9.6 dB

0%-100% is equivalent to -96 dB to 0 dB

Rate	Attack Time (in milliseconds) (10%-90%)	(0%-100%)	Decay/Release Time (in milliseconds) (90%-10%)	(100%-0%)
44	1.45	2.76	8.02	38.36
43	1.70	3.12	9.24	43.84
42	1.94	3.68	10.76	51.12
41	2.26	4.40	12.84	61.36
40	2.90	5.52	16.04	76.72
39	3.39	6.24	18.48	87.68
38	3.87	7.36	21.52	102.24
37	4.51	8.80	25.68	122.72
36	5.79	11.04	32.08	153.44
35	6.78	12.48	36.96	175.36
34	7.74	14.72	43.04	204.48
33	9.02	17.60	51.36	245.44
32	11.58	22.08	64.16	306.88
31	13.57	24.96	73.92	350.72
30	15.49	29.44	86.08	408.96
29	18.05	35.20	102.72	490.88
28	23.17	44.16	128.32	613.76
27	27.14	49.92	147.84	701.44
26	30.98	58.88	172.16	817.92
25	36.10	70.40	205.44	981.76

Table 10 Rate Tables For Rates 44 to 25

*10%-90% is equivalent to -86.4 dB to -9.6 dB

0%-100% is equivalent to -96 dB to 0 dB

Rate	Attack Time (in milliseconds) (10%-90%)	(0%-100%)	Decay/Release Time (in milliseconds) (90%-10%)	(100%-0%)
24	46.34	88.32	256.64	1227.52
23	54.27	99.84	295.68	1402.88
22	61.95	117.76	344.32	1635.84
21	72.19	140.80	410.88	1963.52
20	92.67	176.84	513.28	2455.04
19	108.54	199.68	591.36	2805.76
18	123.90	235.52	688.64	3271.68
17	144.38	281.60	821.76	3927.04
16	185.34	353.28	1026.56	4910.08
15	217.09	399.36	1182.72	5611.52
14	247.81	471.04	1377.28	6543.36
13	288.77	563.20	1643.52	7854.08
12	370.69	706.56	2053.12	9820.16
11	434.18	798.72	2365.44	11223.04
10	495.62	942.08	2754.56	13086.72
9	577.54	1126.40	3287.04	15708.16
8	741.38	1413.12	4106.24	19640.32
7	868.35	1597.44	4730.88	22446.08
6	991.23	1884.16	5509.12	26173.44
5	1155.04	2252.80	6574.08	31416.32
4**	1482.75	2826.24	8212.48	39280.64

Table 11 Rate Table For Rates 24 To 4

*10%-90% is equivalent to -86.4 dB to -9.6 dB

0%-100% is equivalent to -96 dB to 0 dB

**There is no waveform for rates less than 4.

Key Scaling Level Tables

The two tables that follow list the key scaling level attenuation for each octave, and for notes within the octave, for the 3 dB/octave KSL setting of the KSL/ Total Level registers (40h to 55h). The first table shows the attenuation (in dB) for octaves 0 through 3, while the second shows the same for octaves 4 through 7.

Note that you can calculate the 1.5 dB and 6 dB KSL attenuation by halving and doubling the values shown in this table.

F-Num High Nibble	Octave 0	Octave 1	Octave 2	Octave 3
0	0.000	0.000	0.000	0.000
1	0.000	0.000	0.000	0.000
2	0.000	0.000	0.000	0.000
3	0.000	0.000	0.000	1.875
4	0.000	0.000	0.000	3.000
5	0.000	0.000	1.125	4.125
6	0.000	0.000	1.875	4.875
7	0.000	0.000	2.625	5.625
8	0.000	0.000	3.000	6.000
9	0.000	0.750	3.750	6.750
10	0.000	1.125	4.125	7.125
11	0.000	1.500	4.500	7.500
12	0.000	1.875	4.875	7.875
13	0.000	2.250	5.250	8.250
14	0.000	2.625	5.625	8.625
15	0.000	3.000	6.000	9.000

Table 12 Key Scaling Levels for Octaves 0 Through 3

* The value of the F-Number four most significant bits

**1.5 dB values are one-half of these; 6 dB are twice

F-Num High Nibble	Octave4	Octave 5	Octave 6	Octave 7
0	0.000	0.000	0.000	0.000
1	0.000	3.000	6.000	9.000
2	3.000	6.000	9.000	12.000
3	4.875	7.875	10.875	13.875
4	6.000	9.000	12.000	15.000
5	7.125	10.125	13.125	16.125
6	7.875	10.875	13.875	16.125
7	8.625	11.625	14.625	17.625
8	9.000	12.000	15.000	18.000
9	9.750	12.750	15.750	18.750
10	10.125	13.125	16.125	19.125
11	10.500	13.500	16.500	19.500
12	10.875	13.875	16.875	19.875
13	11.250	14.250	17.250	20.250
14	11.625	14.625	17.625	20.625
15	12.000	15.000	18.000	21.000

Table 13 Key Scaling Levels for Octaves 4 Through 7

* The value of the F-Number four most significant bits

**1.5 dB values are one-half of these; 6 dB are twice

Standard Pitch Values

The tables below show standard pitch values for all notes within the eight octave range of the FM synthesizer. Use them to set F-Numbers in Block and F-Number registers (A0h to A8h and B0h to B8h).

NOTE: OCTAVE	C	C#	D	D#
0	16.35	17.32	18.35	19.45
1	32.70	34.65	36.71	38.89
2	65.41	69.30	73.42	77.78
3	130.81	138.59	146.83	155.56
4	261.63	277.18	293.66	311.13
5	523.25	554.37	587.33	622.25
6	1046.50	1108.73	1174.66	1244.51
7	2093.00	2217.46	2349.32	2489.02
8	4186.01			

Table 14 Standard Pitch Values: C, C#, D, and D#

NOTE: OCTAVE	E	F	F#	G
0	20.60	21.83	23.12	24.50
1	41.20	43.65	46.25	49.00
2	82.41	87.31	92.50	98.00
3	164.81	174.61	185.00	196.00
4	329.63	349.23	369.99	392.00
5	659.26	698.46	739.99	783.99
6	1328.51	1396.91	1479.98	1567.98
7	2637.02	2793.83	2959.96	3135.96

Table 15 Standard Pitch Values: E, F, F#, and G

NOTE: OCTAVE	G#	A	A#	B
0	25.96	27.50	29.14	30.87
1	51.91	55.00	58.27	61.74
2	103.83	110.00	116.54	123.47
3	207.65	220.00	233.08	246.94
4	415.30	440.00	466.16	493.88
5	830.61	880.00	932.33	987.77
6	1661.22	1760.00	1864.66	1975.53
7	3322.44	3520.00	3729.31	3951.07

Table 16 Standard Pitch Values: G#, A, A#, and B

B CD-ROM Data Structures and Definitions

This appendix lists the data structures and data definitions you will need to know in order to use the CD-ROM API's.

