Developer Kit for Sound Blaster Series

Second Edition

Programmer's Guide

- Setting Up
- What's New
- Programmer's Guide
- Migration Guide

License Agreement/Limitation And Disclaimer Of Warranties

PLEASE NOTE : BY DOWNLOADING AND/OR USING THE SOFTWARE AND/OR MANUAL ACCOMPANYING THIS LICENSE AGREEMENT, YOU ARE HEREBY AGREEING TO THE FOLLOWING TERMS AND CONDITIONS:

The software and related written materials, including any instructions for use, are provided on an "AS IS" basis, without warranty of any kind, express or implied. This disclaimer of warranty expressly includes, but is not limited to, any implied warranties of merchantability and/or of fitness for a particular purpose. No oral or written information given by Creative Technology Ltd., its suppliers, distributors, dealers, employees, or agents, shall create or otherwise enlarge the scope of any warranty hereunder. Licensee assumes the entire risk as to the quality and the performance of such software and licensee application. Should the software, and/or Licensee application prove defective, you, as licensee (and not Creative Technology Ltd., its suppliers, distributors, dealers or agents), assume the entire cost of all necessary correction, servicing, or repair.

RESTRICTIONS ON USE

Creative Technology Ltd. retains title and ownership of the manual and software as well as ownership of the copyright in any subsequent copies of the manual and software, irrespective of the form of media on or in which the manual and software are recorded or fixed. By downloading and/or using this manual and software, Licensee agrees to be bound to the terms of this agreement and further agrees that :

(1) CREATIVE'S BBS/FTP/COMPUSERVE ARE THE ONLY ONLINE SITES WHERE USERS MAY DOWNLOAD ELECTRONIC FILES CONTAINING THE MANUAL AND/OR SOFTWARE,

(2) LICENSEE SHALL USE THE MANUAL AND/OR SOFTWARE ONLY FOR THE PURPOSE OF DEVELOPING LICENSEE APPLICATIONS COMPATIBLE WITH CREATIVE'S SOUND BLASTER SERIES OF PRODUCTS, UNLESS OTHERWISE AGREED TO BY FURTHER WRITTEN AGREEMENT FROM CREATIVE TECHNOLOGY LTD.; AND,

(3) LICENSEE SHALL NOT DISTRIBUTE OR COPY THE MANUAL FOR ANY REASON OR BY ANY MEANS (INCLUDING IN ELECTRONIC FORM) OR DISTRIBUTE, COPY, MODIFY, ADAPT, REVERSE ENGINEER, TRANSLATE OR PREPARE ANY DERIVATIVE WORK BASED ON THE MANUAL OR SOFTWARE OR ANY ELEMENT THEREOF OTHER THAN FOR THE ABOVE SAID PURPOSE, WITHOUT THE EXPRESS WRITTEN CONSENT OF CREATIVE TECHNOLOGY LTD.. CREATIVE TECHNOLOGY LTD. RESERVES ALL RIGHTS NOT EXPRESSLY GRANTED TO LICENSEE IN THIS LICENSE AGREEMENT.

LIMITATION OF LIABILITY

In no event will Creative Technology Ltd., or anyone else involved in the creation, production, and/or delivery of this software product be liable to licensee or any other person or entity for any direct or other damages, including, without limitation, any interruption of services, lost profits, lost savings, loss of data, or any other consequential, incidental, special, or punitive damages, arising out of the purchase, use, inability to use, or operation of the software, and/or licensee application, even if Creative Technology Ltd. or any authorised Creative Technology Ltd. dealer has been advised of the possibility of such damages. Licensee accepts said disclaimer as the basis upon which the software is offered at the current price and acknowledges that the price of the software would be higher in lieu of said disclaimer. Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitations and exclusions may not apply to you.

Information in this document is subject to change without notice. Creative Technology Ltd. shall have no obligation to update or otherwise correct any errors in the manual and software even if Creative Technology Ltd. is aware of such errors and Creative Technology Ltd. shall be under no obligation to provide to Licensee any updates, corrections or bug-fixes which Creative Technology Ltd. may elect to prepare.

Creative Technology Ltd. does not warrant that the functions contained in the manual and software will be uninterrupted or error free and Licensee is encouraged to test the software for Licensee's intended use prior to placing any reliance thereon.

Copyright 1993-1996 by Creative Technology Ltd. All rights reserved.

Sound Blaster, Sound Blaster Pro, Sound Blaster 16, and Wave Blaster are trademarks of Creative Technology Ltd.

IBM is a registered trademark of International Business Machines Corporation.

MS-DOS is a registered trademark and Windows is a trademark of Microsoft Corporation.
All other products are trademarks or registered trademarks of their respective owners.

Contents

Chapter 4 High-Level Digitized Sound Drivers

Chapter 5 High-Level Auxiliary Driver

Chapter 6 Creative Multimedia System Driver

Chapter 7 MIDI Driver

Chapter 8 CD-ROM Audio Interface

Appendix A Migration Guide

Appendix B Relevant Information

Glossary

Index

Introduction

Welcome to the Developer Kit for Sound Blaster Series!

This Developer Kit (SBK) is produced for Third Party DOS Developers and other users who intend to develop software programs for Creative's Sound Blaster cards (SBC):

- Sound Blaster™ Version 1.5 or earlier (SB1.5)
- Sound Blaster™ for Micro Channel Version (SBMCV)
- Sound Blaster™ Version 2.0 (SB2.0)
- Sound Blaster™ 2.0 CD Interface (SB2CD)
- Sound Blaster™ Pro (SBPRO)
- Sound Blaster™ Pro Micro Channel Version (SBPRO MCV)
- Sound Blaster™ 16 (SB16)
- Sound Blaster™ 16 Advanced Signal Processing™
- Sound Blaster™ Advanced WavEffects™ 32 (SBAWE32)
- Sound Blaster™ 32 (SB32)

This manual provides high quality technical documentation and programming information on the Creative specific Sound Blaster hardware. Easy-to-use functions and a complete interface that support Digitized sound, FM synthesized music, Mixer control, MIDI interface and CD-ROM Drive Audio interface are included. We at Creative Labs and Creative Technology have spent much effort in creating the drivers and libraries to save your development time.

We have taken great care to meet the requirements of the various types of developers and to reduce the possibilities of clashes with other TSRs, since all DOS programmers face these same problems.

We have developed the **loadable drivers** to make supporting the various SBC hardware easier. We have kept these drivers in separate files. Each Sound Blaster card carries its own set of drivers. The same Application Programming Interface (API) is maintained across the Sound Blaster cards; thus allowing your application to run on the whole series of Sound Blaster cards.

We include the support of the two major digitized sound file formats, and the raw PCM data file. The two file formats supported are **Creative Voice File (.VOC)** and

Multimedia Waveform File (.WAV). The .WAV file format is commonly used in the Multimedia Windows environment.

We provide three methods for digitized sound I/O: **Conventional Memory Method**, **Extended Memory Method** and **Disk Double-Buffering Method**. All these methods impose no software restrictions to the digitized sound I/O; you can record the digitized sound until your memory or disk space is full!

Three 16-bit compression methods are supported with the aid of Creative Advanced Signal Processor on Sound Blaster 16 Advanced Signal Processing card. These are Creative ADPCM, CCITT A-Law and CCITT μ-Law.

This Developer Kit supports the following programming languages:

- Microsoft Assembler version 4.0 or later
- Microsoft C version 6.0 and 7.0
- Microsoft Visual C++ version 1.0 and 1.5
- Turbo C version 2.0
- Turbo C++ version 1.0
- Borland C++ version 2.0, 3.0 and 3.1
- Microsoft QuickBasic version 4.5
- Microsoft Basic Professional Development System version 7.1
- Microsoft Visual Basic for DOS version 1.0
- Turbo Pascal 6.0 and 7.0 (TPU only)

What You Should Know

This manual assumes you are an experienced programmer who is familiar with using the Sound Blaster cards or any of its derivatives. It also assumes that you have working knowledge with DOS function calls via Interrupt 21H. Therefore, the focus will be on technical aspects.

Some chapters in this manual assume additional knowledge on your part. The introductions to these chapters list these assumptions.

How to use this Manual

This manual provides information that experienced programmers can use to include SBC features in their application.

In this Programmer's Guide, we describe how to call the drivers and the library functions in order to invoke the following capabilities on the SBC:

- Digitized Sound Input and Output
- FM Synthesized Music Output
- MIDI Port Input and Output
- Mixer Control
- CD-ROM Drive Audio Interface

We also explain the interface of supported programming languages to the drivers and library functions. Sample programs which are ready for compilation and execution are provided. Supplement functions which help you use the SBK library are included together with their source code.

This manual is organized as follows:

Chapter 1, "Setting Up", describes the process of setting up the SBK and the relevant environment strings.

Chapter 2, "Programming Overview", discusses the general programming information you need to know when using SBK. We advise you to read this chapter carefully before proceeding to your development. This will help prevent unnecessary problems from occurring while you are programming.

Chapter 3, "Introduction to Creative Drivers", gives an overview of Creative Drivers.

Chapter 4, "High-Level Digitized Sound Drivers", describes the interface of the Creative High-Level Digitized Sound drivers, CT-VOICE, CTVDSK, CTWMEM and CTWDSK.

Chapter 5, "High-Level Auxiliary Driver", describes the interface and functions of the Creative High-Level Auxiliary driver. It covers how to add special effects like fade and pan to digitized sound and music.

Chapter 6, "Creative Multimedia System Driver", describes the interface of the Creative Multimedia System drivers, CTMMSYS.SYS.

Chapter 7, "MIDI Driver", describes the Creative MIDI driver interface. Read this when your program plays FM music, using Wave Blaster™, Midi Blaster™ or interfaces with other external MIDI devices.

Chapter 8, "CD-ROM Audio Interface", discusses the CD-ROM audio interface. It covers how to transform your CD-ROM drive into a CD Player.

Appendix A "Migration Guide", discusses how to convert your existing applications to use the new SBK libraries. This is only for existing Developer Kit users.

Appendix B, "Relevant information", lists the source of the supplement materials mentioned.

Glossary at the end of this manual defines the terms used in this manual and in the Library Reference and Hardware Programming Reference manuals.

Two other manuals that come with the SBK are Library Reference and Hardware Programming Reference. Library Reference is a reference to all the details of the drivers APIs and library functions. It also contains the Creative Voice file (VOC) format.

Hardware Programming Reference covers the details you need to know to program the Sound Blaster cards. This includes the hardware specifications, characteristics, commands, register maps etc. It is only intended for those who need to direct interface to SBC hardware.

This developer kit is only targeted for DOS developers. If you wish to write Microsoft Windows applications, you need to have the Microsoft Windows 3.1 Software Development Kit package. Microsoft Windows 3.1 Software Development Kit is a product of Microsoft Corporation.

Document Conventions

In this manual, the word "you" refers to you the developer or sometimes your application. The word "user" refers to the person who uses your applications.

SB1.5, SBMCV and SB2.0 are referred collectively as Sound Blaster, SBPRO and SBPRO MCV are referred collectively as Sound Blaster Pro, SB16 and Sound Blaster 16 Advanced Signal Processing are referred collectively as Sound Blaster 16. They are mentioned explicitly when refer to only one of them.

The term "Sound Blaster cards" is used to refer to the whole series of Sound Blaster cards.

Borland C++, Turbo C++, Turbo C and Microsoft C are referred collectively as C language.

Microsoft QuickBasic 4.5, Microsoft Basic PDS 7.1 and Microsoft Visual Basic version 1.0 are referred collectively as Microsoft Basic.

Turbo Pascal 6.0 and 7.0 are referred collectively as Turbo Pascal.

In this manual, we use the hexadecimal notation in C language to document hexadecimal number. For example, 0x0080 means 80 in hexadecimal.

To help you to locate and identify information easily, this manual uses visual cues and standard text formats. The following typographic conventions are used throughout this manual:

Example	Description
voice_drv, ctvdOutput	Bold letters indicate variable names, library functions or commands. These are case-sensitive i.e. upper and lower case are significant. Bold letters also use for terms intended to use as keywords or for emphasis in certain words.
MIXERVOL_CD	Bold all capital letters indicate manifest constant.
CT-VOICE.DRV	All capital letters indicate file names or directory names.
placeholders	Italic letters represent actual values or variables that you are expected to provide.
CTRL+ENTER	Small capital letters signify names of keys on the keyboard. Notice that a plus (+) indicates a combination of keys. For example, CTRL+ENTER means to hold down the CTRL key while pressing the ENTER key.
`program`	This font is used for example codes.

`program`
`.`
`.`
`.`
`fragment`
	Vertical ellipsis in an example program indicate that part of the program has been intentionally omitted.

[ ]	Square brackets in a command line indicate that the enclosed item is optional. It should not be typed verbatim.
< >	Angle brackets in a command line indicate that you must provide the actual value of the enclosed item. It should not be typed verbatim.
/	Slash in a command line indicates an either/or choice. It should not be typed verbatim.
Sound Blaster Pro (SBPRO)	Acronyms are usually spelled out the first time they are used.

SBK Version

To assist you in determining the version of the SBK libraries you are using, the following files are supplied on the distribution disks. When they are compiled and run, the SBK library version will be displayed.

These files are:

- SBKVER.BAS (for Basic)
- SBKVER.C (for C)
- SBKVER.PAS (for Turbo Pascal)

Chapter 1
Setting Up

This chapter explains how to setup the Developer Kit for Sound Blaster Series (SBK) and how to setup your development environment.

Also, look for a file named README.TXT in the package. This file contains the latest information about the SBK and changes that were not available at printing time.

What's in the package

The software provided on the SBK can be divided into several categories. In order to help you decide what to install, the following descriptions cover what is on the SBK disks.

Drivers For the latest drivers, you may download from our ftp or WWW sites at *ftp.creaf.com* or *http://www.creaf.com* respectively. For Compuserve, you can type "GO BLASTER".

Libraries The SBK comes with the libraries for all the high level languages and compilers it supports. You may want to remove the languages you are not interested in.

Examples The SBK includes the source code for example programs in each high-level languages it supports. These examples demonstrate the use of the SBK drivers and libraries. They are ready to be compiled and run. You may use these examples as the basis for writing your own SBC applications.

Helper code To facilitate the use of the SBK libraries, SBK includes many helper functions for the high-level languages that it supports. The source code for the helper libraries are also included.

Setting up your Development Environment

You need to setup your development environment so that your compiler can access the SBK files and libraries. You also need to setup the environment variables which will be used for your application development.

Environment Variables

We have enforced the concept of using the environment variables to signify information of our SBC. There are three environment variables used: **SOUND**, **BLASTER** and **MIDI**. Your applications need to access these variables. The users will have these environment variables set up during their SBC installation.

SOUND

The **SOUND** environment variable specifies the directory location of SBC drivers and software. Your application can check this environment setting to locate the SBC drivers and other SBC files.

The command for setting the SOUND environment variable is as follows:

SET SOUND=<*path*>

Note that there is no space before and after the = (equal) sign.

All our loadable drivers are located on the \DRV subdirectory below the SOUND path. For example, if your SOUND environment variable is C:\SB16, the loadable drivers are located in the C:\SB16\DRV subdirectory.

BLASTER

The **BLASTER** environment variable specifies the base I/O address, interrupt number and DMA channel hardware configuration of the SBC. The BLASTER environment reflects the current hardware configuration of users' SBC.

The command for setting the BLASTER environment is as follows:

SET BLASTER=A*220* **I***5* **D***1* [**H***5* **M***220* **P***330*]

where:

A	specifies the SBC base I/O port
I	specifies the interrupt request line
D	specifies the 8-bit DMA channel
H	specifies the 16-bit DMA channel
M	specifies the mixer chip base I/O port
P	specifies the MPU-401 base I/O port

Note that there is no space before and after the = (equal) sign, but there must be at least one space between two settings. Some SBC may have less environment parameters. For instance, 8-bit sound cards have no "**H***h*" on the BLASTER environment variable. If "**M***mmm*" is not specified, Mixer chip base I/O port will be the same as SBC base I/O port.

On Sound Blaster 16, 16-bit digitized sound data is usually transferred through 16-bit DMA channel (specified on the "**H***h*" parameter of BLASTER environment variable). However, the hardware also supports the transfer of 16-bit digitized sound data via the 8-bit DMA channel. To make this possible, the program **SBCONFIG.EXE** that comes with Sound Blaster 16 package must be run first to configure the Sound Blaster 16. When SBCONFIG is run, the BLASTER environment entries "**D***d*" and "**H***h*" must be set such that *d* and *h* are the same 8-bit DMA channel number.

MIDI

The **MIDI** environment variable specifies the MIDI file format used and where the MIDI data is sent to. The MIDI data can be sent to FM chips, Sound Blaster MIDI port or MPU-401 port if Sound Blaster 16 is used. Generally, there are three MIDI file formats available in the market, **General MIDI, Extended MIDI** and **Basic MIDI**. The differences between these files will be discussed in **MIDI Driver** chapter later.

The command for setting the MIDI environment is as follows:

SET MIDI=SYNTH:<*1/2*> **MAP:**<*G/E/B*>

where

SYNTH:	1	specifies internal synthesizer
	2	specifies MIDI port

MAP:	G	specifies General MIDI file format
	E	specifies Extended MIDI file format
	B	specifies Basic MIDI file format

If Sound Blaster 16 is used, the MIDI port will be MPU-401. For other Sound Blaster cards, MIDI port will be Sound Blaster MIDI.

This environment is interpreted by the MIDI driver to direct the output of MIDI code to. If it is not specified, it will be defaulted to internal synthesizer and Extended MIDI file format.

Include File and Library Directories

You need to setup your compilers working directories so that the compiler can access the SBK include files and libraries.

In the following discussion, we assume that your SBK software is installed in the C:\SBK directory. We also assume that you are familiar in configuring your compiler. Refer to your compiler manuals on how to configure it.

Microsoft C or Microsoft Basic

If you are using Microsoft C, Microsoft QuickBasic, Microsoft Basic Professional Development System (PDS) or Microsoft Visual Basic for DOS, you need to update the INCLUDE and LIB environment variables to tell the compiler where to find the SBK include files and libraries. Assuming your original INCLUDE and LIB are as follow:

SET INCLUDE=C:\INCLUDE

SET LIB=C:\LIB

You need to update them to include the SBK include files and libraries as follow:

SET INCLUDE=C:\SBK\INCLUDE;C:\INCLUDE

SET LIB=C:\SBK\LIB;C:\LIB

Notice that the path for SBK include files and libraries are added in front of the existing paths. This ensures that the latest include files and libraries are used in your development.

Borland C++, Turbo C++ or Turbo C

Integrated Development Environment(IDE)

If you are using Borland C++, Turbo C++ or Turbo C IDE, you need to update the IDE settings to tell the compiler where to find the SBK include files and libraries. These settings are in the **Include directories** and **Library directories** under the **Option | Directories** menu. For instance, if your settings of the **Include directories** and **Library directories** are C:\TC\INCLUDE and C:\TC\LIB, you need to update them to C:\SBK\INCLUDE;C:\TC\INCLUDE and C:\SBK\LIB;C:\TC\LIB. Remember to save these changes before you exit the IDE.

Command line level

If you are using Borland C++, Turbo C++ or Turbo C on command line level, you need to update the TURBOC.CFG configuration file to tell the compiler where to find the SBK include files and libraries. Assuming the option "**-I**" and "**-L**" in your TURBOC.CFG file are as follow:

-IC:\TC\INCLUDE

-LC:\TC\LIB

You need to update them to include the SBK include files and libraries as follow:

-IC:\SBK\INCLUDE;C:\TC\INCLUDE

-LC:\SBK\LIB;C:\TC\LIB

Turbo Pascal

Integrated Development Environment(IDE)

If you are using Turbo Pascal IDE, you need to update the IDE settings to tell the compiler where to find the SBK include files and units. These settings are in the **Include directories** and **Unit directories** under the **Option | Directories** menu. For instance, if your settings of the **Include directories** and **Unit directories** are C:\TP and C:\TP;C:\TP\BGI, you need to update them to C:\SBK\INCLUDE;C:\TP and C:\SBK\LIB;C:\TP;C:\TP\BGI. Remember to save these changes before you exit the IDE.

Command line level

If you are using Turbo Pascal on command line level, you need to update the TPC.CFG configuration file to tell the compiler where to find the SBK include files and units. Assuming the option "**/I**" and "**/U**" in your TPC.CFG file are as follow:

/IC:\TP

/UC:\TP;C:\TP\BGI

You need to update them to include the SBK include files and units as follow:

/IC:\SBK\INCLUDE;C:\TP

/LC:\SBK\LIB;C:\TP;C:\TP\BGI

Chapter 2
Programming Overview

This chapter gives an overview of the use of the various libraries and include files. It also discusses the information you need to know when using the SBK libraries.

Supported Programming Languages

This Developer Kit supports the following programming languages:

- Microsoft Assembler version 4.0 or later
- Microsoft C version 6.0 and 7.0
- Microsoft Visual C++ version 1.0
- Turbo C version 2.0
- Turbo C++ version 1.0 and 1.5
- Borland C++ version 2.0, 3.0 and 3.1
- Microsoft QuickBasic version 4.5
- Microsoft Basic Professional Development System version 7.1
- Microsoft Visual Basic for DOS version 1.0
- Turbo Pascal 6.0 and 7.0 (TPU only)

Borland C++, Turbo C++, Turbo C and Microsoft C use the same C libraries, include files and example programs provided on the SBK disks.

Microsoft QuickBasic 4.5, Microsoft Basic PDS 7.1 and Microsoft Visual Basic version 1.0 use the same Basic example programs included in the SBK disks.

Turbo Pascal 6.0 and 7.0 use the same Turbo Pascal example programs included in the SBK disks.

SBK Libraries

The following are libraries incorporated in the SBK disks:

C Language

SBKLIBS.LIB	for **small** memory model
SBKLIBM.LIB	for **medium** memory model
SBKLIBC.LIB	for **compact** memory model
SBKLIBL.LIB	for **large** and **huge** memory model

Microsoft Basic

SBKQB45.QLB SBKQB45.LIB	MS QuickBasic 4.5
SBKBC7.QLB SBKBC7.LIB	MS Basic PDS 7.1
SBKVB.QLB SBKVB.LIB	MS Visual Basic 1.0

Turbo Pascal

SBKTP6.TPU	for Turbo Pascal 6.0
SBKTP7.TPU	for Turbo Pascal 7.0

SBK Include Files

The include files are language-dependent and contain:

- Macro definitions
- Type-definitions
- Manifest constant declarations
- Global variable declarations
- Function and procedure declarations

You **must** include them whenever necessary so as to compile and link successfully. The following are include files incorporated in the SBK disks:

C Language

SBKAUX.H	for AUXDRV driver
SBKCD.H	for CD-ROM drive audio functions
SBKMIDI.H	for CTMIDI driver
SBKVOICE.H	for CT-VOICE and CTVDSK drivers
SBKWAVE.H	for CTWMEM and CTWDSK drivers

Microsoft Basic

SBKAUX.BI	for AUXDRV driver
SBKCD.BI	for CD-ROM drive audio functions
SBKMIDI.BI	for CTMIDI driver
SBKVOICE.BI	for CT-VOICE and CTVDSK drivers
SBKWAVE.BI	for CTWMEM and CTWDSK drivers

SBK Helper Functions

To help you use of SBK libraries, SBK includes many helper functions for supported high-level languages. The source code of these functions are also included.

When using the helper functions, you must incorporate the include files and link it with the libraries depending on the compiler you use.

Include File

SBKX.H	for C language
SBKX.BI	for Microsoft Basic

Library

CSBKX.LIB	for C language
QBSBKX.LIB	for Microsoft Quick Basic version 4.5
BC7SBKX.LIB	for Microsoft Basic PDS 7.1
VBSBKX.LIB	for Microsoft Visual Basic version 1.0
TP6SBKX.TPU	for Turbo Pascal 6.0
TP7SBKX.TPU	for Turbo Pascal 7.0

The helper functions' Quick Library is combined with the SBK Basic's Quick Library[1]. The same CSBKX.LIB C library is used for small, medium, compact and large C memory models.

In our example programs for C language, we use an include file SBKHELP.H to map the Microsoft C specific runtime library functions to the equivalent Turbo C or Borland C runtime library functions.

Building a SBC Application

Before writing a SBC application, you should have set up your development environment. Refer to the chapter "Setting Up" if you have not set up your development environment.

When using the SBK library functions, you must include the correct header files in all source modules that call a SBK library functions as mentioned. In addition, you must link to the correct SBK libraries when linking an application that uses SBK library functions.

You also need to include the helper functions include file and link with the helper functions library if your application uses any of the helper functions.

In the following discussion, DEMOMIX represents your program. The source code of this program can be found in the SBK disks.

String Format

The SBK library functions require that string be in a C compatible format (null-terminated string). Functions for converting Basic string and Turbo Pascal string format to C string format are included in the helper libraries.

Callback Constraint

The SBK drivers use callback mechanism to notify your application when certain events occurs. These callback actions are activated during the interrupt time and impose certain constraints.

---

[1]We would have preferred to separate the SBK quick libraries and helper functions' quick libraries, but Microsoft Basic only allows one quick library to be loaded at any one time.

The following are some constraints on the callback mechanism:

- PASCAL calling convention.
- No stack checking.
- Limitation on local variable within the callback function.
- No DOS function call is allowed within the callback function.
- The overall execution time of the callback function must not take too long to complete.
- Set DS register to your Data Segment before accessing your global variables

PASCAL calling convention

In order to use a single driver to support both the C and Pascal high-level programming languages, the Pascal calling convention has been adopted. For Pascal calling convention, the stack will be adjusted upon exit by the called function, so as to remove the parameters been pushed onto the stack during invoking. If the callback function is coded in Assembly Language, it will be your responsibility to adjust the stack accordingly.

No stack checking

The callback action occurs at interrupt time, and the stack had been switched to the driver stack. If you are using high level language like C, and the check stack function is enabled, it is very likely that the stack check function will indicate check stack failed when the function is activated. Therefore, you should disable the stack checking of your callback function. For instance, if you are using Microsoft C, include the /Gs switch in your command line compilation.

Limitation on local variable within the callback function

Usually, the SBK drivers reserve a minimum space for stack which is just sufficient for its operation to minimize the driver size. If you uses a large number of local variables, stack overflow may occur. This is because at the callback, the stack had been switched to the driver stack, and local variable uses the stack as storage. Therefore, it is your responsibility to ensure that the stack is not corrupted. You may switch to your own stack to accommodate your local

variables, but do remember to switch back to the driver stack before exiting from the callback function.

No DOS function call is allowed within the callback function

Due to the fact that DOS is non re-entrant, when the callback function is activated, it could be within one of the DOS function call. If the callback function uses any DOS function call, it may confuse DOS and hangs the system. Therefore, no DOS function call is allowed within the callback function.

The overall execution time of the callback function must not take too long to complete

As a rule of thumb, your callback function should does as minimum as possible. It should avoid holding the interrupt time for too long as no other programmable interrupts can occur within an interrupt service routine (unless you grant them specially).

Set DS register to your Data Segment before accessing your global variables

At the time the callback function is invoked by the driver, the data segment points to the driver data segment. Therefore, you should save the original data segment and set the DS register back to your data segment before accessing your global variables. If you do not set DS to your data segment, when you assign values to your global variables, you are actually corrupting the driver's data. Also, remember to restore the original callback data segment before your callback function exits if you have changed the original callback data segment.

Programming in C

The SBK supports Borland C++, Turbo C++, Turbo C and Microsoft C. They are using the same C include files and libraries.

The following discusses how to compile and link a program with SBK libraries in different C compilers environment.

Microsoft C

To compile and link your program in small memory model, you enter the following commands at DOS command line:

```
CL  /c  DEMOMIX.C
LINK  DEMOMIX, , ,SBKLIBS.LIB  CSBKX.LIB
```

Borland C++

There are two way to compile and link your program:

Integrated Development Environment

To compile and link your program, create a project file, e.g. DEMOMIX.PRJ and add all the relevant file items into the project file. For the case of DEMOMIX, add the following file items:

```
DEMOMIX
CSBKX.LIB
SBKLIBS.LIB
```

Command line level

Enter the following command at DOS command line:

```
BCC  DEMOMIX.C  CSBKX.LIB  SBKLIBS.LIB
```

Turbo C++

There are two way to compile and link your program:

Integrated Development Environment

To compile and link your program, create a project file, e.g. DEMOMIX.PRJ and add all the relevant file items into the project file. For the case of DEMOMIX, add the following file items:

```
DEMOMIX
CSBKX.LIB
SBKLIBS.LIB
```

Command line level

Enter the following command at DOS command line:

```
TCC  DEMOMIX.C  CSBKX.LIB  SBKLIBS.LIB
```

Turbo C

There are two ways to compile and link your program:

Integrated Development Environment

To compile and link your program, create a project file consisting of the following lines:

```
DEMOMIX
CSBKX.LIB
SBKLIBS.LIB
```

Then, specify the project file you have just created and press ALT+R.

Command line level

Enter the following command at DOS command line:

```
TCC  DEMOMIX  CSBKX.LIB  SBKLIBS.LIB
```

Programming in Microsoft Basic

We have included many helper functions for Microsoft Basic. They are mainly used for file I/O processes via DOS Interrupt 21H services. You are advised to refer to the DOS Technical Reference manual for DOS 3.0 or later for these functions if you encounter any problems.

By default, Microsoft Basic occupies all the remaining memory after it startup. If you are using DOS Interrupt 21H service to allocate memory, you have to use the Microsoft Basic **SETMEM** function to free sufficient memory in order for DOS to allocate memory successfully.

The **sbkLoadDriver** function uses the DOS Interrupt 21H service to allocate memory for the loadable driver. Therefore, you must use **SETMEM** function to set aside sufficient memory for the loadable driver when you use the **sbkLoadDriver** function.

There are two ways to compile and link your program:

Basic Environment

To start Microsoft Basic and compile the program DEMOMIX using the SBK Basic Quick library:

For Microsoft QuickBasic:

```
QB  /Ah  DEMOMIX  /L SBKQB45.QLB
```

For Microsoft Basic PDS 7.1:

```
QBX  /Ah  DEMOMIX  /L SBKBC7.QLB
```

For Microsoft Visual Basic for DOS version 1:0:

```
VBDOS  /Ah  DEMOMIX  /L SBKVB.QLB
```

Then press ALT+R and select **Make EXE File** option under the **Run** menu to compile.

Command line level

Enter the following command at DOS command line:

For Microsoft QuickBasic:

```
BC  DEMOMIX  /Ah  /O
LINK  DEMOMIX, , , SBKQB45.LIB  QBSBKX.LIB;
```

For Microsoft Basic PDS 7.1:

```
BC  DEMOMIX  /Ah  /Lr  /Fs  /O
LINK  DEMOMIX, , , SBKBC7.LIB  BC7SBKX.LIB;
```

For Microsoft Visual Basic for DOS version 1:0:

```
BC  DEMOMIX  /Ah  /O
LINK  DEMOMIX, , , SBKVB.LIB  VBSBKX.LIB;
```

Note that the /Ah command option is only required if any of the arrays take more than 64KB of space or it has more than 16,384 elements (since each integer takes 4 bytes).

Programming in Turbo Pascal

In Turbo Pascal programming, we assume that you have working knowledge with DOS Interrupt 21H services. You are advised to refer to the DOS Technical Reference manual for DOS 3.0 or later for these functions if you encounter any problems in the example programs.

The default memory allocation for Turbo Pascal is 16KB for the stack and 640KB for the heap. This means that, by default, the heap will occupy the remaining memory. Therefore, if you are using DOS to allocate memory, you need to free some heap memory (by reducing the maximum heap size using the **{$M}** directive) for DOS to allocate memory successfully.

When using the SBK Units, you have to include the following line

```
uses SBKTP6, TP6SBKX;     For Turbo Pascal 6.0
```

or

```
uses SBKTP7, TP7SBKX;     For Turbo Pascal 7.0
```

at the top of your program to tell the compiler that your program uses these units. For examples:

```
Program  demomix;

uses  sbktp6,  tp6sbkx;

Begin
   :
   :
End.
```

There are two ways to compile and link your program:

Integrated Development Environment

To start Turbo Pascal and compile the program DEMOMIX:

TURBO DEMOMIX

Then press ALT+C to compile.

Command line level

Enter the following command at DOS command line:

TPC DEMOMIX

Chapter 3
Introduction to Creative Drivers

This chapter describes the drivers available for the Sound Blaster family and how to use them.

This chapter covers the following topics:

- About Creative Drivers
- Architecture of Creative audio drivers
- Architecture of Creative MIDI loadable driver
- Architecture of Creative CD-ROM drivers
- Using Creative loadable drivers
- Using Creative low-level driver

About Creative Drivers

The following drivers are provided on the SBK distribution disks:

Loadable Drivers

CT-VOICE.DRV	.VOC memory driver
CTVDSK.DRV	.VOC disk-double buffering driver
CTWMEM.DRV	.WAV memory driver
CTWDSK.DRV	.WAV disk-double buffering driver
CTMIDI.DRV	MIDI driver
AUXDRV.DRV	Auxiliary driver

Device Drivers

CTMMSYS.SYS	Low-level audio driver
CTSOUND.SYS	Hardware dependent sound driver
CSP.SYS	Creative Advanced Signal Processor driver

A **Loadable Driver** is a binary image of the driver code. Your application program first allocates a block of memory and loads the loadable driver into it. The driver is then invoked with a FAR CALL. However, you will not have to deal directly with these setup and invocation details because the SBK library provides covenient high-level language interfaces to these drivers.

A **Resident Driver** is a terminate and stay resident (TSR) program. It has to be installed into memory before your application begins. The driver is then invoked by making interrupt calls. Again, the SBK provides a more convenient loadable interface driver to the resident driver.

A **Device Driver** is a driver installed at boot-up time. The device drivers included in SBK are character-mode DOS device drivers. The driver is invoked not via IOCTL commands but through far calls to the driver entry-point. Again, we have included helper functions and sample programs to show you how to use these drivers.

The **Loadable**, **Resident** and **Device Drivers** are all external drivers. They are separate from your program code. The main reason for adopting this approach is the ease of upgrade. We keep these drivers in separate files. Thus, your programs are insulated from changes in the drivers.

If we update any of these drivers, your programs need not be re-compiled or re-linked. They will continue to run. We will also be able to maintain better third-party software compatibility when we make changes to the hardware. We will continue to

use the external driver approach and provide compatible drivers for the development of future audio cards. It will be easy for your applications to support the new hardware soon after they become available.

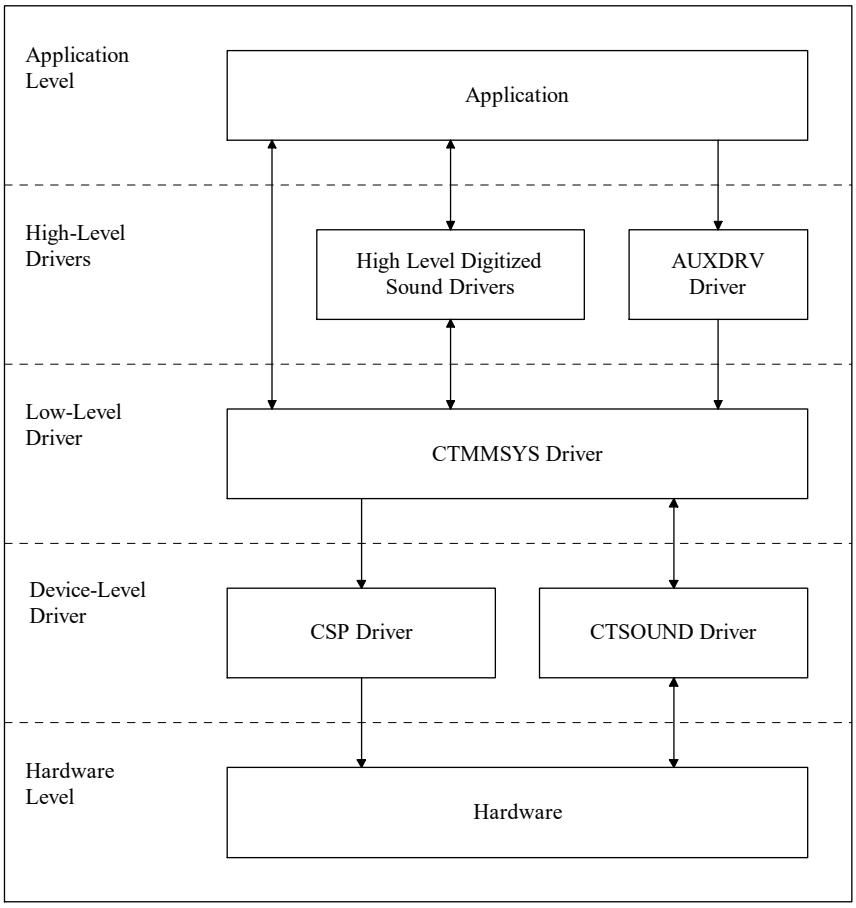
Another advantage is that only a copy of the driver needs to be kept on the system. This makes your programs smaller and saves user disk space. The loadable drivers are kept in the DRV sub-directory specified in the "SET SOUND=" environment variable.

You do not really have to worry about these drivers because the SBK library makes programming easy by providing the high-level language interfaces. You will be calling the library functions to process Digitized sound and Music.

Architecture of Creative Audio Drivers

The architecture of Creative audio drivers is separated into three different levels: high-level, low-level and device-level.

The following illustration shows the relationship between the application and Creative audio drivers.



The relationship between the application and the Creative audio drivers

High-Level Drivers

The high-level drivers are hardware-independent drivers. These are the high-level digitized sound drivers and AUXDRV driver.

The high-level digitized sound drivers provided are CT-VOICE, CTVDSK, CTWMEM and CTWDSK. Individual drivers support playback from or recording to either memory or disk. The Creative Voice File (.VOC) and Multimedia Wave File (.WAV) formats are supported.

The high-level digitized sound drivers make recording and playback easier, and require less programming.

The AUXDRV driver is a driver that controls the volume, tone, gain source selection and so on.

Low-Level Driver

The low-level driver, CTMMSYS isolates applications from the device-level drivers and centralizes device-independent code. Applications and high-level drivers invoke this driver to perform digitized sound I/O or control auxiliary device. Called to the low-level driver will be translated to the corresponding device-level driver calls, or they might even be handled purely by CTMMSYS itself.

CTMMSYS gives you more control over digitized sound recording and playback, but requires more programming than high-level drivers. Therefore, if your application requires digitized sound I/O or auxiliary device controls, you should first try the high-level drivers. If they do not meet the needs of your application, then use CTMMSYS to write your own routines to manage digitized sound recording and playback.

Device-Level Drivers

The device-level drivers are the hardware-dependent drivers which communicate directly with Sound Blaster hardware.

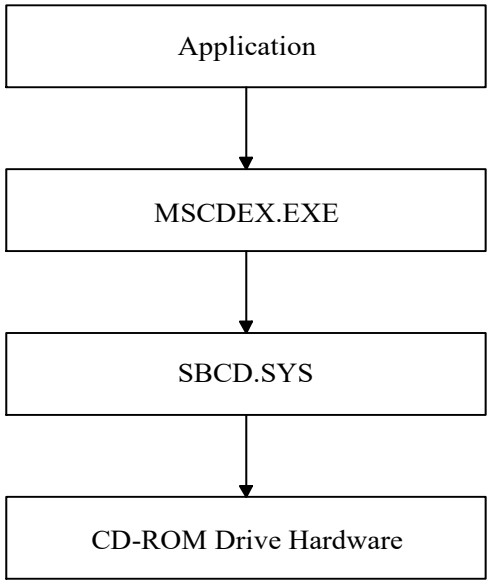
CTSOUND.SYS is a device driver that communicates with Sound Blaster's Digital Sound Processor (DSP) and Mixer.

CSP.SYS is a device driver that mediates access to the Creative Advanced Signal Processor chip on the Sound Blaster 16 Advanced Signal Processing.

Your application does not to invoke these driver directly. It should instead invoke the low-level driver CTMMSYS if any low-level device controls are needed.