Request header structure

The following structure is called the *request header.* Use it to communicate directly with the CD-ROM device driver. This structure is an integral part of all the other following defined structures.

struct cdreqheader

```
{
   char len;                         // always 13 - size of (cdreqheader)
   char unit;                        // the drive #
   char cmd;                         // the command, defined above
   int stat;                         // set by device driver
   char reserved0[8];
};
```

struct ioctlread

```
{
  struct cdreqheader cdh;        //see above
  char mdb;                      // "media descriptor byte", usually 0
  void far *buffer;              // address of additional command
                                    information
  int size;                      // size of additional command
                                    information
  int ssn;                       // "starting sector number", usually
                                    0
  void far *errbuf;              // pointer to vol ID if error 0Fh,
                                    usually 0
};
```

struct ioctlwrite

```
{
  struct cdreqheader cdh;        // see above
  char mdb;                      // same as "ioctlread"
  void far *buffer;              // same as "ioctlread"
  int size;                      // same as "ioctlread"
  int ssn;                       // same as "ioctlread"
  void far *errbuf;              // same as "ioctlread"
};
```

struct ioctlseek

```
{
  struct cdreqheader cdh;        // see above
  char addrmode;                 // ****
  void far *buffer;              // ignored
  int sectorcount;               // ignored
  long startsector;              // frame address as seek destination
};
```

struct ioctlplay

```
{
  struct cdreqheader cdh;       // see above
  char addrmode;                // ****
  long startsector;             // frame address to start play
  long sectorcount;             // frame address to end play
};
```

struct ioctlstop

```
{
  struct cdreqheader cdh;       // see above
};
```

struct ioctlresume

```
{
  struct cdreqheader cdh;       // see above
};
```

struct ioctlstat

```
{
  char cmd;                     // CD_GETAUDIOSTAT
  int status;                   // returned value
  long startloc;                // returned last start frame address
  long endloc;                  // returned last end frame address
};
```

struct cdtable

```
{
  struct discinfo di;
  struct trackinfo * ti;
}
```

struct discinfo

```
{
   char cmd;                           // CD_GETDISCINFO
   char strk;                          // returned first track # on disc
   char ltrk;                          // returned last track # on disc
   long eodisc;                        // returned last frame+1 on disc
};
```

struct trackinfo

```
{
   char cmd;                           // CD_GETTRACKINFO
   char track;                         // set track #
   char frame;                         // returned frames on track
   char sec;                           // returned seconds on track
   char min;                           // returned minutes on track
   char dummy;                         // not used
   char control;                       // returned track info
};
```

struct qchaninfo

```
{
   char cmd;                           // CD_GETQCHANINFO
   char caa;                           // returned control and adr byte (?)
   char track;                         // returned current track in BCD
   char index;                         // returned point
   char min;                           // returned minute in track
   char sec;                           // returned second in track
   char frame;                         // returned frame in track
   char reserved1;
```

```
  char amin;               // returned minute in disc
  char asec;               // returned second in disc
  char aframe;             // returned frame in disc
};
```

struct redbookaddr

```
{
  char frame;              // 0x000000FF
  sec;                     // 0x0000FF00
  min;                     // 0x00FF0000
  unknown;                 // 0xFF000000
};
```

Data Definitions

These values are placed in the `cdreqheader.cmd` field:

```
  #define IOCTL_READ          3
  #define IOCTL_WRITE         12
  #define CD_CMD_EJECT        0
  #define CD_CMD_RESET        2
  #define CD_CMD_SEEK         131
  #define CD_CMD_PLAY         132
  #define CD_CMD_STOP         133
  #define CD_CMD_RESUME       136
```

These values are used with the IOCTL_READ command:

```
  #define CD_GETDISCINFO      10
  #define CD_GETTRACKINFO     11
  #define CD_GETQCHANINFO     12
  #define CD_GETAUDIOSTAT     15
```

These values are used with the CD_GETTRACKINFO command, returned by the driver in the `trackinfo.control` field:

```
#define CD_GTI_AUDIOINFO      0x90    /* and with this, then */
#define CD_GTI_2CH_NOEMPHASIS 0x00   /* compare to these values */
#define CD_GTI_2CH_EMPHASIS   0x10    /*... */
#define CD_GTI_4CH_NOEMPHASIS 0x80    /*... */
#define CD_GTI_4CH_EMPHASIS   0x90    /*... */
#define CD_GTI_DATATRACK      0x40
#define CD_GTI_CHECKCOPY      0x20    /* and with this, then */
#define CD_GTI_NOCOPYING      0x00   /* compare to these values */
#define CD_GTI_COPYOKAY       0x20    /*... */
```

These values are OR'ed into the return from `cdstatus`:

```
#define CDISHERE              0x01
#define CDISPLAYING           0x02
#define CDISPAUSED            0x04
```

These values are set by the `cdmediachanged()` function:

```
#define MEDIAWASCHANGED       -1
#define MEDIAMAYCHANGED       0
#define MEDIANOTCHANGED       1
```

Microsoft CD-ROM Extensions data structures

Use the data structures shown here as a template to create your own data structures to use with Microsoft CD-ROM Extensions. Although the Microsoft CD-ROM Extension function calls make no explicit reference to the data structures shown here, they assume that certain buffers are defined in a like manner.

You can find these data structures in the `CDROM.H` file.

struct cdreqheader

```
{
char len;                     /* 13, size of (cdreqheader) */
char unit;                    /* drive number */
char cmd;                     /* command to execute */
int status;                   /* driver return status */
char reserved[8];
};
```

struct cdromdrives

```
{
char unitcode;                /* drive number */
void far *driverheader;       /* address of driver header */
};
```

C Programming the PC's Interrupt Controller and DMA Channels

This appendix describes how to program the IBM PC/AT's interrupt and DMA controllers to work with Pro AudioSpectrum hardware and software. The techniques and descriptions are not intended to be comprehensive. For a comprehensive description of PC hardware devices, refer to the appropriate data sheets or see one of the many popular PC programming books available in computer book stores.

Programming the PC's interrupt controller

The IBM AT class of machines incorporate two Intel 8259 interrupt controllers for handling hardware device interrupts. These devices are linked serially into the host CPU: one interrupt controller feeds the other and the second feeds the CPU. The following illustration shows how the devices are serially linked.

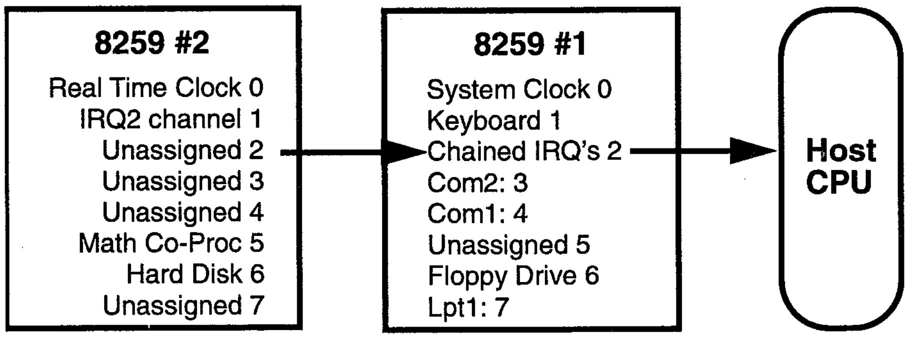


Figure 11 IBM PC/AT Interrupt Controllers

The first IRQ controller I/O addresses are 20h and 21h. The second IRQ controller addresses are A0h and A1h.