Architecture of Creative CD-ROM Drivers

The following block diagram shows the relationship between the application and CD-ROM drivers:



The Relationship between the Application and the CD-ROM drivers

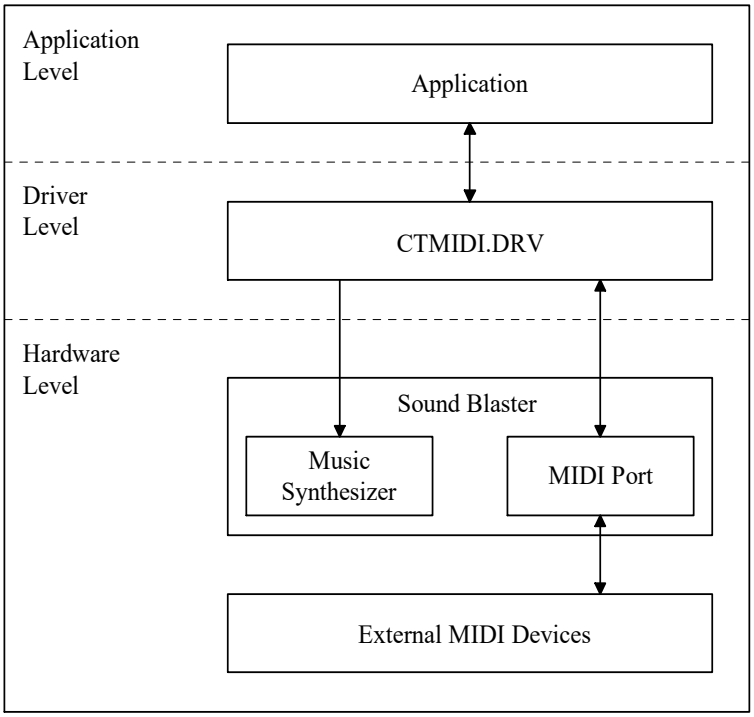
The Microsoft Compact Disc Extension driver MSCDEX.EXE, is a driver distributed by Microsoft Corporation. It provides a hardware-independent interface between application programs and the hardware-dependent CD-ROM driver. MSCDEX.EXE is a Terminate and Stay Resident (TSR) program. It must be installed before you can run any of the CD-ROM applications.

The Creative hardware-dependent CD-ROM driver SBCD, provides a means of interfacing an application with the CD-ROM drive hardware. It is a device driver which is installed through CONFIG.SYS.

The two drivers mentioned come with your CD-ROM drive and hence are not included in the SBK distribution disks. To facilitate programming access to the CD-ROM drive, the SBK provides a comprehensive CD-ROM Audio Interface Library.

Architecture of Creative MIDI Loadable Drivers

The following block diagram shows the relationship between the application and the Creative MIDI loadable driver, CTMIDI.DRV:



The Relationship between the Application and the MIDI Loadable Driver

CTMIDI.DRV is a loadable interface driver that processes the MIDI file, and channels the data to the selected MIDI output device. The MIDI output devices could be the music synthesizer chip on a Sound Blaster card or external MIDI devices. Output to external MIDI devices is sent through the MIDI port on the Sound Blaster card. CTMIDI.DRV also handles in-bound MIDI data.

There are two ways for specifying the MIDI output device. By default, CTMIDI.DRV reads MIDI environment for the MIDI output device (see "Environment Variables" on chapter 2). Therefore, your application will not control what output device to be used. It depends on how your user specifies the MIDI environment. CTMIDI.DRV also provides functions for your application to control the output device.

Using Creative Loadable Drivers

The proper sequence for using a Creative loadable driver is outlined below:

1. Check the existence of the driver. The driver should be located in the \DRV sub-directory of the directory specified by **SOUND** environment. If the driver could not be found, then check the current directory for it.

2. Allocate a memory buffer big enough for the driver, and ensure that the buffer starts at a 16-byte paragraph boundary (i.e. the normalized far address must have an offset of zero). If the allocated buffer is not on a paragraph boundary, allocate 15 bytes more and perform a paragraph boundary adjustment.

3. Load the driver into the memory buffer.

4. Invoke the driver by using the SBK library functions or make a far call to offset zero of the memory buffer.

The services provided by the loadable driver are invoked by making a far call to the driver entry point. The function arguments are conveyed to the driver via registers. However, you do not have to worry about these as the SBK high-level library functions place your function arguments in the appropriate registers.

If you are using C or Turbo Pascal, you need to initialize the appropriate global far pointer (see the table below) to the loaded driver's entry-point for the SBK libraries to invoke the assembly interface functions. In C, these global far pointers are declared **extern** in the header file. In Turbo Pascal, they are declared in the **SBKTP?.TPU** unit.

Driver	far pointer
CT-VOICE.DRV	voice_drv
CTVDSK.DRV	ctvdsk_drv
CTWMEM.DRV	CTwmemDrv
CTWDSK	CTwdskDrv
AUXDRV.DRV	CTAuxDrv
CTMIDI.DRV	CTmidiDrv

If you are using Microsoft Basic, you need to call the appropriate subroutine in the table below to set up the loaded driver entry point for the SBK libraries to invoke the assembly interface functions:

Driver	Subroutine
CT-VOICE.DRV	ctvmSetDriverEntry
CTVDSK.DRV	ctvdSetDriverEntry
CTWMEM.DRV	ctwmSetDriverEntry
CTWDSK	ctwdSetDriverEntry
AUXDRV.DRV	ctadSetDriverEntry
CTMIDI.DRV	ctmdSetDriverEntry

To help you understand how to load the loadable driver from your program, we have included source code for the load driver function (in version for Microsoft C, Turbo C, Turbo Pascal and Microsoft Basic) in the SBK disks. You may include this function in your program to load the drivers. Refer to the sample programs to see how to load a driver and establish a link to the driver.

Using Creative Low-Level Driver

You must install the low-level driver before using it. The low-level driver is installed with an entry in CONFIG.SYS. The syntax of the entry is:

Device=<*full path to device driver*> <*parameter lists*>

Communication with the Creative low-level driver is through a far call to the driver-entry point. The steps needed to obtain the entry-point to the low-level driver are:

1. Open the low-level device driver with the device name.
2. Perform a Device IOCTL read to the driver to obtain the entry point.
3. Close the device driver.

The function arguments are conveyed to the device driver via the user stack, using the Pascal calling convention (i.e. arguments are pushed into the stack from left to right).

We have included helper functions and example programs to show you how to use the Creative low-level driver. You may use these example programs as the basis for writing your applications.

Chapter 4
High-Level Digitized Sound Drivers

This chapter describes the programming information on the digitized sound recording and playback in **Conventional Memory**, **Extended Memory** and **Disk** using Creative high-level digitized sound drivers.

Two major digitized sound file formats are supported with the aid of Creative high-level digitized sound drivers. They are **Creative Voice File (.VOC)** and **Multimedia WAVE file (.WAV)**. The .WAV file is a digitized sound file format commonly used in Multimedia Windows environment.

This chapter covers the following topics:

- Using Creative high-level digitized sound drivers
- Playing and recording digitized sound

This chapter assumes you have some knowledge on the digitized sound. If you need additional information on this subject, you are advised to refer to the **Relevant Information** appendix.

Using Creative High-Level Digitized Sound Drivers

This section imparts the essence of using the high-level digitized sound drivers. There are four high-level digitized sound drivers: CT-VOICE, CTVDSK, CTWMEM and CTWDSK.

Functionality

The functionalities of these drivers are shown as follow:

Driver	Functionality
CT-VOICE	Providing Memory-mode .VOC I/O
CTVDSK	Providing Disk-mode .VOC I/O
CTWMEM	Providing Memory-mode .WAV I/O
CTWDSK	Providing Disk-mode .WAV I/O

The drivers maintain the same set of API across the cards. Each Sound Blaster card has a specific set of these drivers. The APIs of these drivers are kept very similar. Essentially, we hope that if you understand one driver well, you will practically have mastered them all.

Function Prefixes

Function names of the high-level digitized sound drivers begin with the following prefixes:

Prefix	Driver
ctvm	CT-VOICE
ctvd	CTVDSK
ctwm	CTWMEM
ctwd	CTWDSK

Include Files

You need to include the following header files in all your applications that use the high-level digitized sound drivers functions:

Driver	C Language	Microsoft Basic
CT-VOICE	SBKVOICE.H	SBKVOICE.BI
CTVDSK	SBKVOICE.H	SBKVOICE.BI
CTWMEM	SBKWAVE.H	SBKWAVE.BI
CTWDSK	SBKWAVE.H	SBKWAVE.BI

Loading and Initializing the Drivers

You must load the driver into memory with an offset zero of a segment before you can use it. You must also assign the far pointer (listed below) to the far address of the very first byte of the loaded driver. These far pointers are required by the high programming language wrappers to invoke the assembly interface of the drivers.

Driver	Far Pointer
CT-VOICE	voice_drv
CTVDSK	ctvdsk_drv
CTWMEM	CTwmemDrv
CTWDSK	CTwdskDrv

After loading the driver, you should check for the driver version number using the **ct??GetParam** function. All drivers must have a version number 4.00 or later for the new API supports. You should notify the user if the driver version number is lower than 4.00 and unload the driver. The process of unloading a loadable driver is simple; just release the memory where the driver was loaded.

Having determined the correct driver version number, you pass the BLASTER environment string to the driver for it to determine the hardware setting to use. You do this by calling the **ct??GetEnvSettings** function. After that, call the **ct??Init** function to initialize the driver. The **ct??Init** function must be called before any other driver function of the driver except the **ct??GetParam** and **ct??GetEnvSettings** functions.

When you have finished using the driver, you must call the **ct??Terminate** function to perform the necessary driver termination activities. When these activities are completed, release the memory buffer for the driver.

The following pseudo code fragment describe the process for using the CT-VOICE high-level digitized sound driver. The same code skeleton applies to other high-level digitized sound drivers:

```
// Load CT-VOICE.DRV driver and assign the entry to voice_drv
voice_drv := sbkLoadDriver("CT-VOICE.DRV", 0, lpOrgPtr);

IF driver loaded successful THEN
{
        // check for correct driver version
        ctvmGetParam(CTVOC_DRIVERVERSION, (Address of)lpdwParam);
        wDrvVersion := (low word of)lpdwParam;

        IF wDrvVersion >= 305 hex THEN
        {
                // pass BLASTER environment string to driver
                //
                // 1.  Get BLASTER environment string
                // 2.  Pass to driver by ctvmGetEnvSettings()
                ctvmGetEnvSettings(BLASTER string);

                IF ctvmInit() = 0  THEN
                {

                        // Initialization successful.
                        // strut your CT-VOICE function calls here
                        .
                        .
                        .

                        // when you have finished using CT-VOICE
                        ctvmTerminate();
                }
        }
        ELSE
        {
                // incorrect driver version
        }


        // unload the loaded driver here by releasing
        // the loaded driver memory location.
}
ELSE
{
        // Load driver error
}
```

The **sbkLoadDriver** function is included in the SBK as a helper function in Microsoft Basic and Turbo Pascal or as a module file in C language. Refer to this function's source code to see how it works.

Playing and Recording Digitized Sound

This section describes the use of high-level digitized sound drivers for digitized sound playing and recording.

General Information

I/O Handle

All the digitized sound drivers function calls dealing with I/O processes require an **I/O Handle**. An I/O handle is a unique value used to identify a digitized sound I/O process. The drivers use the I/O handle internally to identify which I/O channel the process is using. Therefore, you must query the digitized sound drivers to ensure at least one I/O handle is available before proceeding with your digitized sound I/O process.

The high-level digitized sound drivers include the following functions for you to query the number of input and output handles available:

ctv?GetParam(CTVOC_INPUTHANDLES, &*NumInputHandles*)
ctw?GetParam(CTWAV_INPUTHANDLES, &*NumInputHandles*)

ctv?GetParam(CTVOC_OUTPUTHANDLES, &*NumOutputHandles*)
ctw?GetParam(CTWAV_OUTPUTHANDLES, &*NumOutputHandles*)

You use a value in the range of 0 to one less than the total of *NumInputHandles* and *NumOutputHandles* for the I/O handle parameter to perform digitized sound I/O. For example, if the *NumInputHandles* returns a value of 2 and the *NumOutputHandles* returns a value of 4, the valid I/O handles are from 0 and 5 for either recording or playing.

Querying the Digitized Sound Drivers

The digitized sound drivers provide the following functions to query the drivers and digitized sound I/O information:

Function	Description
ct??GetIOParam	Gets the digitized sound I/O information.
ct??GetParam	Gets the digitized sound drivers information.

You can call the **ct??GetParam** function before the **ct??Init** function. The **ct??GetParam** function is used to query the following information:

- version number of the digitized sound driver
- card type number supported by the driver
- name of the supported sound card
- I/O channels supported by the sound card
- sampling rate range supported by the sound card
- I/O handles available
- build number of the driver
- DMA transfer buffer is required
- actual size of the driver without embedded DMA buffer (for .VOC drivers)
- embedded DMA buffer size (for .VOC drivers)

You use the **ct??GetIOParam** function to retrieve the digitized sound I/O information set using the **ct??SetIOParam** function. Refer to the **Library Reference** manual for the details of these two functions.

Setting DMA Buffer

To ensure the highest quality recording and playback, the drivers now make use of the **Auto-initialize** DMA mode for digitized sound I/O processes. For this, a DMA buffer is required by the driver, which uses the **double-buffering** technique to move data in or out of the buffer. The application has to allocate a block of memory and then call **ct??SetDMABuffer** to inform the driver of the size and location of the buffer.

However, to ensure that an application coded for the old drivers will continue running on the new drivers, it was necessary to embed[2] a DMA buffer in the CT-VOICE and CTVDSK drivers. The application may query the size of the embedded buffer by using **ctv?GetParam(CTVOC_EMBDDMABUFSIZE**, *&BufferSize*).

---

[2]We would have preferred to call upon DOS to allocate the memory for the DMA buffer from the drivers, except that this interferes with the run-time memory management system in applications produced by some compilers.

If the buffer size is deemed to be satisfactory, then the application need not call the **ctv?SetDMABuffer** function. Be forewarned, however, that the embedded DMA buffer is only assigned to I/O handle zero, even when the driver supports more than one I/O handle. So for an I/O handle of value greater than zero, the application must allocate DMA buffer, and call the **ctv?SetDMABuffer** function to inform the driver.

If the buffer size is deemed unsatisfactory, the application should allocate its own DMA buffer and call the **ctv?SetDMABuffer** function. However, this would leave the embedded buffer sitting around uselessly in memory. Furthermore, due to DMA constraints, the DMA buffer cannot straddle a 64KB page boundary. And since the application can load the driver anywhere in conventional memory, we have been forced to take precautionary measure of embedding a buffer that is two times larger than what is actually used by the driver. Fortunately, since the embedded buffer is placed at the very end of the driver, the application can query the size of the driver, without the embedded buffer, with **ctv?GetParam**(**CTVOC_DRIVERSIZE**, &*DriverSize*).

The application could then free the current memory block allocated to the driver, allocate a smaller block of size *DriverSize* bytes, and then load in only the first *DriverSize* bytes of the driver. This time round, the application must call **ctv?SetDMABuffer** function before attempting any I/O with I/O handle 0; otherwise the driver will mistakenly use memory that does not belong to it, for the default DMA buffer.

Please note that the two .WAV drivers, namely CTWMEM and CTWDSK, **do not** have embedded DMA buffers. The application must call the **ctw?SetDMABuffer** function before attempting any I/O.

The syntax of the **ct??SetDMABuffer** function is as follows:

ct??SetDMABuffer(*wIOHandle*, *dw32BitAddx*, *w2KBHalfBufferSize*)

The *wIOHandle* parameter is a WORD and identifies the digitized sound I/O process. This must be a valid I/O handle obtained through the call to the **ct??GetParam** function.

The *dw32BitAddx* parameter is a DWORD and specifies the 32-bit linear address of the DMA buffer. The DMA buffer must be sited entirely within the lowest 1MB of memory.

The *w2KBHalfBufferSize* parameter is a WORD and specifies the size of DMA buffer. It is in the unit of 2KB per half buffer. For example, if *w2KBHalfBufferSize* is 4, it means a 16KB DMA buffer is to be used. The acceptable range for *w2KBHalfBufferSize* is from 1 to 16.

Because the DMA buffer cannot straddle a 64KB page boundary constraints, you should always allocate the DMA buffer two times larger than the desired size (unless you are sure that the buffer allocated is within a 64KB page boundary). Set the DMA buffer with the **ct??SetDMABuffer** function. If the function returns fail, you should call the function once again with the *dw32BitAddx* parameter set to the next half of the buffer (original *dw32BitAddx* + ½ * *allocated DMA buffer size*). If the allocated buffer does not start on the paragraph boundary, you should allocate the buffer with an additional 15 bytes, because the driver will do a paragraph adjustment internally.

The following pseudo code fragment describe the process for setting the DMA buffer of the high-level digitized sound drivers:

```
// Allocate two 8KB (8192 bytes) for DMA buffer.
// The other half is used when the 1st half straddles a 64KB boundary.
// Also, allocate 16 bytes more for each buffer to paragraph alignment.
lpDMABuffer := AllocateMemory((8192 + 16) * 2);

// Now, convert the lpDMABuffer to 32 bit linear address
dw32BitAddx := (32-Bit linear address of lpDMABuffer);

// Setup DMA buffer.  If failed, the DMA buffer may has straddled
// 64KB boundary, thus use the 2nd DMA buffer.
if (ct??SetDMABuffer(wIOHandle, dw32BitAddx, 2)) THEN
{
        dw32BitAddx := dw32BitAddx + (8192 + 16);

        if (ct??SetDMABuffer(wIOHandle, dw32BitAddx, 2)) THEN
        {
                // Set DMA error !!!!
        }
}
```

Setting Disk Buffer

In addition to requiring one DMA buffer per I/O handle, the disk drivers (CTVDSK and CTWDSK) also require a corresponding disk buffer for digitized sound I/O.

The disk buffer serves as an intermediate storage for the digitized sound data before it is transferred to/from the DMA buffer. Hence, the disk buffer must be **at least twice** the size of the DMA buffer.

The two-layer buffered approach will help to achieve smoother I/O at high sampling rates provided the disk buffer is large enough. Lets use the digitized sound output for illustration. The digitized sound data is pre-loaded into the disk buffer before being transferred to the DMA buffer at each DSP interrupt. If the disk buffer is larger, it will be able to hold more data, thus reducing the frequency of disk access.

The following function is provided to set the disk buffer for disk drivers:

ct?dSetDiskBuffer(*wIOHandle*, *lpBuffer*, *w2KBHalfBufferSize*)

The buffer size is in the unit of 2KB per half buffer as the DMA buffer. The acceptable range for *w2KBHalfBufferSize* is from 2 to 32. However, it must be at least twice the size of the DMA buffer. For example, if *w2KBHalfBufferSize* of DMA buffer is 2, the buffer size must be 4 or more.

Unlike the DMA buffer, it does not have the 64KB page boundary constraints. You can allocate the disk buffer anyway in conventional memory. If the allocated buffer does not start on a paragraph (16-byte) boundary, you must allocate the disk buffer with an additonal 16 bytes because the driver will do a paragraph alignment internally.

No digitized sound I/O for disk drivers can be carried out until a valid disk buffer has been setup.

Handling Disk Driver Errors

When a disk driver function fails, you can obtain the error code and determine what was the cause of the error. The following functions are provided to handle the disk driver errors:

Function	Description
ct?dGetDrvError	Get the driver's error code for the last operation.
ct?dGetExtError	Get the error code returned by the failed DOS call for a driver I/O operation.

The **ct?dGetDrvError** function returns the error code for the last operation. If the error involves DOS, you should also invoke the **ct?dGetExtError** function to correspond with the error code returned by DOS.

Playing Digitized Sound

After setting the DMA buffer (and disk buffer if you are using disk drivers), you are ready to use the driver for the playing of digitized sound. The high-level digitized sound drivers support three mode of digitized sound playback: **conventional memory**, **extended memory** and **disk**. The following functions are provided to start digitized sound playing:

Function	Description
ct?mOutputCM	Playing digitized sound from conventional memory.
ct?mOutputXM	Playing digitized sound from extended memory.
ct?dOutput	Playing digitized sound from disk.
ctvdOutputOffset	Playing digitized sound from disk with the offset specified.

If you are playing digitized sound from conventional memory, you must first load the digitized sound file into the conventional memory. Similarly, if you are using the extended memory for digitized sound playing, you must load the digitized sound data into extended memory. The loading of digitized sound data to extended memory must be done through an extended memory manager which is **HIMEM.SYS** compatible. We have included some helper functions and example programs for transferring digitized sound data between conventional memory and extended memory. If you need more information using the extended memory manager, refer to the **XMS Extended Memory Specification Version 2.0**.

If you are playing digitized sound from disk, you need to open the file and pass the file handle to the disk driver using the **ct?dOutput** function.

You must pass the first .VOC data block to the **ctvmOutputCM** and **ctvmOutputXM** functions. For the following four functions, **ctvdOutput**, **ctwdOutput**, **ctwmOutputCM** and **ctwmOutputXM**, you must pass the start pointer of the digitized sound file.

If you wish to play a specific block of .VOC digitized sound file from disk, you can use the **ctvdOutputOffset** function. The **ctvdOutputOffset** function plays a .VOC digitized sound file from disk with the offset specified. In such as case, you must ensure that the specified offset is on a start of a .VOC block.

After you start playing, control returns to your application immediately. The playing takes place in the background.

Monitoring Digitized Sound Status

While a digitized sound is playing, you can monitor the process status with the help of its status word. The high-level digitized sound drivers allow you to set the address of the digitized sound status word using the following function:

ctv?SetIOParam(CTVOC_IO_LPSTATUSWORD, &*wVoiceStatus*)

ctw?SetIOParam(CTWAV_IO_LPSTATUSWORD, &*wVoiceStatus*)

After setting the address of the status word, you can use it to monitor the digitized sound process. The digitized sound drivers update the status word under the following conditions:

1. Resets the status word to zero during initialization.
2. Sets the status word to FFFF hex when it starts a new digitized sound I/O process.
3. Sets the status word to 0 at the end of the digitized sound I/O process.
4. Updates the status word with the marker value in the .VOC marker block when one is encountered.
5. Keeps the status word unchanged when the pause, continue and break loop functions are invoked.

With the aid of the markers in a .VOC file, you are able to synchronize your foreground application with the background playing digitized sound. For instance, your application can monitor the digitized sound status word until a desired marker value is encountered before triggering an event.

Controlling Digitized Sound Playback

The high-level digitized sound drivers provide the following functions for controlling digitized sound playback:

Function	Description
ctv?BreakLoop	Break out a repeat loop in the current .VOC digitized sound output process.
ct??Continue	Continue the paused digitized sound output process.
ct??Pause	Pause the active digitized sound output process.
ct??Stop	Stop the digitized sound input or output process.

While the digitized sound is playing, you can stop or pause playback. Pause, continue and stop functions perform immediately when they are invoked.

You use the **ctv?BreakLoop** function to break out a repeat loop in the current .VOC digitized sound output process.

Recording Digitized Sound

In addition to setting the DMA buffer (and disk buffer if you are using disk drivers), you need to setup the digitized sound recording parameters before you can begin recording.

Before you setup the recording parameters, you should query the driver with the **ct??GetParam** function to determine the sampling rate and recording channel (mono or stereo) that the sound card can support. This will avoid setting the recording parameters beyond the capabilities of the sound card.

You use the **ct??SetIOParam** function to set the following recording parameters:

- recording sampling rate
- recording channels (mono or stereo)
- recording source
- recording digitized sound format
- bits per recorded sample

Default parameters will be used if a parameter is not set. Refer to the **Library Reference** manual for the details of the **ct??SetParam** functions.

The high-level digitized sound drivers support three modes of digitized sound recording: **conventional memory**, **extended memory** and **disk**. The following functions are provided to start digitized sound recording:

Function	Description
ct?mInputCM	Recording digitized sound to conventional memory.
ct?mInputXM	Recording digitized sound to extended memory.
ct?dInput	Recording digitized sound to disk.

If you are recording digitized sound to conventional memory, you must first allocate a conventional memory buffer and pass it to the **ct?mInputCM** function with the buffer size specified. Similarly, if you are using the extended memory for digitized sound recording, you must allocate an extended memory buffer and pass it to the **ct?mInputXM** function with the buffer size specified. Like the playing from extended memory, the extended memory allocation must be done through an extended memory manager which is **HIMEM.SYS** compatible. If you need more information on using the extended memory manager, refer to the **XMS Extended Memory Specification Version 2.0**.

If you are recording digitized sound from disk, you need to create a file and pass the file handle to the disk driver via the **ct?dInput** function.

After initiating the recording, control returns to your application immediately. The recording takes place in the background. You may call the **ct??Stop** to end the digitized sound recording. Otherwise, the recording ends when the memory buffer or disk space is full depending on whether you record to memory or to disk respectively.

The **ctvmInputCM** and **ctvmInputXM** functions do not create a .VOC File Header block in front of the recorded digitized sound data. It is your responsibility to add the .VOC File Header Block if you wish to save it as a disk .VOC file.

For the **ctvdInput** function, a .VOC File Header Block will be created in front of the recorded data.

For the **ctwdInput**, **ctwmInputCM** and **ctwmInputXM** functions, a .WAV File Header will be created before the digitized sound data.

You can use the digitized sound status word to monitor the recording progress. The drivers set the status word to FFFF hex during the recording and set it to zero when the recording is terminated.

Chapter 5
High-Level Auxiliary Driver

This chapter describes how to program the mixer chip using the Creative high-level loadable auxiliary driver (AUXDRV.DRV).

The driver provides the API of mixing sources like digitized sound, MIDI, auxiliary line-in, CD audio, microphone input and the PC speaker output.

It provides software volume control for digitized sound, MIDI[3], auxiliary line-in, CD audio, microphone and overall master volume. It also allows the tone control of bass and treble, input and output gain control and automatic gain control that features on SB16.

By controlling the individual left/right volume levels of stereo sources, the auxiliary driver is able to produce fading and panning effect.

This chapter covers the following topics:

- Using the auxiliary driver
- Controlling volume, tone level and input/output gain
- Selecting recording and playback source
- Producing fade and pan effects

[3]The terminology has been changed from FM to MIDI because a non-FM device like Wave Blaster (which provides wave-table MIDI synthesis) may be attached to the MIDI Extension Connector. To the mixer, it's one and the same.

Using the AUXDRV driver

Although there are enhancements of the mixer chip on Sound Blaster from card to card, the AUXDRV driver maintains a consistent set of API. Each Sound Blaster card has a specific AUXDRV.DRV. If the features being accessed are not available on the card, the AUXDRV performs a dummy return.

You must load the AUXDRV into memory with an offset zero of a segment before you can use it. The global variable **CTAuxDrv** must be assigned to the far address of the very first byte of the loaded driver. Refer to the chapter "Introduction to Creative Drivers" on the details of loading a loadable driver.

After loading the driver, you should check for the driver version number via the **ctadGetDrvVer** function. The driver version number must be 3.04 or later for the new API supports. You should notify the user if the driver version number is lower than the mentioned and unload the loaded driver. The process for unloading a loadable driver is simply to release the memory where the driver was loaded.

Having determined the correct driver version number, you pass the BLASTER environment string to the driver for it to determine what hardware setting to use. You do this by calling the **ctadGetEnvSettings** function. Now, call the **ctadInit** function to initialize the driver. The **ctadInit** must be called before any other AUXDRV functions except **ctadGetDrvVer** and **ctadGetEnvSettings**.

While controlling the mixer audio, you should query the current settings before changing them. Restore them to their original settings as soon as you've finished using them. You should follow this rule unless your application is designed to be a master control for all audio devices in a system.

When you have finished using the driver, you must call the **ctadTerminate** function to perform necessary driver termination activities. Unload the driver as soon as these activities are completed.

You need to include the header file SBKAUX.H for C language or SBKAUX.BI for Microsoft Basic in all your applications that use the AUXDRV functions. All the AUXDRV driver function names are prefixed with **ctad**.

The following pseudo code fragment describes the process for using the AUXDRV driver:

```
// Load AUXDRV driver and assign the entry to CTAuxDrv
CTAuxDrv := sbkLoadDriver("AUXDRV.DRV", 0, lpOrgPtr);

IF driver loaded successful THEN
{
```

```
        // check for correct driver version
        wDrvVersion := ctadGetDrvVer();

        IF wDrvVersion >= 304 hex THEN
        {
                // pass BLASTER environment string to driver
                //
                // 1.  Get BLASTER environment string
                // 2.  Pass to driver by ctadGetEnvSettings()
                ctadGetEnvSettings(BLASTER string);

                ctadInit();

                // strut your AUXDRV function calls here
                .
                .
                .

                // when you have finished using AUXDRV
                ctadTerminate();
        }
        ELSE
        {
                // incorrect driver version
        }


        // unload the loaded driver here by releasing
        // the loaded driver memory location.
}
ELSE
{
        // Load driver error
}
```

The **sbkLoadDriver** function is included in the SBK as a helper function in Microsoft Basic and Turbo Pascal, or as a module file in C language. Refer to the source code of this function on how it works.

Volume Controls

The AUXDRV driver provides the following functions to query and to set the volume for a mixer:

Function	Description
ctadGetVolume	Gets the volume level of the specified mixer audio source.
ctadSetVolume	Sets the volume level of the specified mixer audio source.

The following manifest constants and their meaning has been defined in the include file to facilitate the use of these functions:

Constant	Meaning	Mono/Stereo
MIXERVOL_MASTER	Overall Master	Stereo
MIXERVOL_VOICE	Digitized Sound	Stereo
MIXERVOL_MIDI	MIDI	Stereo
MIXERVOL_CD	CD Audio	Stereo
MIXERVOL_LINE	Line-In	Stereo
MIXERVOL_MIC	Microphone	Mono
MIXERVOL_PCSPK	PC Speaker	Mono

Values for the volume level range from 0 (silence) to 255 (maximum volume). Some sources can support individual volume control on both the left and right channels (stereo). For these sources, the upper 8 bits (high byte) of the volume level value specify the left channel volume and the lower 8 bits (low byte) of the volume level value specify the right channel volume. For the sources that do not support stereo, the lower 8 bits specify the volume level, and the upper 8 bits are ignored.

The following pseudo code fragment describes the use of the **ctadGetVolume** and **ctadSetVolume** functions:

```
// query current master volume; preserve.
wOrgMasVol := ctadGetVolume(MIXERVOL_MASTER);

// now, set master left to 255 and right to 0
wMasLeftVol := 255;
wMasRightVol := 0;

// put left channel volume in upper 8 bits and
//  right channel volume in lower 8 bits
wMasVol := (wMasLeftVol SHL 8) + wMasRightVol;

wSetResult := ctadSetVolume(MIXERVOL_MASTER, wMasVol);
if (wSetResult = 0) THEN
       // set volume successful
ELSE
       // set volume failed

// now, restore the original master volume
ctadSetVolume(MIXERVOL_MASTER, wOrgMasVol);
```

Tone Controls

The AUXDRV driver provides the following functions to query and to set the treble and bass tone level of a mixer:

Function	Description
ctadGetToneLevel	Gets the tone level of the specified mixer audio source.
ctadSetToneLevel	Sets the tone level of the specified mixer audio source.

The following manifest constants and their meaning have been defined in the include file to help you use these functions:

Constant	Meaning
TONE_TREBLE	Treble tone
TONE_BASS	Bass tone

Values for the tone level range from 0 (silence) to 255 (maximum tone). The upper 8 bits of the tone level value specify the left channel tone and the lower 8 bits of the tone level value specify the right channel tone.

The following pseudo code fragment describes the use of the **ctadGetToneLevel** and **ctadSetToneLevel** functions:

```
// query current Bass level; preserve.
wOrgBassLevel := ctadGetToneLevel(TONE_BASS);

// now, set bass left to 255 and right to 0
wBassLeftLevel := 255;
wBassRightLevel := 0;

// put left channel Bass level in upper 8 bits and
//  right channel Bass level in lower 8 bits
wBassLevel := (wBassLeftLevel SHL 8) + wBassRightLevel;

wSetResult := ctadSetToneLevel(TONE_BASS, wBassLevel);
if (wSetResult = 0) THEN
        // set tone successful
ELSE
        // set tone failed

// now, restore the original Bass level
ctadSetToneLevel(TONE_BASS, wOrgBassLevel);
```

Gain Controls

The AUXDRV driver provides the following functions to query and to set the gain level for the input and output of a mixer:

Function	Description
ctadGetMixerGain	Gets the gain level of the mixer input/output mixing path.
ctadSetMixerGain	Sets the gain level of the mixer input/output mixing path.

The following manifest constants and their meaning have been defined in the include file to help you use these functions:

Constant	Meaning
GAIN_IN	Input gain
GAIN_OUT	Output gain

Values for the gain level range from 0 (minimum) to 3 (maximum). Individual gain on both the left and right channels can be controlled. The upper 8 bits of the gain level value specify the left channel gain and the lower 8 bits of the gain level value specify the right channel gain.

The following pseudo code fragment describes the use of the **ctadGetMixerGain** and **ctadSetMixerGain** functions:

```
// query current input gain; preserve.
wOrgInputGain := ctadGetMixerGain(GAIN_IN);

// now, set left input gain to 3 and right to 0
wInputLeftGain := 3;
wInputLeftGain := 0;

// put left channel gain in upper 8 bits and
//  right channel gain in lower 8 bits
wInputGain := (wInputLeftGain SHL 8) + wInputRightGain;

wSetResult := ctadSetMixerGain(GAIN_IN, wInputGain);
if (wSetResult = 0) THEN
        // set gain successful
ELSE
        // set gain failed

// now, restore the original Bass level
ctadSetMixerGain(GAIN_IN, wOrgInputGain);
```

Automatic Gain Control

The AUXDRV driver provides the following functions to query and to set the automatic gain control (AGC) of a mixer:

Function	Description
ctadGetAGC	Gets the AGC status.
ctadSetAGC	Sets the AGC status.

AGC is used to maintain the microphone input signal at a reasonable level. By default, it is ON. You should turn it off if your application does not want the input signal to be adjusted in an application such as Speech Recognition.

Mixer Reset

The AUXDRV driver provides the following function to reset a mixer to its default state.

Function	Description
ctadResetMixer	Resets Mixer to its default state.

The default state varies from mixer to mixer. The mixers default settings are documented on the **Hardware Programming Reference** manual. Refer to the manual for the default settings of each mixer.

Mixing Controls

On Sound Blaster 16, you can mix multiple sources for both recording and playback. The AUXDRV driver provides the following functions to mix the recording and playback sources of a mixer.

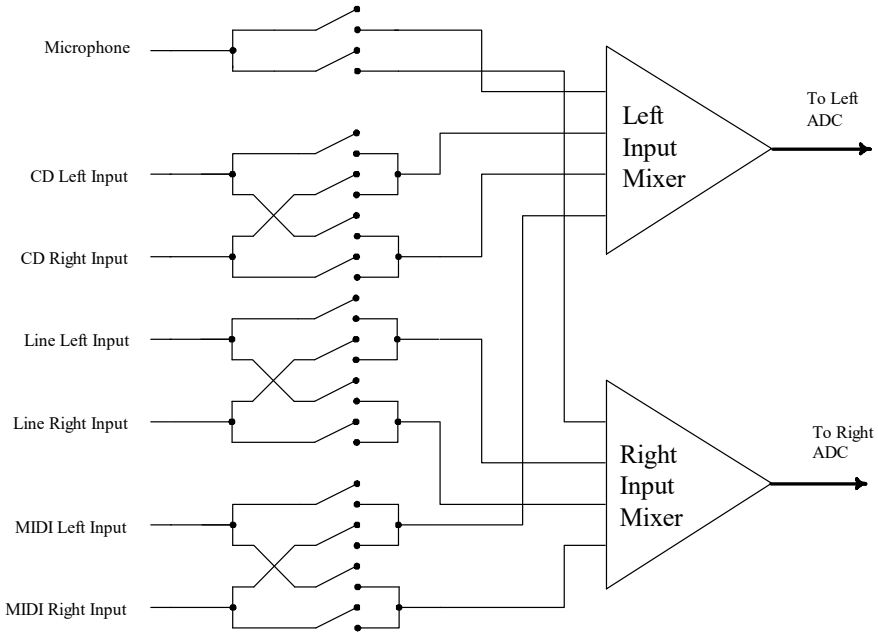
Function	Description
ctadGetMixerSwitch	Gets the recording or playback source(s).
ctadSetMixerSwitch	Sets the recording or playback source(s).

The following constants and their meaning have been defined in the include file to help you use these functions:

Constant	Meaning
MIXERSWI_MIC	Microphone
MIXERSWI_CD_R	CD audio right channel
MIXERSWI_CD_L	CD audio left channel
MIXERSWI_LINE_R	Line-In right channel
MIXERSWI_LINE_L	Line-In left channel
MIXERSWI_MIDI_R	MIDI right channel
MIXERSWI_MIDI_L	MIDI left channel

Input Mixing Control

The following diagram shows the logical block of the Input Mixing Control:



Input Mixing Control Logical Diagram

The input mixing controls the recording sources. The Sound Blaster 16 supports recording from the **Microphone, CD, Line-In** and **MIDI** sources concurrently. You can mix any of these sources in your recording.

The mixer on Sound Blaster 16 allows you to direct the left and/or right channels of the stereo sources to the left and/or right channels of the input mixer. As an example, before making a recording, you can direct the left CD channel to the input mixer's right channel, and the right CD channel to the input mixer's left channel. This way, you get left/right stereo reversal at the hardware level.

You call the **ctadSetMixerSwitch** function to turn on or off an input mixing source. To control multiple sources, you simply bit-or a combination of **MIXERSWI_MIC, MIXERSWI_CD_R, MIXERSWI_CD_L, MIXERSWI_LINE_R, MIXERSWI_LINE_L, MIXERSWI_MIDI_R** or **MIXERSWI_MIDI_L** constants. The upper word of the mixer switch value specify the left input mixer settings and the lower word of the mixer switch value specify the right input mixer settings.

When changing a mixer input switch, you must query the current mixer switch settings using **ctadGetMixerSwitch** and only change the particular switch's bit. You should also restore the original settings as soon as you've finished using it. You should follow this rule unless your application is designed to be a master control for all audio devices in a system.

When recording in mono, the samples will be taken from the left input mixer. So, if a mono recording of a stereo source is desired, the input mixer switches must be manipulated to enable both channels of a stereo source to be mixed together first into the left input mixer, before being sampled.

The following pseudo code fragment describes how to use the **ctadGetMixerSwitch** and **ctadSetMixerSwitch** functions to select only CD input as a recording source for both left and right input mixer settings.

```
// manifest constants
CONST
        IN  = 0;
        OUT = 1;
.
.
.
// query current input mixer settings; preserve.
dwOrgInMixer := ctadGetMixerSwitch(IN);

// now, set only CD input switch on both left and right input mixer
dwLeftInMixer  := MIXERSWI_CD_L (bitwise OR) MIXERSWI_CD_R;
dwRightInMixer := MIXERSWI_CD_L (bitwise OR) MIXERSWI_CD_R;

// put left input mixer settings in upper word and
//  right input mixer setting in lower word
dwInMixer := (dwLeftInMixer SHL 16) + dwRightInMixer;

wSetResult := ctadSetMixerSwitch(IN, dwInMixer);
if (wSetResult = 0) THEN
        // set mixer switch successful
ELSE
```

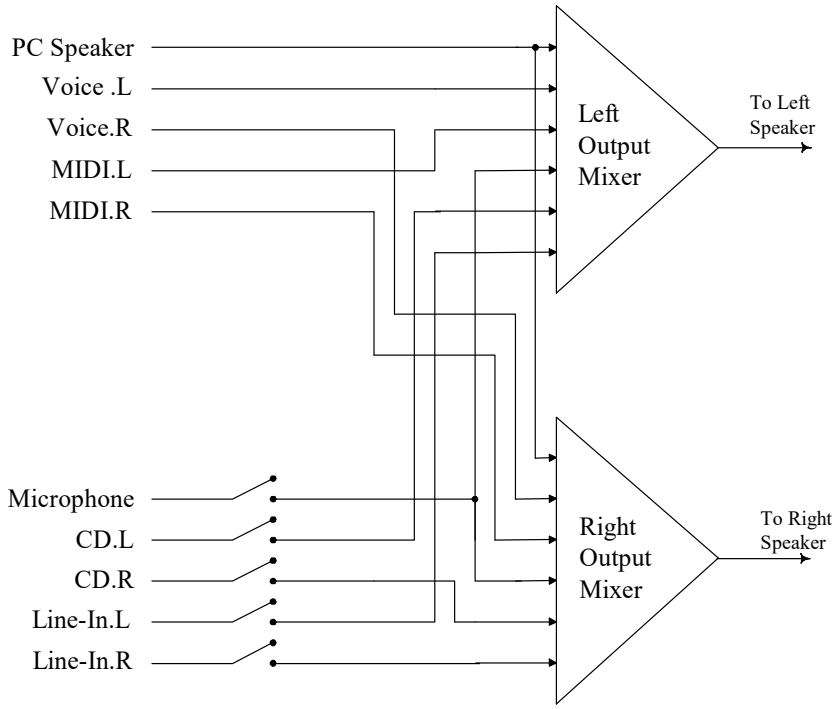
```
        // set mixer switch failed

.
.  // strut in your recording code here
.

// now, restore the original input mixer settings
ctadSetMixerSwitch(IN, wOrgInMixer);
```

Output Mixing Control

The following diagram shows the logical block of the Output Mixing Control:



Output Mixing Control Logical Diagram

The output mixing path takes signals from the **PC Speaker, Voice, MIDI, Microphone, CD or Line-In** sources. On SBPRO, the only way of silencing a source (aside from terminating the source activity) is to turn the source volume down to zero. On Sound Blaster 16, three of the sources, namely **Microphone, CD** or **Line-In**, can be silenced by toggling some mixer switches to cut off these three sources from the output mixing path.