I/O addresses 20h and A0h are Interrupt Status Registers. Each bit of these registers correspond to an IRQ channel. In the first controller, IRQ's 0 through 7 correspond to bits 0 through 7. In the second controller, IRQ's 8 through 15 correspond to bits 0 through 7.

When bit 0 through 7 of 20h or A0h is set to one (1), the IRQ corresponding to that bit number is waiting to be serviced. I/O addresses 21h and A1h are Interrupt Mask Registers. When a mask bit is set to 1, the IRQ corresponding to that bit is disabled. Setting the bit to 0 allows the channel to send interrupts to host CPU.

Handling IRQ's on either controller is straight forward. When an interrupt is generated, the hardware responsible for the interrupt must be acknowledged. Secondly, the interrupt controller must be acknowledged.

The following example shows the sequence to to acknowledge a Pro AudioSpectrum PCM IRQ:

```
mov     dx,INTRCTLRST          ; get the Int status reg.
xor     dx,[_MVTranslateCode]  ; remember to do this...
out     dx,al                  ; ACK the H/W PCM int

mov     al,20h                 ; 8259 non-specific ACK
out     20h,al
```

Handling IRQ's on the second interrupt controller requires one additional step. Since the IRQ flows from the hardware to the second 8259, from the second 8259 to the first 8259, and from the first 8259 to the host CPU, that path must be followed when acknowledging the interrupt:

```
mov     dx,INTRCTLRST          ; get the Int status reg.
xor     dx,[_MVTranslateCode]  ; remember to do this...
out     dx,al                  ; ACK the H/W PCM int

mov     al,20h                 ; 8259 non-specific ACK

out     A0h,al                 ; ACK the 2nd 8259
out     20h,al                 ; ACK the 1st 8259
```

Programming the AT DMA Controllers

The IBM AT class of machines use two Intel 8237 DMA controllers for transferring data and refreshing the DRAM memory. Each controller has four separate channels, but the channels are significantly different from one another due to the way each was incorporated into the motherboard design.

The DMA controller is a 16- bit device and can only read from a 64K address space. Since the PC addresses a minimum of 1 megabyte, PC designers had to add hardware to support this size memory model. As a result, programming is more difficult due to restrictions placed on the location and length of the DMA buffer.

A special *Page Register* was created with the original PC and carried forward with the AT. This register receives a value indicating which 64K page of memory the DMA controller will read or write to. The 1 megabyte address space is divided into 16 separate pages.Using a simple trick, the second DMA controller is able to transfer 128K of data from 128K blocks by way of the page register. The following table illustrates the 16 megabyte address space, as divided by the Page Register:

Page register value	**DMA Controller #1**	**DMA Controller #2**
	Linear Address	Linear Address
	Minimum Maximum	Minimum Maximum
0	000000 - 00FFFF	000000 - 01FFFF
1	010000 - 01FFFF	020000 - 03FFFF
2	020000 - 02FFFF	040000 - 05FFFF
3	030000 - 03FFFF	060000 - 07FFFF
4	040000 - 04FFFF	080000 - 09FFFF
5	050000 - 05FFFF	0A0000 - 0BFFFF
6	060000 - 06FFFF	0C0000 - 0DFFFF
7	070000 - 07FFFF	0E0000 - 0FFFFF
	through...	through...
EF	EF0000 - EFFFFF	EF0000 - FFFFFF

Note: The Page Register on AT class machines has been increased to allow access to the full 16 megabytes of address space, so the page register can be programmed with values of 0x00 through 0xFF.

It is important to keep track of which pages of memory your program is using for DMA purposes. Programs must load the starting address and block length into the DMA hardware. If this starting address plus block length exceeds 64K or 128K, the DMA controller will wrap back to offset 0000 within the current page; it will not increment the page register and continue through memory.

The second DMA controller only transfers 16-bit quantities on word boundaries. This is due to the fact that the second DMA controller, and it's page registers, occupy address lines A1 through A23. The second DMA controller never uses the least significant bit of the memory address lines. Therefore, the values programmed into the base address and base count must be shifted right by one (effectively a divide by 2) in order to transfer the correct number of bytes. As a result, the PCM Sample Buffer Timer (register 0x1389) value must also be divided by two.

DMA addresses

To begin a DMA transfer, all of the following registers must be programmed for the target DMA channel:

- Page register
- Base address
- Base count
- Read/write status count
- Write request
- Write single mask
- Write mode
- Clear byte flip/flop

The following tables list register addresses for each of the eight PC/AT IRQ channels. All values are listed in hexadecimal notation:

Channel/Pre-Defined Use	Page Register	Base Address	Base Count	Read/Write Status Command	Write Request	Write Single Mask Register	Write Mode	Clear Byte Flip Flop
Channel 0 - Available	87h	00h	01h	08h	09h	0Ah	0Bh	0Ch
Channel 1 - Available	83h	02h	03h	08h	09h	0Ah	0Bh	0Ch
Channel 2 - Floppy	81h	04h	05h	08h	09h	0Ah	0Bh	0Ch
Channel 3 - Available	82h	06h	07h	08h	09h	0Ah	0Bh	0Ch

Table 17 DMA Controller 1 (8-bit) Register Addresses

Channel/Pre-Defined Use	Page Register	Base Address	Base Count	Read/Write Status Command	Write Request	Write Single Mask Register	Write Mode	Clear Byte Flip/Flop
Channel 4- memory refresh	8Fh	C0h	C2h	D0h	D2h	D4h	D6h	D8h
Channel 5 - Available	8Bh	C4h	C6h	D0h	D2h	D4h	D6h	D8h
Channel 6 - Available	89h	C8h	CAh	D0h	D2h	D4h	D6h	D8h
Channel 7 - Available	8Ah	CCh	CEh	D0h	D2h	D4h	D6h	D8h

Table 18 DMA Controller 2 (16-bit) Register Addresses

DMA mode

The DMA mode determines if the DMA controller will perform a record or playback function. There are two types of transfers to consider for both record and playback:

- Auto-initialize

  Although it is the least used, the DMA controller's auto-initialize mode is the best for audio. It allows unlimited, uninterrupted transfers to occur and forces the DMA controller to auto-initialize itself when the end of buffer has been reached.

- Single shot approach

  With the single shot approach, the DMA controller must be reprogrammed for every new block transfer.

Programming procedure

The DMA controller must be programmed before the Pro AudioSpectrum PCM engine is enabled.

Before enabling the DMA channel, "secure" the channel via the DRQ bit in the Cross Channel register (register 0xF8A). This causes the Pro AudioSpectrum to drive the DMA channel from a floating state to a known good state. If you enable the DMA channel with an unsecured DRQ line, the DMA will perform a rapid-fire data transfer due to the floating DRQ line.

Use these sequential programming steps to setup the DMA for a transfer.

1. **Disable the DMA channel by writing one of these values to the Write Mask register:**

   Channel 0: 04h

   Channel 1: 05h

   Channel 3: 07h

   Channel 5: 05h

   Channel 6: 06h

   Channel 7: 07h

2. **Write the page number to the page register.**

   The page number is the top eight bits of the DMA buffer's 24 bit linear address.

3. **Clear the byte pointer flip-flop register.**

   You can write any value.

4. **Write the base address and the base count registers.**

   The base address is the lower 16 bits of the DMA buffer's 24 -bit address. Be sure to write the low byte of each value first, then the high byte.

5. **Write the DMA transfer mode to the Write Mode register.**

   Write a value corresponding to the desired transfer method:

   Auto-initialize continuous playback: 59h

   One shot non-continuous playback: 49h

   Auto-initialize continuous record: 55h

   One shot non-continuous record: 45h

6. **Enable the DMA channel by writing one of these values to the Write Single Mask register.**

   Channel 0: 00H

   Channel 1: 01h

   Channel 3: 03h

   Channel 5: 00h

   Channel 6: 01h

   Channel 7: 03h

D Relocating Pro AudioSpectrum I/O Addresses

The new Pro AudioSpectrum models (Pro AudioSpectrum Plus, Pro AudioSpectrum 16, and CDPC) support relocatable I/O addresses. The purpose of this feature is to allow multiple cards to be installed in a single PC. Up to four Pro AudioSpectrum boards can be resident within the computer, all at separate I/O addresses.

The *native* Pro AudioSpectrum I/O addresses use four 16-bit I/O addresses. These addresses are sequential, and normally start at 0x388 and run through 0x38A. The exceptions being the implementation of other standard devices (the joystick, Sound Blaster emulation, and MPU-401), which have fixed, predefined addresses.

The following diagram shows the Pro AudioSpectrum 16-bit addressing scheme:

```
|<-----------------------16 BIT ADDRESS ----------------------->|
7654          3210          7654          3210          Description
                                          |<--- 388 --->|  OPL3 FM
                            |<----------- 1788 ------------>|  MIDI
              |< -------------------- 5388 -------------------->|  SCSI
|<---------------------------------- F388 --------------------------------->|  Configuration
```

Table 19 Relocatable Hardware Addresses

The original Pro AudioSpectrum used 388h through 38Bh for it's base address. This address range was originally chosen by Adlib even though it is within the reserved area of the SDLC or 2nd Bisynchronous communications hardware. Fortunately, almost no hardware has been found to have conflicts with this default address range.

Media Vision's I/O relocation capability is software selectable to any one of 256 locations, but only four are recommended. To avoid creating I/O conflicts with other devices, Media Vision only suggests using the base address of 388h, 384h, 38Ch, or 0x288h.

You must reset the hardware before the board accept a new base address.

To relocate I/O addresses, write two 8-bit values to the Master Address Pointer register at address 9A01h.

1. **Write the board ID value address 9A01h.**

   Valid settings are BCh through BFh.

2. **Write the base address value shifted right by two positions.**

The following assembler code sets up the base address at 388h:

```
mov  dx,09a01h;                Master Address Pointer

mov  al,0BCh;                  1st Board ID

out  dx,al;                    Send the ID

mov  ax,0388h;                 ax = 1110001000B

shr  ax,2;                     ax =   11100010B

out  dx,al;                    Send the Address
```

This procedure can be followed for the next three boards. To support the I/O relocation, the Pro AudioSpectrum Developer's Toolkit provides a library call to locate the existing hardware.

The routine `MVGetHWVersion` returns a variety of information including a *translate code*. This translate code is XORed into the original I/O address to derive the physical I/O address.

The following assembler code segment programs the Cross Channel register:

```
include common.inc; PAS equates

mov  dx,CROSSCHANNEL;          An EQUated address

xor  dx,[_MVTranslateCode];    Adjust the address

in   al,dx;                    get all the bits

and  al,fCCpcmbits;            save the PCM bits

or   al,bCCl2l+bCCr2r;         set l-2-l, r-2-r

out  dx,al;                    send it back out...
```

E Pro AudioSpectrum Utility Programs

The Pro AudioSpectrum Developer's Toolkit includes a number of useful utility programs that you can use to explore the Pro AudioSpectrum. Source is available on the Developer's Toolkit disk for many of these utilities. Read the file `SOURCE.LST` in the root directory for a list of the source files.

Several of these utilities require that you have the `MVSOUND.SYS` device driver loaded by `CONFIG.SYS`. Most of these utilities display the command line parameters if you type the program name without switch settings. For example, to get information about the utility `PLAYFILE.EXE`, type `PLAYFILE` at the MS-DOS prompt.

The following conventions apply to all the utilities below, except when explicitly noted otherwise:

- Command flags can be typed in either upper- or lowercase, and a dash (-) or slash (/) preceding the flag is optional.

  PLAYFILE tigers.wav D3 S120

  PLAYFILE tigers.wav /s120 -d3

- Flags need not be entered in a prescribed order. The two command lines below are equivalent:

  PLAYFILE tigers.wav D3 S120

  PLAYFILE tigers.wav S120 D3

PLAYFILE

The `PLAYFILE.EXE` utility plays back PCM digitized audio data stored in a file. It converts the digital sound data into audio that can be mixed and played through the speakers or headphone. A companion program, `RECFILE.EXE`, records digital sound data. `PLAYFILE` can playback audio files at up to 44,100 samples per second (double for stereo) on most machines.

Syntax

```
PLAYFILE <sound filename> [Dx] [Ix] [FCC] [S] [Sxxx]
[Rxxxxx] [16]
```

where:

sound filename	Filename of file to be played. You must specify the file extension.
Dx	Optional. Specifies the DMA channel and overrides the default DMA setting. Enter a DMA channel value of 1, 2, 3, 5, 6, or 7.
IX	Optional. Specifies the IRQ channel and overrides the default IRQ setting. Enter an IRQ channel value of 3, 5, 6, 7, 10, 11, 12, or 15.
Fcc	Optional. Sets the cut-off frequency for the low-pass filter. Enter a filter value of 1 through 6. 1 is the recommended setting for male speech (cut-off at about 6 kHz). 6 is recommended for high-fidelity music (cut-off at about 22 kHz).
S	Optional. Forces a monaural file to be played back as stereo sound. Half of the sound samples (every other one) become left -source sound data; the other half become the right source.
Sxxx	Optional. Sets the speed adjustment. Enter a speed adjustment value from 0 (silence) to 200 (double speed). A value of 100 sets the speed adjustment level to "no change."
Rxxxxx	Optional for .WAV and .VOC files; required for other file types. Sets the sampling rate and overrides the sampling rate in the file headers of .WAV and .VOC files. Enter a value from 4000 to 44100.
16	Optional. Specifies that the file be played as 16-bit PCM.