You call the **ctadSetMixerSwitch** function to turn on or off an output mixing source. To control multiple sources, you simply bit-or a combination of **MIXERSWI_MIC, MIXERSWI_CD_R, MIXERSWI_CD_L, MIXERSWI_LINE_R** or **MIXERSWI_LINE_L** constants.

When changing a mixer output switch, you must query the current mixer switch settings using **ctadGetMixerSwitch** and only change the particular switch's bit. You should also restore the original settings as soon as you've finished using it. You should follow this rule unless your application is designed to be a master control for all audio devices in a system.

The following pseudo code fragment describes how to use the **ctadGetMixerSwitch** and **ctadSetMixerSwitch** functions to change the Microphone output switch. First, it turns on the microphone output and later turns it off. Before exit the program, it restores the original output mixer settings.

```
// manifest constants
CONST
        IN  = 0;
        OUT = 1;
.
.
.
// query current output mixer settings; preserve.
dwOrgOutSwitches := ctadGetMixerSwitch(OUT);

// now, set microphone switch on
dwOutSwitches := dwOrgOutSwitches;

// use bit-or operation to turn on microphone bit only
dwOutSwitches := dwOutSwitches (bitwise OR) MIXERSWI_MIC;

wSetResult := ctadSetMixerSwitch(OUT, dwOutSwitches);
if (wSetResult = 0) THEN
        // set mixer switch successful
ELSE
        // set mixer switch failed

.
.
.
// query current output mixer settings
dwOutSwitches := ctadGetMixerSwitch(OUT);

// now, set microphone switch off
// use bit-and operation to turn off microphone bit only
dwSwitchMask  := (bitwise negation of MIXERSWI_MIC);
dwOutSwitches := dwOutSwitches (bitwise AND) dwSwitchMask;

wSetResult := ctadSetMixerSwitch(OUT, dwOutSwitches);
if (wSetResult = 0) THEN
        // set mixer switch successful
ELSE
        // set mixer switch failed
```

```
.
.
.

// now, restore the original output mixer settings
ctadSetMixerSwitch(OUT, wOrgOutSwitches);
```

Fade and Pan Effects

The AUXDRV driver lets you control the fading and panning sound effects.

- **Fading** is a process of gradually increasing (or decreasing) the sound volume until a certain level is reached.
- **Panning** is a process of having the sound move from left to right (or vice versa).

Setting Up Fading or Panning Effects

The AUXDRV driver provides the following functions to let you setup the fading and panning sound effects:

Function	Description
ctadFade	Set up a stereo source for fading.
ctadPan	Set up a stereo source for panning.

You use **ctadFade** function to setup a stereo source for fading or **ctadPan** function for panning. The syntax of the **ctadFade** and **ctadPan** functions are as follow:

ctadFade(*wVolSource*, *wFinalVol*, *wCycleTime*, *wFadeMode*, *wRepeatCount*)

ctadPan(*wVolSource*, *wInitialPos*, *wFinalPos*, *wCycleTime*, *wPanMode*, *wRepeatCount*)

The *wVolSource* parameter is a WORD and specifies a stereo volume source. This is the same volume source on the volume control. See the **Volume Controls** section for the constants.

The *wInitialPos* parameter is a WORD and specifies the starting position of the panning process. It can be any position between the extreme left (position 0) and extreme right (position 255).

The *wFinalPos* parameter is a WORD and specifies the ending position of the panning process. Just as *wInitialPos,* it takes any value between 0 to 255.

The *wFinalVol* parameter is a WORD and specifies the final volume level of the fading process. The high byte of the word denotes the left channel volume level and the low byte denotes the right channel volume level. Volume level can be any value between 0 (minimum) to 255 (maximum). Fading process uses the current volume level of the specified source as the initial fade volume level. Hence, you should setup the initial volume level of the source to be faded before invoking the **ctadFade** function.

The *wCycleTime* parameter is a WORD and specifies the time taken to complete one fade or pan process in the unit of millisecond ranging from 1 to 65535. This is one cycle time when the "yo-yo" effect (see below) is off. If the "yo-yo" effect is on, the total time taken to complete one fade or pan process will double the *wCycleTime*.

The *wFadeMode* or *wPanMode* parameters are WORD size parameters that allow you to turn on the "yo-yo" effect of the fade and pan process respectively. A value of 1 turns on the "yo-yo" effect. A "yo-yo" effect is created when you fade to one sound level and then return to the original sound level. For panning, the "yo-yo" effect is created by panning from left to right and back to the left again (or vice versa). In other words, the "yo-yo" effect can be known as loopback effect.

The *wRepeatCount* parameter is a WORD and specifies the number of cycles a fade or pan process repeats ranging from 1 to 65535.

If a source is already setup for fade or pan, it is not allowed to setup for other effects. Also, no change of volume or pan position is allowed. This takes effect till the fade or pan process is cleared.

While any source is in the process of panning or fading, no other source is allowed to setup for pan or fade before the completion of the control.

Fade and Pan Status Words

While a fade or pan process is active, you can monitor the process status with the help of its status word. The AUXDRV driver provides you with the following functions to set the addresses of the fade and pan process status word:

Function	Description
ctadSetFadeStAddx	Sets the address of the fade status word.
ctadSetPanStAddx	Sets the address of the pan status word.

After setting the address of the status words, you can use them to monitor the fade and pan process. The individual bit settings in the status word show which of the stereo volume sources is currently active in the fade or pan process. The status word is organized in the following manner:

D15 - D5	D4	D3	D2	D1	D0
0	Line-In	CD	MIDI	Voice	Master

where a 1 at a particular bit position denotes an active fade or pan process for that particular volume source.

The AUXDRV driver updates the status words under the following conditions:

1. Resets the status words to zero during initialization.
2. Sets the corresponding bit to 1 when it starts the fade or pan process for a source.
3. Sets the corresponding bit to 0 at the end of the fade or pan process.
4. Keeps the status words unchange when the **ctadPauseCtrl** function is invoked.

Getting the Current Pan Position

While a pan process is active, you can monitor the current pan position using the following function in the AUXDRV driver:

Function	Description
ctadGetPanPosition	Gets the current pan position.

This function returns the pan position if successful. The pan position returned is within the range of *wInitialPos* and *wFinalPos* set in the **ctadPan** function. For instance, if the *wInitialPos* and *wFinalPos* are set to 50 and 200, the pan position returned will be within the range of 50 and 200.

Starting, Pausing, Resuming, and Stopping an Effect

The AUXDRV driver provides the following functions for you to control a fade or pan process:

Function	Description
ctadStartCtrl	Starts the fade and pan processes which have already been set up.
ctadPauseCtrl	Pauses the active fade and pan processes.
ctadStopCtrl	Stops all the active fade and pan processes.
ctadClrSource	Stops an individual effect on an individual volume source.

After setting up the fade and pan processes, you can start them using **ctadStartCtrl** function. When the fade and pan processes are active, you can stop or pause them. You pause the processes with **ctadPauseCtrl** function. To resume the paused processes, use **ctadStartCtrl** function again. The **ctadStartCtrl** function serves two purposes:

1. Starts the fade and pan processes which have already been set up.
2. Resumes the paused processes which were paused with **ctadPauseCtrl** function.

To deactivate stop a fade or pan process, use the **ctadClrSource** function. This function allows you to stop an effect on a volume source. You use the **ctadStopCtrl** function when you want to stop all the fade and pan processes. Once a fade or pan process is stopped, you cannot resume it with **ctadStartCtrl** function.

The following pseudo code fragment illustrates how to initiate a fade and pan process for the digitized sound. Assuming the digitized sound is already setup for playing:

```
// set the status words addresses for fade and pan processes
ctadSetFadeStAddx( (address of)CTFadeStatus );
ctadSetPanStAddx( (address of)CTPanStatus );

// now, preserve current voice volume settings
wOrgVoiceVol := ctadGetVolume(MIXERVOL_VOICE);

// set voice left and right volume to 0
wResult := ctadSetVolume(MIXERVOL, 0);

// setup voice for fading in "yo-yo" mode
//  and repeat for 5 times
wResult := ctadFade(MIXERVOL_VOICE, 0xffff, 2000, 1, 5);
```

```
// also, setup voice for panning in "yo-yo" mode
//  and repeat for 5 times
wResult := ctadPan(MIXERVOL_VOICE, 0, 255, 2000, 1, 5);

// we are ready for fade and pan now....
wResult := ctadStartCtrl();


// wait until the fade or pan processes end
while ( (CTFadeStatus <> 0) OR (CTPanStatus <> 0) )
{
        // stop the fade and pan processes when a key is pressed
        IF KeyPressed  THEN
                wResult := ctadStopCtrl();
}


// now, restore original voice volume settings
wResult := ctadSetVolume(MIXERVOL_VOICE, wOrgVoiceVol);
```

Chapter 6
Creative Multimedia System Driver

This chapter describes how to program the Creative sound, auxiliary and signal processing devices using the Creative Multimedia System (CTMMSYS) driver.

The Creative Multimedia System driver provides the API for digitized sound playback and recording, auxiliary device as well as signal processing device control, by accessing the hardware-dependent device drivers. Users can write programs in high level languages to access the services provided. Currently, the only languages supported are C and C++.

In contrast to the high-level digitized sound and auxiliary drivers (CT-VOICE, CTVDSK, CTWMEN, CTWDSK and AUXDRV), CTMMSYS gives you more control over digitized sound recording and playback, but requires more programming than the high-level drivers. Therefore, if your application requires digitized sound I/O or auxiliary device controls, you should use the high-level drivers. If the high-level drivers do not meet the needs of your application, then use the low-level driver services to manage the device-level drivers yourself.

You do not need to invoke the signal processing device if you are using Creative drivers for digitized sound I/O. The drivers will handle automatically for you. You should invoke the signal processing device only when you are programming Sound Blaster hardware directly and wish to have 16-bit sound data compression support.

This chapter covers the following topics:

- Using the Creative Multimedia System driver
- Querying sound device information
- Opening and closing sound device
- Buffer handling
- Controlling digitized sound I/O
- Monitoring digitized sound I/O
- Querying auxiliary device information
- Opening and closing auxiliary device

- Controlling the auxiliary device
- Querying signal processing device information
- Opening and closing signal processing device
- Downloading code to the signal processing device
- Setting the signal processing device parameter.
- Controlling the signal processing device

Using the Creative Multimedia System Driver

The Creative Multimedia System driver maintains a consistent set of API services throughout the Sound Blaster cards. Only the lowest level device-level driver **CTSOUND** is unique to each Sound Blaster device.

CTMMSYS is loaded through the **CONFIG.SYS** at boot time using the statement:

DEVICE = <*full path to CTMMSYS.SYS*>

All calls to the driver are handled by the entry-point function of type **MMSYSPROC** defined in the **CTMMSYS.H** file.

The Creative Multimedia System driver CTMMSYS.SYS is a character-mode DOS device driver, with device name "CTMMSYS$", that provides the high-level control interface to the Sound Blaster card. However, unlike a normal DOS device driver, the programmer does not communicate with CTMMSYS via IOCTL commands. Instead, like HIMEM.SYS (Microsoft's extended memory driver), the services are invoked by making far calls to the driver's entry-point. Function arguments are passed through the user stack. The header files **CTMMSYS.H** and **CTSTDDEF.H** provides C and C++ programmers with the necessary **typedefs** to make accessing the driver easy and safe.

All calls to the driver is handled by the entry-point function of type **MMSYSPROC** defined in the **CTMMSYS.H** file. A detailed description of the various parameters in the **MMSYSPROC** function is given in the Library Reference Manual.

To begin, the driver has to be opened using the DOS file open service. This function is commonly available as a run-time library function in most compiler packages.

The entry point of the driver is then obtained using the DOS Device IOCTL Read function (DOS Int 21H Function 4402H) with CX = number of bytes to read = 4. Before invoking DOS Device IOCTL Read function, the **MMSYSPROC** variable must be assigned to **CTMM_ENTRYPOINT_KEY** to inform the driver to return the entry-point.

The following C code fragment illustrates how to set up the call to the driver:

```
// declare the entry point function
MMSYSPROC      MmSysProc = NULL;

WORD           wFileHandle;

// read the DOS device driver file CTMMSYS.SYS to get file handle
wFileHandle = dosOpenFile("CTMMSYS$");

// call dos IOCTL read device to get entry point
MmSysProc = (MMSYSPROC)CTMM_ENTRYPOINT_KEY;
dosIOCTLRead(wFileHandle, &MmSysProc, sizeof(MMSYSPROC));

// close file
dosFileClose(wFileHandle);
```

In order to keep the salient points of gaining access to the driver in clear view, the above example deliberately avoided checking for error return values. Of course, in an actual program, the user will have to check the return status.

The driver is invoked with a far call to its entry-point. The function parameters are passed on the user stack in Pascal calling convention, i.e. pushed from left to right.

The general sequence of programming the CTMMSYS driver for digitized sound I/O is as follows:

1. Get the driver's entry point.
2. Query the number of devices available.
3. Query the configuration and capabilities of the desired device.
4. Specify the digitized sound format and transfer buffer.
5. Set up the callback function.
6. Open the sound device and obtain the device handle.
7. Add data buffers to the buffer queue.
8. Start digitized sound I/O.

.... perform other desired actions

9. Reset digitized sound I/O.
10. Close the sound device.

The general sequence of programming the CTMMSYS driver for auxiliary control is as follows:

1. Get the driver's entry point.
2. Query the number of devices available.
3. Query the configuration and capabilities of the desired device.
4. Open the auxiliary device and obtain the device handle.
5. Get/set various auxiliary device settings.
6. Close the auxiliary device.

The general sequence of programming the CTMMSYS driver for signal processing control is as follows:

1. Get the driver's entry point.
2. Query the number of devices available.
3. Query the configuration and capabilities of the desired device.
4. Open the signal processing device and obtain the device handle.
5. Download signal processing code and set appropriate settings.
6. Start signal processing.

.... perform other desired actions

7. Stop signal processing.
8. Close the signal processing device.

The following pages in this chapter describe each of these steps.

Querying Sound Device Information

CTMMSYS provides the following query messages for the sound device:

Message	Description
SxDM_QUERY_NumDevs[4]	Query the number of input or output devices installed.
SxDM_QUERY_Capabilities	Query the capabilities of the device.
SxDM_QUERY_SamplingRange	Query the maximum and minimum sampling rates supported.
SxDM_QUERY_TransferBuffer	Get information on the transfer (DMA) buffer needed.
SxDM_CONFIGURATION_Query	Query the configuration of the device.

The following structures, and the corresponding messages that use them, have been defined in the CTMMSYS.H header file:

Structure	Used with message
SOUNDCAPS	**SxDM_QUERY_Capabilities**
SOUNDSAMPLINGRANGE	**SxDM_QUERY_SamplingRange**
SOUNDQYXFERBUF	**SxDM_QUERY_TransferBuffer**
DEVCONFIG	**SxDM_CONFIGURATION_Query**

It is necessary that the user query the driver on the capabilities of the device before performing digitized sound I/O.

A more detailed explanation of the various function parameters and structure elements is found in the Library Reference Manual.

Note that the query functions are called with respect to either input (recording) or output (playback). The information returned by the driver is therefore true only for the mode the user has specified. For example, in querying the sampling range, the maximum supported sampling rate may be different for recording and playback.

[4] All occurrences of 'x' in SxDM_ function names in this document are to be replaced by either 'I' or 'O', referring to 'Input' and 'Output' respectively.

Query Number of Devices

The user must first query the number of devices available before any digitized sound I/O is performed. There is a twofold purpose in doing this. It detects the presence of the sound device/driver and the returned value is used to determine the value of the Device ID for other API calls. The Device ID is between zero to one less than the number of devices. If no devices are installed, the returned value is zero.

Query the Configuration

This is for the user to determine the configuration of the device. The driver will return in the DEVCONFIG.szzConfiguration structure member a far pointer to an array of ASCIIZ strings. This array may consist of more than one environment string, concatenated together. The array is terminated with a null string, i.e. a single null character. Minimally, the array will look like:

'BLASTER=A:220 I:5 D:0 H:5',0,0

Note that if there are no devices installed, the call with message **SxDM_CONFIGURATION_Query** will be return **MMSTATUS_NOT_ENABLED**. In this case, the members of the DEVCONFIG structure passed to the driver are not to be used.

Query the Capabilities

The message **SxDM_QUERY_Capabilities** returns the driver version, card type, card name and the number of channels (mono or stereo) supported by the device.

The user has to provide a structure of type SOUNDCAPS with the Device ID determined from an earlier **SxDM_QUERY_NumDevs** call. The information returned is stored in the structure elements.

This call should be made before attempting any digitized sound I/O because not all features of the API are supported by all the sound devices.

Query the Sampling Range

To find out the minimum and maximum sampling rates of the device, the user has to specify the Device ID and whether it's mono or stereo (they can be different) in a structure of type SOUNDSAMPLINGRANGE. The minimum and maximum

sampling rates supported will be stored in the two structure elements **dwMaxSamplesPerSec** and **dwMinSamplesPerSec**.

Query the Transfer Buffer Requirements

Before initiating digitized sound I/O, the user has to allocate a memory block for the driver to use as a transfer buffer (DMA buffer). The requirements of this buffer can be obtained by invoking **SxDM_QUERY_TransferBuffer**. The information returned covers:

- type of memory (conventional or extended)
- block alignment
- minimum and maximum size of the buffer
- granularity (increment step size, e.g. 2KB)
- whether it cannot cross 64KB or 128KB page boundary

All the above information is stored in a structure of type SOUNDQYXFERBUF which is supplied by the user.

If the returned values of **dwcbMinSize** and **dwcbMaxSize** both equal zero, there is no need to specify a transfer buffer. There might be certain devices in the future which do not make use of DMA transfer, thereby negating the need for a transfer buffer.

The following C code fragment illustrates a typical query transfer buffer sequence:

```
// declare variable
SOUNDQYXFERBUF query;

// initialize device ID
query.wDeviceID = 0;

// call driver
MmSysProc(PARAM_UNUSED, MMDEVICE_SOUNDIN, SIDM_QUERY_TransferBuffer,
               (DWORD)(LPSOUNDQYXFERBUF)&query, PARAM_UNUSED);

// Allocate memory based on information returned.
// Ensure that restrictions are complied with.
```

Opening and Closing Sound Devices

CTMMSYS provides the following messages to open and close the sound device:

Message	Description
SxDM_OPEN	Opens the device for digitized sound I/O.
SxDM_CLOSE	Frees the opened device for use by another application.

The following structures are used with the **SxDM_OPEN** message:

Structure	Used for
SOUNDFORMAT	Specifying the sound data format.
SOUNDXFERBUFDESC	Specifying the address, type and size of the transfer buffer.
SOUNDOPEN	Encapsulating the above structures and the callback function.

Opening a sound device is a necessary step before any playback or recording can be done. The user has to set up the transfer buffer, sound data format and the callback function for the driver.

This call to the driver will also return a device handle **hDev** which is required for controlling and monitoring the digitized sound I/O.

The C code fragment below illustrates the procedure to open a sound device:

```
// declare variables
SOUNDOPEN             SoundOpen;
SOUNDFORMAT           SoundFormat;
SOUNDXFERBUFDESC      XferBuf;

// initialize values
SoundOpen.wDeviceID   = 0;
SoundOpen.dwFlags     = 0;

// initialize SoundFormat
...
SoundOpen.lpFormat = &SoundFormat;

// initialize XferBuf
...
SoundOpen.lpXferBufDesc = &XferBuf;
```

```
// assign the callback function, e.g. MyCallback()
...
SoundOpen.Callback = MyCallback;
SoundOpen.dwCallbackData = myCallbackData;

// call the driver
MmSysProc(PARAM_UNUSED, MMDEVICE_SOUNDOUT, SODM_OPEN,
               (DWORD)(LPSOUNDOPEN)&SoundOpen, PARAM_UNUSED);
```

The **SxDM_OPEN** function call may also be used to check if the desired sound format is supported by the hardware, by setting the **dwFlags** parameter in the SOUNDOPEN structure to **SOUNDOPEN_QUERYFORMAT**. It is recommended that this step be done before the actual device open. Please refer to "Determining whether a Sound Format is Supported" later in this section.

Specifying the Sound Format

The driver is normally ignorant of the digitized sound file type (.voc or .wav format). It simply plays 8 or 16 bit data in PCM format. However, some cards support certain compressed data types, such as Creative ADPCM, CCITT A-Law and CCITT μ-Law format. The driver recognizes this and will program the card accordingly. The user needs to specify the compressed data format in the **wFormatFamily** and **wFormatTag** members of the SOUNDFORMAT structure.