Note: `PLAYFILE` will play back *any* file you give it. If it does not recognize the file type, it assumes that the file contains 8-bit unpacked PCM audio data.

Source code

Provided.

Example

```
PLAYFILE TEST123.WAV S120
```

This plays sound file `TEST123` at 120% of the speed it was recorded (20% speed increase).

RECFILE

`RECFILE.EXE` records audio input and converts it into PCM digitized audio that is saved to disk in either the `.VOC` or `.WAV` file format. Before using `RECFILE`, use the `PAS` utility to setup the mixer inputs and volume control.

`RECFILE` records at up to 44,100 samples per second (double for stereo) on most machines.

To terminate recording, press the [ESC] key.

Note: Since the PCM circuitry on the Pro AudioSpectrum can be used for either playing back or recording digitized audio, you must not select digitized audio as an input when recording with `RECFILE`.

Syntax

```
RECFILE <sound filename> [Rxxxxx] [Dx] [Ix] [Fcc] [S]
```

where:

sound filename	Required. Filename of file to capture audio data. If you specify the .VOC file extension, the sound data is saved in Sound Blaster .VOC file format. If you do not specify .VOC, the sound data is saved in Microsoft WAVE (.WAV) file format.
Rxxxxx	Required. Sets sampling rate. Enter a value from 4000 and 44100. Note that if you select 44100 and are recording in stereo, you are actually saving 88200 samples per second.
Dx	Optional. Specifies the DMA channel and overrides the default DMA setting. Enter a DMA channel value of 1, 2,3, 5, 6, or 7.
IX	Optional. Specifies the IRQ channel and overrides the default IRQ setting. Enter an IRQ channel value of 3, 5, 6, 7, 10, 11, 12, or 15.
Fcc	Optional. Sets the cut-off frequency for the low-pass filter. Enter a filter value of 1 through 6. 1 is the recommended setting for male speech (cut-off at about 6 kHz). 6 is recommended for high-fidelity music (cut-off at about 22 kHz).
S	Optional. Forces stereo recording. If you record a stereo source in monaural, only the left- channel sound is captured.
16	Optional. Specifies that the file be recorded as 16-bit PCM.

Note: Use the lowest possible sampling rate that's feasible to avoid filling up the hard disk. For example, to record a male voice a sampling rate of 6000 is adequate. Digitized audio sound files can grow very quickly. If you record stereo sound at 44100 sampling rate you record 88,200 samples per second. Since each sample is 1 byte, you save 88,200 bytes per second. One minute of recording consumes just over 1.77 megabytes!

Source code

Provided.

Example

```
RECFILE LEFT.WAV r11025 S 16
```

This example creates a stereo digitized audio file, in .WAV format, that has been sampled in 16-bit PCM at 11 kHz (recording 22K samples per second).

BLOCKOUT

The `BLOCKOUT.EXE` utility is similar to the `PLAYFILE.EXE` utility, but is much simpler. Like `PLAYFILE`, it plays back PCM digitized audio data stored in a file. It is designed to playback a file created by `BLOCKIN.EXE`, has a fixed playback rate of 22,050 samples per second, and assumes monaural sound.

Syntax

```
BLOCKOUT <sound filename>
```

where:

sound filename	Required. Filename of file to be played. You must specify the file extension. This file should be created by BLOCKIN.EXE.

Note: `BLOCKOUT` plays back *any* file you give it. If it does not recognize the file type, it assumes that the file contains 8-bit unpacked PCM audio data.

Source code
Provided.

Example

```
BLOCKOUT MYVOICE.TST
```

BLOCKIN

The `BLOCKIN.EXE` utility is similar in function to the `RECFILE` utility but is much simpler. It records monaural audio input (left channel only) to disk at a fixed rate of 22050 samples per second. The sound file can be played by `BLOCKOUT`.

Terminate recording by pressing the [ESC] key.

Syntax

```
BLOCKIN <sound filename>
```

where:

sound filename	Required. Filename of file to capture audio data. If you specify the .VOC file extension, the sound data is saved in Sound Blaster .VOC file format. If you do not specify .VOC, the sound data is saved in Microsoft WAVE (.WAV) file format.

Source code

Provided.

Example

```
BLOCKIN MYVOICE.TST
```

This creates a stereo digitized audio file, in unpacked 8-bit binary format (no header), that has been sampled at 22K samples per second.

MERGE

`MERGE` combines two `.WAV` format audio files into a single file. If the two sources files are monaural, the left source becomes left channel information and the right source becomes right channel information in the output file. If the two source files are stereo, `MERGE` combines the first file's left channel with the second file's right channel.

Syntax

```
MERGE <left source filename> <right source filename>
<output>
```

where:

left source	Name of .WAV file that provides left channel audio information.
right source	Name of .WAV file that provides right channel audio information
output	Name of output file that is created by combining the left and right source files.

Error messages

MERGE generates the following two different error messages.

Error Message	Description
One of the input files is specified as the output file name.	You cannot overwrite an input file. Use an output filename that is different from that of the input file.
The left/right channel is already a stereo file. Is this okay?	MERGE has recognized that one (or both) of the input sources are stereo channels. Confirm that you want to merge stereo sources by pressing 'Y'. Press 'N' to abandon the merge.

Source code

Not Provided.

Example

```
MERGE speech.wav music.wav Waf.wav
```

WHATIS

WHATIS lets you examine the contents of a .VOC file and shows the contents of the header for each data block.

Syntax

```
WHATIS <file.voc>
```

where:

file.voc	Name of the file to examine.

Source code

Not Provided.

Example

```
WHATIS GUPPY.VOC
```

WAVEIT

WAVEIT converts .VOC, .SOU, and other 8-bit PCM digitized audio files into Microsoft Wave (.WAV) format. If WAVEIT recognizes the input file type (by examining the file extension), it reads its header information and copies relevant parameters to the header of the .WAV output file that it creates.

Syntax

```
WAVEIT <input file> <output file> [Rxxxxx] [S]
```

where

input file	Name of the .VOC, .SOU, or other 8-bit PCM file that you want to convert to .WAV format. You must specify the file extension.
output file	Name of the .WAV file generated by the WAVEIT utility.
Rxxxxx	Optional. Sets the sample rate to assign the output file. If the file type is .VOC or .SOU, WAVEIT reads the sample rate from the input file header. To convert other file types you must supply the sample rate. You can also override the .VOC or .SOU sample rate using this parameter. The rate can vary from 4000 to 44100. If you specify the conversion to be done in stereo (using the S flag), the actual sampling rate is automatically doubled.
S	Optional. Specifies that the PCM data in the .VOC or .SOU file to be interpreted as stereo data. If this flag is left out, the <input file> is interpreted as monaural data. When S is specified, WAVEIT ignores the .VOC or .SOU file header information indicating that stereo or monaural data is contained in the file.