For example, if the user has a .wav type file in Creative ADPCM format :

```
SoundFormat.wFormatFamily = SOUNDFORMAT_FAMILY_WAVE
SoundFormat.wFormatTag    = WAVE_FORMAT_CREATIVE_ADPCM
```

where **SOUNDFORMAT_FAMILY_WAVE** and
WAVE_FORMAT_CREATIVE_ADPCM

are constants defined in CTMMSYS.H.

Once a sound device is opened with a particular sound format, subsequent digitized sound I/O is carried out using that format. If the user intends to change the format, for example from mono to stereo, the sound device has to be closed and reopened with the new format. This is required for playback of files where data of multiple formats is contained in a single file.

Specifying the Transfer Buffer

The user may have to allocate a block of memory for the driver to use as a transfer buffer to do auto-initialize DMA transfer. The user is expected to have found at the

requirements of the transfer buffer by using the **SxDM_QUERY_TransferBuffer** call.

The transfer buffer is specified through a structure of type SOUNDXFERBUFDESC, with information about the address, type and size of the buffer.

The following C code fragment illustrates how this can be done.

```
// declare variable
SOUNDXFERBUFDESC      XferBuf;

// conventional memory
Xfer.Buffer.wType = MEMORYDESC_MEM;

// far pointer to transfer buffer
Xfer.Buffer.u.lpMem = lpXfer;

// specify size
XferBuf.dwcbBufferSize = dwcbSize;
```

If a transfer buffer is not needed as it is in some devices, set the LPSOUNDXFERBUFDESC pointer to NULL.

This transfer buffer is only valid for the lifetime of the opened I/O handle. Once the call **SxDM_CLOSE** is made, the transfer buffer is considered to be returned to the user. Therefore the user must re-specify the transfer buffer for a fresh round of I/O.

The Callback Function

The callback function is another essential part of the I/O process. The heart of the I/O process involves *adding* data buffers to the queue maintained by the driver. The driver will then process each buffer in sequence.

When the driver has finished processing a buffer, the buffer will be *returned* to the user via this callback function. The user may then choose to re-use the same buffer structure. The callback function is also the place where the user can add new buffers to the queue.

The **SOUNDCALLBACK** typedef in CTMMSYS.H indicates the parameters necessary for the callback function. The relevant parameters are:

- the device handle to the current I/O process
- a returned message value, indicating whether it's an input or output buffer that's being returned
- the callback data specified in the **Sx**DM_OPEN call

- a pointer to the returned buffer

There are some important points to note for the user callback function such as DOS re-entrancy and so on. Refer to the chapter "Programming Overview" for the details.

The following C code fragment illustrates how to define the callback function:

```
void
FAR PASCAL Callback(HMMDEVICE hDev,
                    WORD      wMsg,
                    DWORD     dwCallbackData,
                    DWORD     dwParam1,
                    DWORD     dwParam2)
{
        // declare a structure and point to returned structure
        LPSOUNDBUFFER lpRetStruc = (LPSOUNDBUFFER)dwParam1;

        if (wMsg == SODM_BUFFERDONE)    // playback buffer emptied
        {
                // add more buffers to queue
                ...
        }
        else if (wMsg == SIDM_BUFFERDONE)  // recording buffer filled
        {
                // add more buffers to queue
        }
}
```

Determining whether a Sound Format is Supported

To find out if a particular sound format is supported, the user has to set the SOUNDOPEN structure member **dwFlags** to **SOUNDOPEN_QUERYFORMAT**. The user has to enter the desired format information into the structure SOUNDFORMAT before invoking this call. In this case, the callback function is not needed.

For example, if the user wants to know whether the sound device can play CD-quality (stereo 44.1kHz, 16 bit PCM) digitized data, the user may program in this way:

```
// declare variables
SOUNDFORMAT    SoundFormat;
SOUNDOPEN      SoundOpen;
MMSTATUS               mmstat;

// specify the format
SoundFormat.wFormatTag      = WAVE_FORMAT_PCM;
SoundFormat.wFormatFamily   = SOUNDFORMAT_FAMILY_WAVE;
SoundFormat.wChannels       = 2;             // stereo
SoundFormat.dwSamplesPerSec = 44100;
SoundFormat.wBitsPerSample  = 16;            // 16 bit data
SoundFormat.wBlockAlign       = min(1, 2*16)/8);

// specify open to query format
```

```
SoundOpen.wDeviceID = 0;
SoundOpen.dwFlags   = SOUNDOPEN_QUERYFORMAT;
SoundOpen.lpFormat  = &SoundFormat;

// call the driver
mmstat = MmSysProc(PARAM_UNUSED, MMDEVICE_SOUNDOUT, SODM_OPEN,
                   (DWORD)(LPSOUNDOPEN)&SoundOpen, PARAM_UNUSED);

// check return value
if (mmstat == MMSTATUS_BAD_FORMAT)
{
...
}
```

Closing the Sound Device

When the I/O process is completed or when there is a need to change the sound format, the user has to call the driver with **SxDM_CLOSE**. The driver will free all system resources used for the I/O process and restore the previous system states.

The user must not make this call midway through a playback or recording, else the driver will return the error code **MMSTATUS_STILL_ACTIVE**. All I/O processes have to be stopped using the call **SxDM_STATE_SET : SOUNDSTATE_RESET** command. Please refer to the part on **Controlling Digitized Sound I/O** in this manual.

The user should close the device handle whenever the I/O process is completed or is paused indefinitely, so that other applications may also have access to the sound device.

Buffer Handling

CTMMSYS provides the following messages to add data buffers and to query the status of the buffer queue:

Message	Description
SxDM_BUFFERQUEUE_Add	Add a new data buffer to the buffer queue for I/O.
SxDM_BUFFERQUEUE_Query	Query the status of the buffer queue.

The following structures have been defined in CTMMSYS.H:

Structure	Used for

SOUNDBUFFER	Specifying information on the buffer to be added.
SOUNDBUFQ	Returning information on the status of the buffer queue.

Adding a Buffer to the Queue

The process of adding buffers to the queue is central to the entire I/O process. The user has to specify the memory locations where the playback data exists, or where memory is available for storing the recorded data.

Adding buffers is done with the message **SxDM_BUFFERQUEUE_Add** and the structure of type SOUNDBUFFER. A detailed description is given in the Library Reference Manual.

Note that this structure is the same one returned by the driver in the callback function to indicate the I/O process for the block is completed.

The structure element **dwUserData** is available to the user for storing any information useful to him.

For example, the user may want to place an index number into the **dwUserData** to assist in keeping track of the blocks that are added. When the driver makes a callback, the user may read this value to determine which block is returned.

The user may add as many blocks to the queue as there is memory available. If the user has multiple blocks of digitized sound data of the same format, at least 2 blocks should be in the queue, otherwise the I/O process will pause while the user adds another buffer. This is applicable to sound files which contains multiple block of data preceded by a block header, such as the Creative VOC file where the data does not exist in one contiguous block.

The following C code fragment illustrates the procedure for adding a buffer:

```
// declare a structure for adding buffer
SOUNDBUFFER    SoundBuf;

// initialize structure elements
SoundBuf.dwFlags         = 0;
SoundBuf.Buffer.wType    = MEMORY_MEM;        // conventional mem
SoundBuf.Buffer.u.lpMem  = lpData;            // pointer to data
SoundBuf.dwcbBufferSize  = dwcbBlkSize;       // block size
SoundBuf.dwcbUserData    = wIndex;            // buffer index

// call the driver
MmSysProc(hDev, MMDEVICE_SOUNDOUT, SODM_BUFFERQUEUE_Add,
               (DWORD)(LPSOUNDBUFFER)&SoundBuf, PARAM_UNUSED);
```

Querying the Status of the Buffer Queue

The **SxDM_BUFFERQUEUE_Query** message is provided to allow the user to find out the status of the current data buffer queue. The driver will return two pieces of information.

One indicates whether the driver is out of buffers. The other indicates how many bytes are left to be transferred.

The user has to supply a structure of type SOUNDBUFQ which contains the two variables mentioned earlier.

Controlling Digitized Sound I/O

CTMMSYS provides the following message to control the digitized sound I/O process:

Message	Description
SxDM_STATE_Set	Set the state of the digitized sound I/O.

This message together with an additional parameter assigned to *dwParam1*, is used to start, pause or stop the digitized sound I/O process. The three constant defined in CTMMSYS.H for this purpose are:

Constants	Used for
SOUNDSTATE_START	initiating the digitized sound I/O
SOUNDSTATE_STOP	pausing the digitized sound I/O
SOUNDSTATE_RESET	stopping and resetting the digitized sound I/O

The difference between 'stop' and 'reset' actions is that 'reset' will cause all the buffers added to the queue to be returned to the user (via the callback function) whereas 'stop' just pause the I/O process. To resume the process, the user simply has to issue a start' command again.

If any of the actions requested is not applicable, the driver will return the appropriate error codes. For example, if the sound state is already 'idle', a 'stop' command will cause return an error code of **MMSTATUS_REDUNDANT_ACTION**.

The correct device handle must be specified in making this call.

The following C code statement shows an example of a 'reset' call:

```
MmSysProc(hDev, MMDEVICE_SOUNDIN, SIDM_STATE_Set,
               SOUNDSTATE_RESET, PARAM_UNUSED);
```

Monitoring Digitized Sound I/O

CTMMSYS provides the following messages to monitor the state of the digitized sound I/O process.

Message	Description
SxDM_STATE_Query	Query the state of the I/O process.
SxDM_POSITION_Query	Query the position of the I/O transfer.

The structure below is been defined in CTMMSYS.H:

Structure	Used for
MMTIME	Returning information on the current position of transfer.

Monitoring the Position of the Transfer Process

The message **SxDM_POSITION_Query** returns either the number of bytes transferred, the number of samples transferred or the number of milliseconds elapsed. The type (whichever of the three) desired has to be specified in the **wType** member in a structure of type **MMTIME**.

Note that not all the three types may be supported. If the user specifies on non-supported type, the driver will return position information in one of the supported types, and set the **wType** accordingly. The position is set to zero when the device is opened or reset.

The following C code fragment illustrates a position query call:

```
// declare the structure
MMTIME  mmtime;

// specify the return type desired
mmtime.wType = MMTIME_BYTES;

// call the driver
MmSysProc(hDev, MMDEVICE_SOUNDIN, SIDM_POSITION_Query,
               (DWORD)(LPMMTIME)&mmtime, PARAM_UNUSED)
```

Monitoring the Status of the I/O Process

The message **SxDM_STATE_Query** returns the status of the I/O process through a variable of type **SOUNDSTATE**. The values returned by the driver will indicate whether the sound I/O state is idle, active or paused.

Please refer to the Library Reference Manual for more details.

The following C code fragment illustrates the use of this message:

```
// declare variable
SOUNDSTATE     Soundstate;

// call to driver for I/O device state
MmsysProc(hDev, MMDEVICE_SOUNDIN, SIDM_STATE_Query,
               (DWORD)(LPSOUNDSTATE)&Soundstate, PARAM_UNUSED);

// check the return value and perform some action
switch (Soundstate)
{
        case   SOUNDSTATE_IDLE:
        //... add buffers
        //... restart I/O

        case   SOUNDSTATE_ACTIVE:
        //... wait until sound state idle

        case   SOUNDSTATE_PAUSED:
        //... restart I/O process
}
```

Querying Auxiliary Device Information

CTMMSYS provides the following query messages for the auxiliary device:

Message	Description
AUXDM_QUERY_NumDevs	Get the number of auxiliary devices installed.
AUXDM_QUERY_Capabilities	Query the capabilities of the auxiliary device.
AUXDM_CONFIGURATION_Query	Query the configuration of the device.

The following structures and the corresponding messages that use them, have been defined in CTMMSYS.H:

Structure	Used with message
AUXCAPS	AUXDM_QUERY_Capabilities
DEVCONFIG	AUXDM_CONFIGURATION_Query

It is necessary that the user query the driver on the capabilities of the device before performing any controlling.

Query Number of Devices

The user must first query the number of devices available before any auxiliary control is performed. There is a twofold purpose in doing this. It detects the presence of the auxiliary device and the returned value is used to determine the value of the Device ID for other API calls. The Device ID is between zero to one less than the number of devices. If no devices are installed, the returned value is zero.

Query the Configuration

This is for the user to determine the configuration of the device. The driver will return in the DEVCONFIG.szzConfiguration structure member a far pointer to an array of ASCIIZ strings. This array may consist of more than one environment string, concatenated together. The array is terminated with a null string, i.e. a single null character. Minimally, the array will look like:

'BLASTER=A:220 I:5 D:0 H:5',0,0

Note that if there are no devices installed, the call with message **AUXDM_CONFIGURATION_Query** will be return **MMSTATUS_NOT_ENABLED**. In this case, the members of the DEVCONFIG structure passed to the driver are not to be used.

Query the Capabilities

Every application should determine the general capabilities of the auxiliary device before using it. The AUXCAPS structure is used to obtain such information using the **AUXDM_QUERY_Capabilities** message. The capabilities are returned in the form of bit-or'ed values in the members **dwSupport** and **dwSource**. A bit that is set to one indicates that the corresponding option is available or supported.

dwSupport details the type of controls supported by the auxiliary device. It is set according to the following bit constants:

Constant	Description
AUXCAPS_SUPPORT_VOLUME	Support software volume control
AUXCAPS_SUPPORT_MIXING	Support mixing of sources
AUXCAPS_SUPPORT_FILTER	Support filter
AUXCAPS_SUPPORT_TONE	Support treble/bass tone control
AUXCAPS_SUPPORT_GAIN	Support gain control
AUXCAPS_SUPPORT_AGC	Support automatic gain control

dwSource details the sound sources are available. It is set according to the following bit constants:

Constant	Description
AUX_SOURCE_MASTER	Master output volume
AUX_SOURCE_VOICE	Digitized Sound volume
AUX_SOURCE_MIDI	MIDI volume
AUX_SOURCE_CD	CD volume
AUX_SOURCE_LINEIN	Line-in volume
AUX_SOURCE_MIC	Microphone volume
AUX_SOURCE_PCSPEAKER	PC Speaker volume

Other useful information included in the AUXCAPS data structure includes the driver's version number and the product code. This could be useful for future driver revisions and product upgrades.

Opening and Closing Auxiliary Devices

CTMMSYS provides the following messages to open and close the auxiliary device:

Message	Description
AUXDM_OPEN	Opens the device.
AUXDM_CLOSE	Frees the device for use by another application.

The following structure is used with the **AUXDM_OPEN** message:

Structure	Used for
AUXOPEN	Returning the device handle.

Opening the auxiliary device to obtain the device access handle **hDev** is a mandatory step before accessing any of the auxiliary controls.

After an application has obtained a device handle, it will have exclusive access to the auxiliary device. This is true provided that all applications use CTMMSYS. Any attempt by another application to obtain an access handle for the same auxiliary device will fail. Therefore, unless the application that has the handle releases it by closing the auxiliary device, no other application can access the auxiliary device. If an application terminates without releasing the handle, it will be lost. The only way to regain access to the auxiliary device is to reboot the computer system.

Controlling the Auxiliary Device

There are several controls available for each auxiliary device. The availability of a particular control can be determined through querying the general capabilities of the device. For each control, there is a group of three CTMMSYS messages that are associated with it. They are

- **AUXDM_*CONTROL*_QueryCaps**
- **AUXDM_*CONTROL*_Get**
- **AUXDM_*CONTROL*_Set**

where **AUXDM_*CONTROL*** can be any of the following software controls:

AUXDM_*CONTROL*	Software audio control
AUXDM_VOLUME	volume control
AUXDM_MIXING	sources mixing control
AUXDM_FILTER	filter control
AUXDM_TONE	Treble/bass tone control
AUXDM_GAIN	gain control
AUXDM_AGC	Automatic gain control

AUXDM_*CONTROL*_QueryCaps

Notice that an **AUXDM_*CONTROL*_QueryCaps** message is provided with each control. This is to allow applications to further determine whether a specific feature of a control is available. The following data structures are used to query the control features. They are defined in the header file CTMMSYS.H:

Data structure	Software audio control
AUXVOLUMECAPS	Input/output volume control
AUXMIXINGCAPS	Input/output sources mixing control
AUXFILTERCAPS	Input/output filter control
AUXTONECAPS	Treble/bass tone control
AUXGAINCAPS	Input/output gain control
AUXAGCCAPS	Automatic gain control

AUXDM_*CONTROL*_Get/Set

A **AUXDM_*CONTROL*_Get/Set** message retrieves or sets a particular audio control setting. The AUXSETTINGS data structure is used. AUXSETTINGS.dwItem will always refer to the audio sources, e.g. **AUX_SOURCE_MIDI** or **AUX_MIXING_INPUT**. AUXSETTINGS.dwFlags is usually set to indicate left or right audio sources, e.g. **AUX_MIXING_LEFT**. It must be set to 0 when unused. AUXSETTINGS.dwLeft and AUXSETTINGS.dwRight refers to the left and right channels of a stereo audio source. If it is a mono audio source, however, only the AUXSETTINGS.dwLeft is used; AUXSETTINGS.dwRight is set to 0.

If a control is not available, the **MMSTATUS_UNSUPPORTED_MSG** error code will be returned. If, however, a particular feature of a control is not supported, the **MMSTATUS_BAD_PARAMETER** error code will be returned instead.

Querying Signal Processing Device Information

CTMMSYS provides the following query messages for the signal processing device:

Message	Description
CSPDM_QUERY_NumDevs	Get the number of signal processing devices installed.
CSPDM_QUERY_Capabilities	Query the capabilities of the signal processing device.
CSPDM_CONFIGURATION_Query	Query the configuration of the device.

The following structures and the corresponding messages that use them, have been defined in CTMMSYS.H:

Structure	Used with message
CSPCAPS	**CSPDM_QUERY_Capabilities**
DEVCONFIG	**CSPDM_CONFIGURATION_Query**

It is necessary that the user query the driver on the capabilities of the device before performing any controlling.

Query Number of Devices

The user must first query the number of devices available before any signal processing control is performed. There is a twofold purpose in doing this. It detects the presence of the signal processing device and the returned value is used to determine the value of the Device ID for other API calls. The Device ID is between zero to one less than the number of devices. If no devices are installed, the returned value is zero.

Query the Configuration

This is for the user to determine the configuration of the device. The driver will return in the DEVCONFIG.*szzConfiguration* structure member a far pointer to an array of ASCIIZ strings. This array may consist of more than one environment string, concatenated together. The array is terminated with a null string, i.e. a single null character. Minimally, the array will look like:

'BLASTER=A:220',0,0

Note that if there are no devices installed, the call with message **CSPDM_CONFIGURATION_Query** will be return **MMSTATUS_NOT_ENABLED**. In this case, the members of the DEVCONFIG structure passed to the driver are not to be used.

Query the Capabilities

Every application should determine the general capabilities of the signal processing device before using it. The CSPCAPS structure is used to obtain such information using the **CSPDM_QUERY_Capabilities** message. Information returned includes the driver's version number and the product code. This could be useful for future driver revisions and product upgrades.

Opening and Closing Signal Processing Devices

CTMMSYS provides the following messages to open and close the signal processing device:

Message	Description
CSPDM_OPEN	Opens the device.
CSPDM_CLOSE	Frees the device for use by another application.

The following structure is used with the **CSPDM_OPEN** message:

Structure	Used for
CSPOPEN	Returning the device handle.

Opening the signal processing device to obtain the device access handle **hDev** is a mandatory step before accessing any of the signal processing controls.

After an application has obtained a device handle, it will have exclusive access to the signal processing device. This is true provided that all applications use CTMMSYS. Any attempt by another application to obtain an access handle for the same signal processing device will fail. Therefore, unless the application that has the handle releases it by closing the signal processing device, no other application can access the signal processing device. If an application terminates without releasing the handle, it

will be lost. The only way to regain access to the signal processing device is to reboot the computer system.

Downloading Code to the Signal Processing Device

CTMMSYS provides the following message to download a signal processing code file:

Message	Description
CSPDM_CODE_Download	Download a signal processing code file

The following structure is used with the **CSPDM_CODE_Download** message:

Structure	Used for
CSPCODEDOWNLOAD	Specifying the code information

The correct code file must be downloaded to the signal processing device for it to work. Sound Blaster 16 Advanced Signal Processing package ships with compression/decompression code files for Creative ADPCM, CCITT A-Law and CCITT μ-Law.