Source code

Not provided.

Example

```
WAVEIT thunder.voc thunder.wav
```

PAS

PAS lets you send commands to the mixer, equalizer, and volume control from the MS-DOS command line or from a batch file. PAS also provides an interactive mixer control panel. With this very easy-to-use interface you can see the current mixer settings; toggle mixer input sources on and off and set their individual levels; and, control the total volume control, bass, and treble.

While using the on-screen mixer you can press [F1] for help.

You can find a more complete description of the PAS utility in Pro AudioSpectrum User's Guide.

Syntax

```
PAS [*]
```

where

*	Optional. Causes the interactive mixer control panel to appear.

Source code

The user interface source code, `dialog.c`, is provided in `\pas\subs\mixers`.

OPL3

OPL3 is designed for hardware 'hackers' who want to directly program the FM synthesizer chip. The OPL3 utility displays a map of the FM synthesizer registers. Using a mouse you can point to and click on register bits. When you toggle KON (key on) you hear the sound corresponding to the register values you selected. This utility makes it easy to understand the FM synthesizer by giving you immediate, audible feedback as you program the FM synthesizer operators.

For detailed information on standard FM synthesizer registers, see Chapter 11, "Standard FM Synthesizer Register Functions." For detailed information on enhanced (OPL3) FM synthesizer registers, see Chapter 12, "Enhanced FM Synthesizer Register Functions."

Since there are too many registers to display all on the screen at one time, this utility only shows the relevant set of registers to an operator. To program an individual operator, you program each register in the set that controls that operator. When you select a register to program, OPL3 automatically displays all the related registers, making it easy to program an operator.

For example, if you select register 20 (to program the first operator), registers 40, 60, and 80 are also displayed, because they all relate to operator one. Additionally, you will see registers A0, B0, and C0 because they contain parameters for the first voice, which is comprised of operators 1 and 4. Registers 01h through 04h, register 08h, and register BDh apply to all operators so they are always displayed.

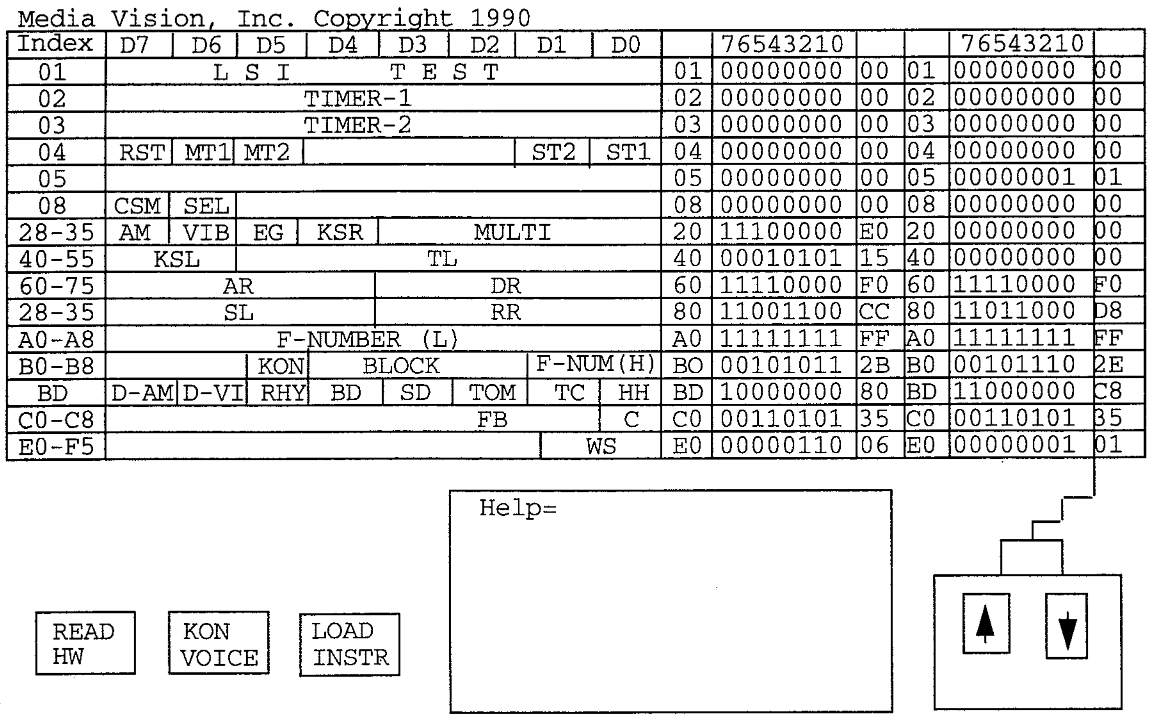
Note: This utility requires a mouse and VGA color adapter. It is loaded only if it detects the correct equipment in the PC.

OPL3 Screen

The OPL3 utility screen contains the following components:

Component	**Description**
Register window	The register window consists of five vertical sections. Each is described below in the order they appear from left to right.
Address (Index)	The Address section is the left-most column titled "Index". This column displays the address offset, from the base address of the Yamaha chip, for a register or register group. For example, the first row of the register map has an Address entry of 01. This row explains the register at offset +01. The middle section of this utility begins with an Address entry of 20h-35h. This row explains the 18 registers in the register group at offsets +20h through +35h. Each register in this group controls the amplitude (AM), vibrato (VIB) and similar characteristics for its corresponding operator.
Register Map (D7-D0)	The register map, which contains eight columns titled D7 through D0, provides short descriptive labels explaining the purpose of each register bit. The meaning of each bit, or group of bits, varies from register to register.

The following figure shows an example OPL3 screen display:



Media Vision, Inc. Copyright 1990

Index	D7	D6	D5	D4	D3	D2	D1	D0		76543210			76543210	
01	L S I T E S T								01	00000000	00	01	00000000	00
02	TIMER-1								02	00000000	00	02	00000000	00
03	TIMER-2								03	00000000	00	03	00000000	00
04	RST	MT1	MT2				ST2	ST1	04	00000000	00	04	00000000	00
05									05	00000000	00	05	00000001	01
08	CSM	SEL							08	00000000	00	08	00000000	00
28-35	AM	VIB	EG	KSR	MULTI				20	11100000	E0	20	00000000	00
40-55	KSL		TL						40	00010101	15	40	00000000	00
60-75	AR				DR				60	11110000	F0	60	11110000	F0
28-35	SL				RR				80	11001100	CC	80	11011000	D8
A0-A8	F-NUMBER (L)								A0	11111111	FF	A0	11111111	FF
B0-B8			KON	BLOCK			F-NUM(H)		BO	00101011	2B	B0	00101110	2E
BD	D-AM	D-VI	RHY	BD	SD	TOM	TC	HH	BD	10000000	80	BD	11000000	C8
C0-C8	FB							C	C0	00110101	35	C0	00110101	35
E0-F5							WS		E0	00000110	06	E0	00000001	01

Help=

Figure 12 OLP3 Sample Display

OPL3 gives you further information on each bit field when you position the cross-hair on the corresponding bit in the Bit Value column.