To download a code file, you must load it into memory; assign the address of that memory to CSPCODEDOWNLOAD.*lpCode*, and the length of the code to CSPCODEDOWNLOAD.*dwcbCode*. The CSPCODEDOWNLOAD.*Flags* should be assigned to **CSPCODEDOWNLOAD_INITCODE** for compression/decompression code files.

Controlling the Signal processing Device

CTMMSYS provides the following message to control the digitized sound I/O process:

Message	Description
CSPDM_STATE_Set	Set the state of the signal processing device.

This message together with an additional parameter assigned to *dwParam1*, is used to activate, deactivate or re-initialize the signal processing device. The three constant defined in CTMMSYS.H for this purpose are:

Constants	Used for
CSPSTATE_ACTIVE	set signal processing device to active mode
CSPSTATE_STANDBY	set signal processing device to standby mode
CSPSTATE_INACTIVE	set signal processing device to inactive mode

After downloaded the signal processing code and set up the appropriate parameters, you can now activate the signal processing device. Send **CSPSTATE_ACTIVE** to start a signal processing. Use **CSPSTATE_STANDBY** to pause the signal processing and **CSPSTATE_INACTIVE** to stop the signal processing. **CSPSTATE_ACTIVE** can also be used to resume a paused signal processing.

Chapter 7
MIDI Driver

This chapter presents a detailed description of the Creative MIDI driver architecture, as well as its entire API functions and how an application can make use of them to gain control over MIDI playback and recording.

This chapter assumes you have some basic knowledge of MIDI, and therefore detailed information on MIDI will not be discussed here. If you need additional information on this subject, you are advised to refer to the **Relevant Information** appendix**.**

The following topics will be covered in this chapter:

- Functionality
- About Creative MIDI Driver
- Selecting MIDI devices
- Selecting MIDI mapper types
- Using Creative MIDI driver
- Playback MIDI events
- Recording MIDI events
- Playing note-by-note MIDI events
- Restrictions of MIDI driver

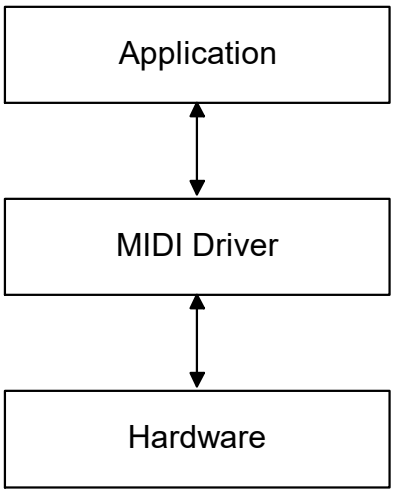
Functionality

Each Sound Blaster card has its own specific MIDI driver. To achieve full portability and device-independence across all Sound Blaster cards, the design of the driver must maintain the same set of API across all the cards. Henceforth, application developed for one sound card can be ported to another sound card easily as long as the respective sound card's MIDI driver is used.

MIDI files supported by the driver are strictly in Standard MIDI File (.MID) format. They are relatively small as compared to voice (.VOC) and waveform (.WAV) files, therefore only conventional memory mode is supported by the driver for playback as well as for recording.

About Creative MIDI Driver

The following diagram shows the relationship between the application and the driver.



Relationship between the application and the Creative MIDI driver

The MIDI driver serves as a middle layer between the application and the hardware. It controls the hardware so as to achieve MIDI events playback and recording functions based on the API commands sent by the application. This arrangement simplifies the application programming task as at the application level, little or no

knowledge of the hardware on the Sound Blaster cards is needed, and therefore device-independent goal can be achieved.

The driver consists of the following functional blocks:

- File parser
- Recording
- Note-by-note events handling
- Hardware handling

File parser

After the synthesizer type and the base I/O address have been made known to the driver, MIDI events playback can be initiated by the application by calling **ctmdPrepareMidiStart,** followed by **ctmdPlayMidiMusic**. The file parser section interprets the MIDI file header to gather MIDI sequencing information pertaining to the MIDI file. Information such as music tempo, resolutions (ticks per quarter note), number of tracks etc. ought to be known to the driver or else no MIDI events can be processed. Subsequently, the driver activates the timing function at the rate relevant to the music tempo and resolution of the MIDI file. At regular intervals, the driver will parse every track within the file to determine if any of them are required for MIDI events processing. This is according to the respective elapsed delta time for individual track.

Recording

The recording module is responsible for monitoring the incoming MIDI code. Again, the base I/O address has to be made known to the driver before it can initialize the MIDI port to UART mode. UART mode is used for MIDI recording as it is possible to achieve playback of MIDI music and recording from external MIDI device through the same MIDI port simultaneously if the synthesizer type for playback selected by the user is "external". When a MIDI code arrives at the MIDI port, an interrupt signal is sent from the Sound Blaster card to the PC, and the interrupt service routine in the driver reads the MIDI code, and determines the time stamp value for the received code. Together with the MIDI code, the time stamp is then written to the buffer.

Note-by-note MIDI Events Handling

Note-by-note MIDI events handling module generates MIDI events instantaneously upon an application's request. A single MIDI event consists of more than one MIDI code. For example, a 'Note On' event consists of three codes: the channel number, the note to be played and the velocity. Although this string of MIDI code generated need not adhere to any MIDI timing sequence, it needs to work synchronously with the file parser module to prevent nested MIDI events from being processed (i.e. a single MIDI event in process should not be interrupted by other MIDI events on the same channel).

Hardware Handling

This module programs the hardware using the information contained in **BLASTER** environment string. It acts upon commands received from the file parser, the recording module and the note-by-note events handling module.

Selecting MIDI Devices

Every Sound Blaster card has a built in music synthesizer chip. This synthesizer chip refer to the **Internal Music Synthesizer** in this chapter. This chip can be software programmed to produce sounds that close to real musical instruments such as piano, saxophone, violin etc.

Another built-in MIDI device that could be found on the Sound Blaster cards is the MIDI port. It is similar to serial communication ports found on most PC. It can send and receive digital data to or from external MIDI devices. Various external MIDI devices can be connected to the Sound Blaster cards' MIDI port by means of MIDI cables. With proper sequencing software (Creative MIDI driver does provide such a feature), MIDI codes can be sent to the external device. Upon receiving the MIDI code, the device generates sounds that correspond to the MIDI events. Some of the external MIDI devices such as a musical keyboard, can produce sound when it receives MIDI code from an external MIDI source. It can also sequence MIDI codes to other MIDI devices.

Creative MIDI driver supports both internal and external synthesizers. The **MIDI** environment variable will specify which type of synthesizer to be used for MIDI playback.

Sound Blaster cards such as Sound Blaster and Sound Blaster Pro have only a SB-MIDI interface. The Sound Blaster 16 has an additional MPU-401 interface on board. The MIDI driver uses the SB-MIDI interface to access the external MIDI devices for all Sound Blaster cards (except the Sound Blaster 16, which uses the MPU-401 interface). The same MIDI port interface will be used for recording MIDI events.

Users can select which synthesizer type to use for playback, by setting the **MIDI** environment. An application will fetch and pass the **MIDI** environment string to the driver so it will know which synthesizer type is currently selected . If no **MIDI** environment information is received, the driver will use the default synthesizer type (i.e. internal) for playback. You can refer to the chapter "Installation" for more detailed information about the **MIDI** environment.

Selecting MIDI Mapper Types

The MIDI driver defines three preset channel mapper types, General, Extended and Basic channel mappers.

General channel mapper uses all the channels from 1 to 16, with channel 10 being the drum channel.

Extended channel mapper uses the lower 10 channels from 1 to 10, with channel 10 being the drum channel.

Basic channel mapper has only upper four 4 channels from 13 to 16, with the last channel (channel 16) being the drum channel.

To select a channel mapper type, you should call **ctmdSetMidiEnvSettings** and if this call is successfully, the driver will use the mapper type specified by the MIDI environment variable. However, if this call is not successful, the driver will use the default, Extended channel mapper.

Using Creative MIDI Driver

Function Prefix

All function names of the MIDI driver begin with the **ctmd** prefix.

Include Files

You need to include the following header files in all your applications that use the MIDI driver functions:

C Language	Microsoft Basic	Turbo Pascal
SBKMIDI.H	**SBKMIDI.BI**	**SBKMIDI.INC**

Loading and Initializing the Driver

You must allocate a block of memory with an offset value of zero and load the driver into this allocated memory space. You must also assign a far pointer to the segment:offset address of the very first byte of the loaded driver. These far pointers are required by the high-level programming language wrappers to invoke the assembly interface of the drivers.

Driver	Far Pointer
CTMIDI	**CTmidiDRV**

After loading the driver, you should check for the driver version number using the **ctmdGetDrvVer** function. As this is the first loadable Creative MIDI Driver, the version number will be 1.00 at the time of release and there will not be any downward compatibility problem. However, it is a good practice to check for the version number to ensure future API calling convention upward compatibility.

Having determined the correct driver version number, you may pass the **MIDI** environment string to the driver for it to determine what synthesizer type and channel mapper type to use. You do this by calling the **ctmdGetMidiEnvSettings** function. As this function is optional, it is called when an application prefers to use settings other than the default.

You must pass the **BLASTER** environment string to the driver for interpretation, to allow the driver to know which I/O settings to use. You do this by calling the **ctmdGetEnvSettings** function. Since we have enforced the concept of using the environment variables to convey information about a Sound Blaster card, this function must be called before **ctmdInit** as the MIDI driver assumed no default hardware settings.

The I/O settings to use will be based on the **MIDI** environment string. If internal music synthesizer (i.e. SYNTH:1) is used for playback, the driver uses the base I/O address specified by the prefix **A** in the **BLASTER** environment string.

If external synthesizer (i.e. SYNTH:2) is used for playback, Sound Blaster uses the SB-MIDI interface to communicate with external MIDI devices. The base I/O address will be specified by the prefix **A** in the **BLASTER** environment string. For Sound Blaster 16 where the MPU-401 MIDI interface is used, the base I/O address is specified by the Prefix **P** (prefix **P** is used exclusively by Sound Blaster 16). For detailed information on **MIDI** and **BLASTER** environment variables, refer to the chapter "Installation".

Next, use **ctmdInit** function to initialize the driver. The **ctmdInit** function must be invoked before any other driver functions except the two functions: **ctmdGetMidiEnvSettings** and **ctmdGetEnvSettings**.

When you have finished using the driver, you must call the **ctmdTerminate** function to perform the necessary driver termination activities. When these activities are completed, unload the driver.

The following pseudo code fragment describes the process for using the CTMIDI driver:

```
// Load CTMIDI.DRV driver and assign the entry to CTmidiDRV
CTmidiDRV := sbkLoadDriver("CTMIDI.DRV", 0, lpOrgPtr);

IF driver loaded successful THEN
{
        // check for correct driver version
        wDrvVersion := ctmdGetDrvVer();

        // pass MIDI environment string to driver
        //
        // 1.  Get MIDI environment string
        // 2.  Pass to driver by ctmdGetMidiEnvSettings()
        ctmdGetMidiEnvSettings(MIDI string);

        // pass BLASTER environment string to driver
        //
        // 1.  Get BLASTER environment string
        // 2.  Pass to driver by ctmdGetEnvSettings()
        ctmdGetEnvSettings(BLASTER string);

        IF ctmdInit() = 0  THEN
        {

                // Initialization successful.
                // strut your CTMIDI function calls here
                        .
                        .
                        .

                // when you have finished using CTMIDI
                ctmdTerminate();
        }

        // unload the loaded driver here by releasing
        // the loaded driver memory location.
```

```
}
ELSE
{
        // Load driver error
}
```

The **sbkLoadDriver** function is included in the SBK as a helper function in Microsoft Basic and Turbo Pascal or as a module file in C language. Refer to the source code of this function on how it works.

Playing MIDI Events

This section describes the use of the high-level MIDI driver for MIDI music playback.

Playback MIDI Music

Before playing a MIDI music, you must first load the MIDI file into conventional memory before start playing. If the file loads successfully, you must next pass the address of the start of the MIDI data block to the **ctmdPrepareMidiStart** function to determine some of the MIDI file-dependent information, such as number of tracks and resolution of the MIDI file. If the return status indicates no error, then the MIDI music playback can be initiated by calling **ctmdPlayMidiMusic**.

After initiating the MIDI driver for playback, the function returns to your application immediately. Playback will take place in the background.

Monitoring MIDI Music Status

While MIDI music is playing, you can monitor the MIDI process status with the help of its status word. The MIDI driver allows you to set the address of the MIDI music status word using the following function:

ctmdSetOutputStatusAddx(&*lpwStatus*)

After setting the address of the status word, you can use it to monitor the MIDI music process. The drivers update the status word under the following conditions:

1. Resets the status word to 0 during initialization.
2. Sets the status word to 1 when it starts a new MIDI music playback.
3. Sets the status word to 0 at the end of the MIDI music playback.

4. Sets the status word to 2 when the MIDI music is paused.
5. Sets the status word to 1 when the MIDI music resumes after paused.

An application can monitor the status word to determine whether the playback has stopped or paused, and decide what actions to be carried out next.

Controlling MIDI Music Playback

The MIDI driver provides the following functions for controlling MIDI music playback:

Function	Description
ctmdPlayMidiMusic	**Starts the MIDI music output process.**
ctmsPauseMidiMusic	**Pauses the active MIDI music output process.**
ctmdResumeMidiMusic	**Resumes the paused MIDI music output process.**
ctmdStopMidiMusic	**Stops the MIDI music output process.**
ctmdSetChannelMapper	**Sets the user-defined channel mapper.**
ctmdSetMapperType	**Changes the current mapping to one of the three driver's preset channel mappers,** General**,** Extended **and** Basic **channel mapper.**
	You can also set it to the User-defined **channel mapper if you have successfully called the** ctmdSetChannelMapper **to setup the** User-defined **channel mapper.**
ctmdSetMusicTempo	**Sets the tempo multiplier of the playing MIDI music.**
ctmdSetMusicTranspose	**Sets the transpose of the playing MIDI music.**

While the MIDI music is playing, you can control the MIDI music playback by calling the above functions. The driver will carry out the relevant actions and return control to the application immediately when the requests are completed.

While MIDI music is playing, you can call the **ctmdSetMapperType** to override the channel mapper type specified by the **MIDI** environment variable, and switch to any of the three driver preset channel mappers.

You can also arrange your own channel mapping and call **ctmdSetChannelMapper** to set your **user-defined** channel mapper to. Subsequently, you will have one more channel mapper type to play with, in addition to the three driver preset channel mapper types.

The **ctmdGetMapperType** function is provided so that you can query the current channel mapper type the driver is using. The return value will be one of the driver preset channel mappers or the user-defined channel mapper type if the **ctmdSetChannelMapper** function have been previously invoked.

If you would like to change the tempo of the MIDI music playback, you can call **ctmdSetMusicTempo** to adjust the music tempo multiplier. The tempo multiplier value ranges from -20 to 20 (0 being the normal tempo). The tempo multiplier value adjusts the music tempo in step sizes of 10%. For example, tempo multiplier value of +10 will increase the music speed by 100% (i.e. twice the normal speed).

It is possible to perform the transpose control function with the MIDI driver. Simply call **ctmdSetMusicTranspose** to change the transpose of the music. The transpose value ranges from -6 to +6 (0 being the normal transpose). The transpose value adjusts the music tune in step sizes of one semitone.

Recording MIDI Events

The driver provides a few flexible methods of handling MIDI recording. One way is to let the driver do everything for you. In this method, what you need to do is to pass the starting address of the buffer to the driver. The driver will store the incoming MIDI codes into the buffer until the buffer is exhausted upon which time the recording will stop. Alternatively, you can store the incoming MIDI codes using your own code. In this case, you need not pass the driver any information about the buffer, but you must provide a callback function so that the MIDI driver can inform you upon receiving MIDI codes. As the arrival of the MIDI codes are unpredictable, the best way for the driver to notify you is through callback function. The callback mechanism may also be made active even though you have decided to let the driver store the MIDI codes for you. The callback function is called instantly without any delay, whenever a single MIDI code is received by the driver.

To enable the callback mechanism, you need to call **ctmdSetMidiCallBackFunct**. This function takes in two parameters, a far pointer to the callback function and a double word value that will return to you as a second parameter to your callback function during callback. The value of the second parameter is not used by the driver. Its use is entirely up to you, and it will remain unchanged at the time the driver calls

the callback function. One way of using the second parameter could be to store the address of the DATA segment, for access to your global variables.

The callback function is prototyped to be a function that accepts two parameters from the driver. The first parameter is a far pointer to the double-word containing the MIDI code and the Time-Stamp. The second parameter is a double-word user-defined data that you passed as a second parameter to **ctmdSetMidiCallBackFunct**.

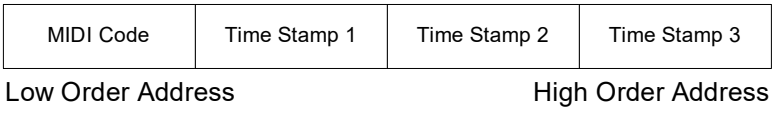
To allow the callback mechanism to function properly, certain criteria must be followed. You can refer to the **Restrictions of MIDI Driver** section for more detailed information.

The callback function is extremely useful for those applications that needs to monitor the MIDI code in real-time. For example, to display the MIDI input event immediately after the MIDI code is received.

In the case of a non timing-critical operation, there is no need to take advantage of this feature. Instead, it can monitor the buffer to determine how many MIDI codes are being received and decide what to do with the MIDI code.

If you have decided to let the driver handle the storing of the MIDI codes into the buffer, the location of the buffer and its size have to be made known to the driver. This can be done by calling **ctmdSetMidiInputBuffer**. The buffer to be used has to follow some of the criteria set by the driver to ensure normal recording operation. This buffer can be dynamically allocated or be static. For both cases, application must use a buffer of size in multiples of four bytes (i.e. double-word) because the driver stores every incoming MIDI code in the form of four consecutive bytes, the first byte (lower address) being the MIDI code and the following three bytes (higher address) being the time-stamp in milliseconds.

The following figure shows the arrangement of the four bytes within the buffer.



For example, if the time stamp for the MIDI code 82 Hex is 012345 Hex milliseconds, then the four bytes within the buffer will be:

The first four bytes of the buffer is not used for storing MIDI codes. The driver stores the count of the MIDI codes received so far in this location. The driver will treat these four bytes as a whole (double word) and will initialize the double word to zero when the **ctmdSetMidiInputBuffer** is called. The size of the buffer includes the first four bytes even though they are not used for storing MIDI codes.

After initiating the recording, these functions returns control to you immediately. The recording will take place in the background. You may call the **ctmdStopMidiInput** to end the MIDI recording. Otherwise, the recording ends when the memory buffer is full.

As this is purely a raw recording MIDI code recording process, the MIDI codes stored in the buffer does not follow the SMF specification. It is your responsibility to rearrange or manipulate the MIDI codes into SMF format after recording.

Time Stamping Modes

MIDI driver provides two time stamping modes, **Differentiate** and **Elapsed**. Both modes use the unit millisecond.

Differentiate mode will record the time differential between the current and the previous MIDI code received. The first MIDI code received will have the time stamp of zero millisecond.

Elapsed mode will record the accumulated running time from the first MIDI code received.

It is up to you to decide which mode to use. You can set the desired mode by calling **ctmdSetTimeStampMode**. By default, the driver uses the **Differentiate** mode.

Monitoring MIDI Recording Status

As for MIDI music playback, while the MIDI recording is taking place, you can also monitor the process status with the help of its status word. The MIDI driver allows you to set the address of the MIDI recording status word by invoking the following function:

ctmdSetInputStatusAddx(&lp*wStatus*)

After setting the address of the status word, you can use it to monitor the MIDI events playback process. The drivers update the status word under the following conditions:

1. Resets the status word to 0 during initialization.
2. Sets the status word to 1 when MIDI recording is started.
3. Sets the status word to 0 at the end of the MIDI recording.

An application can monitor the status word to determine whether the MIDI recording is active or stop, and decide what actions to carry out next.

Playing Note-by-note MIDI Events

In addition to the Standard MIDI file music playback, the driver also supports single note MIDI events playback, two functions are available at the moment.

Function **ctmdSendShortMessage** is for playing a single MIDI channel event. Channel events are those MIDI events that relate to the particular channel only. For example, Note On, Note Off, Program Change, Control Change etc. This function has been prototyped to take in three parameters, first is the MIDI status byte with its most significant bit (MSB) set, that denotes the action to be carried out, its Most Significant Bit (MSB) is set, the second and third parameters are MIDI data bytes with respect to the status byte. Not every channel event has two data bytes. For instance, Program Change event (C*n* Hex) has only one data byte follow the status byte. In this case, the third parameter will be ignored by the driver.