Register Select (un-labeled column)	The narrow column immediately to the right of the register tells you which register is currently selected among those in a register group. Note how the first five rows in this column display the same value as the Index column since there is only one register in each group. Now examine the row that has an Index entry of 20-35. For this row, the Selected Register column tells you that register 20 is currently selected. Any changes you make to bit values, by clicking on the bits in the columns to the right, will revise register 20 only. By clicking on this column you can switch to another register in this group. For example, by positioning the cross-hair on the 20 value and clicking once with the left mouse button you can cause register 21 to become the currently selected register in this group. Conversely, clicking with the right button decrements the register select. Note that when you switch to register 21 you also simultaneously switch to registers 41, 61, 81, and E1 in the rows below. All five registers are related to the same operator.
Bit Value	This column, titled "76543210", displays the individual bits of the currently selected register. You can change the bit values by positioning the cross-hair on register (row) and bit you want to change, and then click on the bit. You will toggle the bit from 0 to 1 or vice-versa. When you position on a bit its meaning is explained in the Helps window. Note that the column to the right shows the hex equivalent for the 8 bits shown in this column.
Hex Value (un-labeled column)	You can also change the register value by clicking on the hex value displayed in the un-labeled, right-most column.
Helps window	Text appears here that explains the register bits. Use the mouse to position the cross-hair on the Bit Value column you want to have explained.
Mouse picture	This picture explains what the mouse buttons do. When you click on a bit you toggle it on or off. When you click on a hex value you increment (left mouse button) or decrement (right mouse button) by one.
KON Voice box	This keys on (turns on) ALL voices simultaneously. Click once to turn on the voices, click twice to turn off. You can sound a single voice by double-clicking on that voices's KON bit (in B0-B8).

Configuration File

OPL3 automatically saves the register settings when you quit. They are stored in the file OPL3.STT. When you next runOPL3, these settings are restored. To quickly clear all registers, delete the OPL3.STT configuration file before running this utility.

Source

Not provided.

Example

The following values demonstrate how to set up a single operator for making a simple sound. First set all bits to zero, if they are not already. Next, prepare configure operator 0 with the settings given below. They toggle the Key-On bit on and off to generate notes.

Configure the registers at offset 20h, 40h, 60h, and 80h that control this operator:

register 20	(Multi)	01h
register 40	(KSL/TL)	00h
register 60	(AR/DR)	F0h
register 80	(SL/RR)	FFh

Now configure the registers that control the first voice (operators 1 and 4, but operator 4 will not be enabled). The F-Number (low bits), F-Number (high bits), and Block parameters in registers A0 and B0 control the operator frequency. Bit D0 is very important, since this is the "connect" that enables sound to be heard from this voice:

register A0	(F-Number (L)	40h
register B0	(Block/F-Number(H))	12h
register C0	(Connect)	01h

To hear the sound click twice on the KON (key on) bit for this voice, D5 in register B0. Now experiment by changing the frequency by clicking on the MUL bits in 20h. Try changing the timbre by adding tremolo (AM) or vibrato (VIB).

Note: The settings above describe the fourth octave A that has a frequency of 440 hertz. For a complete list of F-Numbers, see Appendix A, "FM Hardware Register Charts and Tables."

REPORT

REPORT notifies your of all INT 2F calls made to MVPROAS. It echoes messages to a monochrome adapter (only) before and after an INT 2F call.

REPORT stays memory resident until you reboot.

Syntax

```
REPORT
```

Source code

Provided upon request.

F INT 2F Function Calls

This appendix documents the binary INT 2F interface to the Pro AudioSpectrum. The INT 2F multiplex interrupt is used by applications to talk directly to the control devices on the Pro AudioSpectrum like the mixers and volume control. INT 2F calls are made directly to the MVSOUND.SYS device driver rather than through MS-DOS. This interface is non-reentrant and uses a semaphore to control entry. Return values indicate whether calls were successful or whether a collision occurred with another process.

An especially important call is the Function #3 - Get Pointer to Function Table. It points to a table of function pointers that are used access to most of the devices on the Pro AudioSpectrum board. The linkable mixer sub-routines use this function for mixer control.

Common function call method

All functions share a common method for function call entry and exit. They are called through a single entry point using appropriate function numbers in register AX and function number variables in registers BX, CX, and DX.

Return values are placed in registers AX, BX, CX, and DX. The normal return values are listed below:

Register	Normal return value
AX	'MV' if the call was successful not 'MV' if the call failed
BX	Varies by function call
CX	Varies by function call
DX	Varies by function call

A function call can fail if:

- The driver is not loaded.
- The call collided with another program that just called this driver. A semaphore scheme is employed by the driver to ensure that the driver will not respond to a second call while still handling the first.
- Another process uses 'BC' as its INT 2F interface identifier.

The example below shows the common calling method. Prototype function call entry/exit code can also be found on the developer's kit diskette in the MIXERS.H file.

```
ASM    include binary.inc
AH     0xBC            ; Media Vision INT2F identifier
AX     function        ; Media Vision function number (0xBChh)
BX                     (call dependent)
CX                     (call dependent)
DX                     (call dependent)
```

Check For Driver Function 0

Checks to see if the MVSOUND.SYS device driver is loaded. If the device driver is loaded, the function returns a unique set of values in variables *AX* and *BX*.

Input parameters

AX	0xBC00
BX	0x3F3F

Return values

If the driver is loaded:

AX	0xBC00	(unchanged)
BX	0x6D00	(M)
CX	0x0076	(V)
DX	0x2020	

XORING CX and DX into BX returns the ASCII characters "MV".

If the driver is not loaded, values in the registers are unknown.

Related topics

INT 2F Function #1 (Get Version #), `MVInitMixerCode`

Get Version Function 1

Returns version number of the `MVSOUND.SYS` device driver and the version of the Pro AudioSpectrum board.

Input parameters

AX	0xBC01

Return values

If the function call was successful:

AX	'MV'
BX	library major version id
CX	library minor version id
DX	hardware version

If the function call failed:

AX	does not contain 'MV'
BX	undefined
CX	undefined
DX	undefined

Related topics

INT 2F Function #0 (Check for Driver)

Get Pointer to State Table Function 2

Returns a far pointer to the state table containing hardware state information. The state table is necessary because the Pro AudioSpectrum contains write-only registers.

The structure declaration for the state table, taken from file STATE.H follows the related topics section. The developer's kit diskette contains STATE.INC for MASM programmers.

Note: The comment for each structure element gives the hex offset of the given board address.

Input parameters

AX	0xBC02

Return values

If the function call was successful:

AX	'MV"
BX	offset to the table
CX	length of the table
DX	segment to the table

If the function call failed:

AX	does not contain 'MV'
BX	undefined
CX	undefined
DX	undefined

Related topics

None.

State table

```
struct MVState
{
unsign char _sysspkrtmr;      /* 42 System Speaker Timer Address */
unsign char _systmrctlr;      /* 43 System Timer Control */
unsign char _sysspkrreg;      /* 61 System Speaker Register */
unsign char _joystick;        /* 201 Joystick Register */
unsign char _lfmaddr;         /* 388 Left FM Synth Address */
unsign char _lfmdata          /* 389 Left FM Synth Data */
unsign char _rfmaddr          /* 38A Right FM Synth Address */
unsign char _rfmdata          /* 38B Right FM Synth Data */
unsign char _RESRVD1[4];      /* reserved */
unsign char _audiomixr;       /* B88 Audio Mixer Control */
unsign char _intrctlrst;      /* B89 Interrupt Status */
unsign char _audiofilt;       /* B8A Audio Filter Control */
unsign char _intrctlr;        /* B8B Interrupt Control */
unsign char _pcmdata;         /* F88 PCM Data I/O Register */
unsign char _RESRVD2;         /* reserved */
unsign char _crosschannel;    /* F8A Cross Channel */
unsign char _RESRVD3;         /* reserved */
unsign int _samplerate;       /* 1388 Sample Rate Timer */
unsign int _samplecnt;        /* 1389 Sample Count Register */
unsign int _spkrtmr;          /* 138A Shadow Speaker Timer Count */
unsign char _tmrctlr;         /* 138B Local Timer Control */
unsign char _RESRVD4[8];      /* reserved */
};
```

Note: The library code keeps entries from B8A through 1389 inclusive up-to-date whenever you make library calls. If you write directly to the hardware you are responsible for updating the corresponding entries in this table.

Get Pointer to Function Table
Function 3

Returns a far pointer to the function table containing addresses of driver functions that set or get settings for Pro AudioSpectrum devices. Devices controlled through this interface include the mixer, total volume control, filter, and the cross channel device.

The software routines called indirectly through Function 3 pointers are identical to the statically linked routines.

Note: When `MVInitMixerCode()` is called, Function 3 is called to dynamically link to `MVSOUND.SYS's` ten internal functions. This level of indirection is necessary to insulate your program from future hardware changes.

Input parameters

AX	0xBC03

Return values

If the function call was successful:

AX	'MV'
BX	offset address to table
CX	number of table entries
DX	segment address to table

If the function call failed:

AX	does not contain 'MV"
BX	undefined
CX	undefined
DX	undefined

The table below lists the operations that are activated using Function 3 pointers.

Operation (pointer)	**Entry conditions**	**Return value**	**Description**

Set mixer (0)	BX = level
	CX = mixer select
	DX = channel
		None	Sets the signal level for specified channel

Set volume control (1)	BX = level
	CX = device select
		None	Sets total volume control device levels

Set filter (2)	BX = limit	None	Sets PCM low-pass filter frequency
Set cross channel (3)	BX = channel mask bits	None	Sets cross channel device switch settings

Get mixer (4)	CX = mixer select
	DX = channel
		BX = level	Returns signal level setting for the specified channel

Get volume control (5)	CX = device select	BX = level	Returns current setting for the total volume control
Get filter (6)	None	BX = setting	Returns setting of PCM low-pass filter
Get cross channel (7)	None	BX = cross channel bit map	Returns settings of cross channel device
Real sound switch (8)	CX = mode	BX = state	Sets or reads real sound setting

FM split (9)	BX = state
	CX = state
		BX = state	Sets or reads FM synthesizer split setting

Table 20 Operations Activated By Function 3

Related topics

For additional information on how to use the Function 3 calls described below, refer to the comparable cMV* routines in the Linkable Mixer Calls section.

Get DMA/IRQ/INT Function 4

Returns the DMA, IRQ, and INT numbers that were selected when the `MVSOUND.SYS` driver was loaded. These numbers are specified in the `Device` command line in `CONFIG.SYS`.

Note: The software library currently returns only IBM XT DMA/IRQ values (DMA 1-3 and IRQ 3-7).

Input parameters

AX	0xBC04

Return values

If the function call was successful:

AX	'MV'
BX	DMA channel number (1-3, 5-7)
CX	IRQ channel number (3-15)
DX	INT number (unused)

If the call failed:

AX	does not contain 'MV'
BX	undefined
CX	undefined
DX	undefined

Related topics

None.

Send Command Structure Function 5

Reserved.

Get Driver Message Function 6

Returns a pointer to the text message generated by the driver.

The driver generates messages (ASCII strings containing 79 characters or less plus a null terminator) in response to each command it processes. The typical text message is 'okay' after successful processing of a command.

Note: You must copy the test message immediately after a command is processed. The next call to the driver, which could be generated by another program running in the background or the user issuing driver commands from the keyboard, will overwrite the message response to your command.

Input parameters

AX	0xBC06

Return values

If the function was successful:

AX	'MV'
BX	offset of pointer
CX	undefined
DX	segment of pointer

If the call failed:

AX	does not contain 'MV'
BX	undefined
CX	undefined
DX	undefined

Related topics

None.

Set Hotkey Scan Codes Function 10

Sets new scan codes for the keyboard hot key combinations that control the master volume control. The default settings are:

[Ctrl] + [Alt] + [U]	increase volume
[Ctrl] + [Alt] + [D]	decrease volume
[Ctrl] + [Alt] + [M]	toggle the mute switch

Key scan codes are those of the standard IBM XT or AT keyboard (83, 84, 101, and 102 key keyboards). Special AT keys, such as PgDn, generate extended (2 byte) scan codes where the first byte is always E0. Specify these keys by entering only the second byte of the scan code.

Note: The new hot key combination will stay in effect until modified, as long as the PAS driver is loaded, or until the PC is rebooted.

Input parameters

AX	0xBC0A
BX	hot key selection
CX	new hot key scan code combo:
	CH: key scan code
	CL: shift state scan code

Valid BX and CX/CL scan codes:

	0x00	0x01	0x02	0x04	0x08
Hot key selections (BX)	Volume down	Volume up	Toggle mute on/ off	N/A	N/A
Shift state scan codes (CX/CL)	N/A	Shift right	Shift left	Control	Alt

Table 21 Hot Key Scan Code Values

Return values

If the function was successful:

AX	'MV''
BX	undefined
CX	previous hot key scan code combo
	CH: shift state scan code
	CL: key scan code
DX	undefined

If the call failed:

AX	does not contain 'MV'
BX	undefined
CX	undefined
DX	undefined

Related topics

None.

Get Path to Driver Function 11

Returns a pointer to an ASCII string (`\PROAUDIO`) that is the MS-DOS path to the `MVSOUND.SYS` driver.

MS-DOS locates paths to drivers using `CONFIG.SYS`. This routine provides a way for your application to locate the driver and other associated files in the directory called `\PROAUDIO`.

Input parameters

AX	0xBC0B

Return values

If the function was successful:

AX	'MV'
BX	offset of pointer
CX	undefined
DX	segment of pointer

If the call failed:

AX	does not contain 'MV'
BX	undefined
CX	undefined
DX	undefined

Related topics

None.

Index