If an external synthesizer is used for playback, the MIDI codes for a MIDI event will be sent out to the external MIDI device. For an internal synthesizer, the driver will program the synthesizer chip according to the status byte (i.e. the first parameter) and the data bytes.

The **ctmdSendLongMessage** function is used for sending a buffer of MIDI codes to the MIDI device. You can use this function to send multiple MIDI events, including system-exclusive messages to the MIDI device.

The first parameter for **ctmdSendLongMessage** function is a far pointer to the starting of the MIDI message to be sent, and the second being the length of the MIDI message in units of bytes.

For buffer that contains multiple MIDI events, each MIDI event must be separated by a delta time (as on the standard MIDI file format). Presently, delta time on the **ctmdSendLongMessage** will be ignored by the MIDI driver. Also, MIDI driver will use the current MIDI tempo and transpose to send the MIDI events.

Restrictions of MIDI Driver

Take note of the following restrictions when using the MIDI driver:

Timer Interrupt Problem

The driver uses the System Timer to sequence the MIDI events for playback as well as for MIDI recording. The driver assumes the normal clock rate of 18.2 times per second. The previous Timer ISR, will be intercepted by the driver's Timer ISR and chained back to it at the frequency based on 18.2 times per second. Therefore, if some programs have programmed the timer to run faster, the driver will still think that it is running at 18.2 times per second. This will cause the original Timer ISR to execute at a lower frequency. Before initializing the driver, you have to ensure the Timer is not being programmed to the timing other than its normal rate.

Memory Size Constraint

The driver supports only conventional memory mode. Sometimes, situations such as "insufficient memory for allocation" may occur if you try to load a very large MIDI file for playback, and at the same time allocate a large buffer for MIDI recording. It is your responsibility to make sure sufficient conventional memory is available when using the driver. One of the solution will be to unload some of the TSRs or other unused resident drivers or split the recording session into two or more buffers, so as to free up as much occupied conventional memory space as possible.

Callback Constraint

Refer to the chapter "Programming Overview" for the details of the callback function constraints.

The time stamp accuracy of the in-bound MIDI codes is an important factor for MIDI recording. If the callback function takes too long to process, the delay will affect the time stamp of the subsequent MIDI codes. In more serious cases, the delay may causes missing in-bound MIDI codes.

Chapter 8
CD-ROM Audio Interface

This chapter describes the programming information for performing audio operations on the Creative CD-ROM drive.

A comprehensive CD-ROM Audio Interface Library is implemented for audio- and drive operation-related functions. The programmer can access the CD-ROM drive status, manipulate audio playback and control the disk tray (for drives that support tray operations by software).

This chapter covers the following topics:

- Using the CD-ROM Audio Interface Library
- Compact disc terminology
- Initializing the CD-ROM drive
- Controlling audio playback
- Accessing compact disc information
- CD-ROM drive-related functions

The following discussions assume that you have basic knowledge of the CD-ROM drive and Compact Disc technology. Therefore, the focus will be on the programming aspect.

Using the CD-ROM Audio Interface Library

The functions provided in the CD-ROM Audio Interface Library enable you to access audio Compact Discs (CDs) conforming to the High Sierra May 28th, or ISO 9660 formats. These formats are widely accepted by audio CD manufacturers as their product standards.

Two drivers need to be loaded before an application program can call any function in the library. The two drivers are:

- Creative hardware-dependent CD-ROM driver
- Microsoft Compact Disc Extension driver

Creative hardware-dependent CD-ROM driver provides a means of interfacing an application program with the CD-ROM drive hardware. It is a device driver which installs via CONFIG.SYS file.

Microsoft Compact Disc Extension driver provides a hardware-transparent interface between an application program and the CD-ROM drive. It is distributed by Microsoft Corporation. Microsoft Compact Disc Extension driver is a memory Terminate and Stay Resident (TSR) program which installs from DOS command line.

These two drivers come with the CD-ROM drive. Refer to your CD-ROM installation note for more information on installing these drivers.

With the two drivers loaded, you can start using the CD-ROM Audio Interface Library functions. You need to include the header file SBKCD.H for C language or SBKCD.BI for Microsoft Basic in all your applications that use the CD-ROM Audio Interface Library functions. All the CD-ROM Audio Interface Library function names are prefixed with **sbcd**.

The following pseudo code fragment shows the general structure of an application program using the library functions:

```
// Initialize CD-ROM drive
sbcdInit( lpNumDrives );

IF  drive initialization successful  THEN
{
        // Call library functions here
        .
        .
        .
}
```

Compact Disc Terminology

The following are some frequently used terms when describing compact discs:

Track Usually the length of a song. For example, an audio CD with 10 tracks contains 10 songs. A track is subdivided into sectors.

Sector A physical data block unit that is 1/75th of a second in length. A sector contains 2352 bytes of audio data in digital form.

Frame Equivalent to a sector in Red Book standard. It is the smallest data block unit.

Red Book Address An addressing unit for locating the position on a CD. A Red Book address is made up of 3 portions, **Minute** (0-59+), **Second** (0-59) and **Frame** (0-74). Each element is represented by a single byte in a 32-bit unsigned long integer (Minute-Second-Frame) with the most significant byte not used. The conversion from the Red Book Address into a physical sector is:

$$Sector = (Minute*60*75) + Second*75 + Frame$$

Track Relative Time Addressing unit; starts at (0 min, 0 sec, 0 frame) from the beginning of a track.

Absolute Time Addressing unit; starts at (0 min, 0 sec, 0 frame) from the beginning of a disc.

Lead-In Address Beginning of disc address from which the audio starts.

Lead-Out Address End of disc address where the audio ends. The volume size of a disc can be computed from this address.

TOC Table of content; a table containing the starting addresses of all sound tracks on a disc. The addresses are specified as 32-bit unsigned long integer in Red Book format.

Initializing the CD-ROM Drive

The following function is provided for initializing the CD-ROM drive:

Function	Description
sbcdInit	Initialize CD-ROM drive

The CD-ROM drive can be initialized by calling the **sbcdInit** function. This function should be called prior to using any other CD-ROM Audio Interface Library functions. The number of CD-ROM drives detected is returned if initialization is successful.

Controlling Audio Playback

The CD-ROM Audio Interface Library provides the following functions to control audio playback from the CD-ROM drive:

Function	Description
sbcdPlay	Initiate audio on the CD
sbcdStop	Stop playing audio
sbcdFastForward	Forward audio playback
sbcdRewind	Rewind audio playback
sbcdNextTrack	Forward audio playback to beginning of next sound track
sbcdPrevTrack	Rewind audio playback to beginning of previous sound track
sbcdPause	Pause audio playback
sbcdContinue	Resume audio playback after the CD-ROM drive is paused

An audio track can be initiated using the **sbcdPlay** function. The position to start playback is specified by the track number, and the offset (in units of seconds) from the beginning of the track. The duration of audio playback is specified in seconds. A value of FFFF hex for duration causes the CD-ROM drive to play till the end of a disc.

During audio playback, calling the **sbcdFastForward** function causes the CD-ROM drive to play ahead of its current position by a specified number of seconds. Calling the **sbcdRewind** function has the opposite effect.

The **sbcdNextTrack** and **sbcdPrevTrack** functions cause audio playback from one track higher and one track lower from the current track, respectively.

The **sbcdPause** function pauses audio playback. The **sbcdContinue** function resumes audio playback starting from the location where the CD-ROM drive was last stopped by the **sbcdPause** function. The **sbcdContinue** function cannot resume audio playback that was stopped by the **sbcdStop** function.

The following pseudo code fragment illustrates the use of the **sbcdPlay**, **sbcdStop**, **sbcdNextTrack** and **sbcdPrevTrack** functions:

```
// Initialize CD-ROM drive
sbcdInit( lpNumDrives );

// If initialization is successful
IF  initialization successful  THEN
{
        // Play audio for 100 seconds, from the
        // beginning of the 1st track
        sbcdPlay( 1, 0, 100 );

        // Jump to track 2
        sbcdNextTrack();

        // Jump back to track 1
        sbcdPrevTrack();

        // Stop playing audio
        sbcdStop();
}
```

The following pseudo code fragment illustrates the use of the **sbcdFastForward**, **sbcdRewind**, **sbcdPause** and **sbcdContinue** functions:

```
// Initialize CD-ROM drive
sbcdInit( lpNumDrives );

// If initialization is successful
IF  initialization successful  THEN
{
        // Play audio from the beginning of the 1st
        // track, for 100 seconds
        sbcdPlay( 1, 0, 100 );

        // Fast forward audio playback by 10 seconds
        sbcdFastForward( 10 );

        // Rewind audio playback by 10 seconds
        sbcdRewind( 10 );
```

```
        // Pause audio playback
        sbcdPause();

        // Resume audio playback
        sbcdContinue();
}
```

Accessing Compact Disc Information

The Audio Interface Library provides the following functions for accessing CD information:

Function	Description
sbcdGetVolume	Get CD volume size, in sectors
sbcdGetDiscInfo	Get highest, lowest track number and lead-out track address
sbcdReadTOC	Get TOC of disc in the drive

The **sbcdGetDiscInfo** function returns the lowest track number, the highest track number and the address of the lead-out track on a CD. Track number on a CD starts from 1. The lowest and highest track number does not include the lead-in and lead-out tracks. The lead-out track address returned is in Red Book format. Information is returned via a DISK_INFO structure defined in the include file. The structure of DISK_INFO is as follows:

```
BYTE   bLoTNo;     // lowest track number
BYTE   bHiTNo;     // highest track number
DWORD  dwLeadOut;  // lead-out track address
```

The **sbcdReadTOC** function returns the Red Book address of all audio tracks as well as the lead-out track on a CD. The addresses are returned via an array of unsigned long integers. The caller must ensure that the array passed in is of sufficient size to store all the addresses of the tracks. To cater for the maximum number of tracks, a typical array size of 100 elements is recommended. Alternatively, the actual number of tracks on a CD can first be determined by calling the **sbcdGetDiscInfo** function.

The following pseudo code fragment illustrates the use of the **sbcdGetDiscInfo** function:

```
// Initialize CD-ROM drive
sbcdInit( lpNumDrive );
```

```
// If initialization is successful
IF  initialization successful  THEN
{
        // Get highest, lowest track number and
        // lead-out track address
        sbcdGetDiscInfo( DISK_INFO lpBuffer );

        // Allocate memory for storing all track
        // addresses; required size is the highest track
        // number plus 1 for the lead-out track
        lpTOCBuffer := allocate memory of size
                        ( (lpBuffer.bHiTNo + 1) * DWORD )

        // Get addresses of all tracks
        sbcdReadTOC( lpTOCBuffer );
}
```

CD-ROM Drive-Related Operations

The following functions are provided for controlling the CD-ROM drive:

Function	Description
sbcdSelectDrive	Select the CD-ROM drive to use
sbcdEject	Open tray
sbcdCloseTray	Close tray
sbcdLockDoor	Lock or unlock tray
sbcdGetAudioStatus	Get audio status to check that the CD-ROM drive is in pause mode
sbcdGetDeviceStatus	Get status of CD-ROM drive
sbcdGetLocInfo	Get current location of CD
sbcdMediaChanged	Check if disc has been changed

When more than one CD-ROM drive are daisy-chained, the **sbcdSelectDrive** function selects the drive to use for all Audio Interface functions following the call (until another drive is selected). Each CD-ROM drive is identified by a drive letter ranging from A to Z. The number of CD-ROM drives that are daisy-chained can be determined by calling the **sbcdInit** function. An error code is returned on an attempt to select a non-existent CD-ROM drive. The default drive is the first CD-ROM drive.

The **sbcdEject** function opens the drive tray while the **sbcdCloseTray** function closes it. These two functions have no effect on drives that do not support software tray operations.

The **sbcdLockDoor** function locks and unlocks the tray. A locked tray cannot be opened by pressing the "Eject" button on the drive, or by calling the **sbcdEject** function. If the **sbcdLockDoor** function is called when the tray is open, it will become locked on closing. The **sbcdLockDoor** function has no effect on drives with no door-locking feature.

The **sbcdGetAudioStatus** function returns the audio status of the CD-ROM drive. A value of '1' is returned if the drive is in pause mode. When the CD-ROM drive is in pause mode, audio playback can be resumed by calling the **sbcdContinue** function.

The **sbcdGetDeviceStatus** function returns a 32-bit value representing the status of the CD-ROM drive. The interpretation of the bits are as follows (bit 0 is the least significant bit):

```
Bit 0    0      Door is closed
         1      Door is open
Bit 1    0      Door is locked
         1      Door is unlocked
Bit 2-10        Reserved
Bit 11   0      Disc is in drive
         1      No disc is in drive
Bit 12-31       Reserved
```

The **sbcdGetLocInfo** function returns the current location, in Red Book format, of the CD through a QCHAN_INFO structure defined in the include file. The structure of QCHAN_INFO is as follows:

```
BYTE          bTNo;          // current track number
BYTE          bReserved
BYTE          bMin;          // minute }    running time within
BYTE          bSec;          // second }    a track
BYTE          bFrame;        // frame  }    (relative time)
BYTE          bReserved;
BYTE          bPMin;         // minute }
BYTE          bPSec;         // second }    running time on the
BYTE          bPFrame;       // frame  }    disk (absolute time)
```

`bMin`, `bSec` and `bFrame` are the track relative time of the CD. The track relative time starts from 00:00:00 (0 min, 0 sec, 0 frame) at the beginning of a track. `bPMin`, `bPSec` and `bPFrame` are the absolute time on the CD. The absolute time starts from 00:00:00 from the beginning of the disc (lead-in area).

The **sbcdMediaChanged** function checks if a disc in the CD-ROM is changed. A 0 value is returned if the disc has not been changed. This function can be used to

determine if a previously read TOC is still valid for the disc in the drive. If a new disc has been inserted, its new TOC has to be read by calling the **sbcdReadTOC** function.

The following pseudo code fragment illustrates the use of the **sbcdSelectDrive**, **sbcdEject**, **sbcdCloseTray** and **sbcdLockDoor** functions:

```
// Initialize CD-ROM drive
sbcdInit( lpNumDrive );

// If initialization is successful
IF  initialization successful  THEN
{
        // Select drive D; A=0, B=1, C=2, D=3, ...
        sbcdSelectDrive( 3 );

        // Open the tray for putting in a CD
        sbcdEject();

        // Close the tray
        sbcdCloseTray();

        // Lock the tray to prevent it from opening
        // by anyone pressing the "Eject" button;
        // '1' to lock, '0' to unlock
        sbcdLockDoor( 1 );

        // Play the entire disc
        sbcdPlay( 1, 0, 0xFFFF );
}
```

The following pseudo code fragment illustrates the use of the **sbcdMediaChanged** function:

```
// Initialize CD-ROM drive
sbcdInit( lpNumDrive );

// If initialization is successful
IF  initialization successful  THEN
{
        // Read TOC
        sbcdReadTOC( lpTOCBuffer );

        // Other Audio Interface functions
        .
        .
        .

        // Check if the same disc is still in the drive
        change := sbcdMediaChanged();

        // Read TOC if disc has been changed
        IF (change != 1) THEN
                sbcdReadTOC( lpTOCBuffer );
}
```

Migration Guide

This chapter is intended to aid the existing Sound Blaster developers to convert their existing applications to use the new libraries. It highlights the essentials that you need to know when converting your applications.

We use the same chapter headers on the previous edition manual for the following sections in our discussions. This helps you locate the information you want easily.

You should keep a copy of the previous edition of the Developer Kit so that you could cross-reference to any of its functions when needed.

For simplicity sake, we will use a C function name to identify the function. If you are Turbo Pascal or Microsoft Basic users, refer to previous edition manual for the corresponding function names.

General Functions

The use of the **GetEnvSetting** function to initialize the global variables' approach has been removed. There will be no **ct_io_addx**, **ct_int_num** and **ct_dma_channel** global variables in the current implementation. The limitation with the **GetEnvSetting** function is that it is automatically rendered out of date whenever a new Sound Blaster with a different range of hardware settings is introduced. To solve this problem, the new approach lets the drivers parse the **BLASTER** environment settings to decide the hardware settings it can use (since each driver is tailored for a specific card). All the loadable drivers now include the function **ct*??*GetEnvSettings** for passing the BLASTER environment settings to the driver. You should use this function instead.

The card testing functions **sbc_check_card**, **sbc_test_int**, and **sbc_test_dma** test the card based on the abovementioned global variables. They have also been removed. The initialization function of the driver will now take care of all the hardware testing.

CT Voice

The API of the CT-VOICE and CTVDSK drivers have been rethought. Considerable changes have been made to the old API. Observe the important notes below when converting your applications to use the new API:

1. Remove the call to the **GetEnvSetting** function in the driver initialization sequence; instead, call the **ctv?GetEnvSettings** function before **ctv?Init**.

2. Get rid of the sequence of **sbc_check_card**, **sbc_test_int** and **sbc_test_dma** calls before initializing the driver. Let the drivers' **ctv?Init** function takes care of all the hardware testing.

3. The **ctv?Init** function no longer sets the address of the global variable **ct_voice_status**. It is now your application's responsibility to explicitly call the function **ctv?SetIOParam** to set the address of the status word.

4. For Turbo Pascal users, the leading underscore of global far pointer to the driver entry point has been removed. For instance, **_voice_drv** has been changed to **voice_drv**, and **_ctvdsk_drv** to **ctvdsk_drv**. For Microsoft Basic users, the function to set the driver entry point is **ctv?SetDriverEntry**.

5. The required include file names for **ctv?** functions have been changed from SBCVOICE.??? to SBKVOICE.???, where the ??? is the file extension depending on the programming languages you use.

To aid in migration to the new API, the table below lists the successors to the old functions. Where more than one function are placed in the right-hand column, they will be numbered in ascending order, possibly with an alphabetic suffix. This has a certain significance: function #1a and #1b must be called before function #2, but function #1b may be called before function #1a.

Old Functions	New Functions
ctv*?*_break_loop(*BreakMode*)	ctv*?*BreakLoop(*wIOHandle*, *BreakMode*)
ctvd_buffer_addx()	ctvvdSetDiskBuffer(*wIOHandle, lpBuffer, BufferSixe*)
ctv*?*_continue()	ctv*?*Continue(*wIOHandle*)
ctvd_drv_error()	ctvdGetDrvError()

ctvd_ext_error()	ctvdGetExtError()
ctv*?*_get_ADC_range(*mode*)	ctv*?*GetParam(CTVOC_SAMPLINGRANGE, &*dwSamplingLimit*)
ctv*?*_get_cardtype()	ctv*?*GetParam(CTVOC_CARDTYPE, &*dwCardType*)
ctv*?*_get_DAC_range	ctv*?*GetParam(CTVOC_SAMPLINGRANGE, &*dwSamplingLimit*)
ctvd_init(*BufferSize*)	1. ctvdGetEnvSettings(*lpszBlaster*) 2. ctvdInit() 3a. ctvdSetDiskBuffer(*wIOHandle*, *lpBuffer*, *BufferSize*) 3b. ctvdSetIOParam(*wIOHandle*, CTVOC_IO_LPSTATUSWORD, &*wVoiceStatus*)
ctvm_init()	1. ctvmGetEnvSettings(*lpszBlaster*) 2. ctvmInit() 3. ctvmSetIOParam(*wIOHandle*, CTVOC_IO_LPSTATUSWORD, &*wVoiceStatus*)
ctvd_input(*FileHandle*, *SamplingRate*)	1a. ctvdSetIOParam(*wIOHandle*, CTVOC_IN_FORMAT,*FormatTag*) 1b. ctvdSetIOParam(*wIOHandle*, CTVOC_IN_BITSPERSAMPLE, *SamplingRate*) 1c. ctvdSetIOParam(*wIOHandle*, CTVOC_IN_NCHANNELS,*NumChannels*) 1d. ctvdSetIOParam(*wIOHandle*, CTVOC_IN_SAMPLESPERSEC, *SamplingRate*) 2. ctvdInput(*wIOHandle*, *FileHandle*)
ctvm_input(*lpBuf*, *BufLen*, *SamplingRate*)	1a. ctvmSetIOParam(*wIOHandle*, CTVOC_IN_FORMAT,*FormatTag*) 1b. ctvmSetIOParam(*wIOHandle*, CTVOC_IN_BITSPERSAMPLE, *SamplingRate*) 1c. ctvmSetIOParam(*wIOHandle*, CTVOC_IN_NCHANNELS,*NumChannels*) 1d. ctvmSetIOParam(*wIOHandle*, CTVOC_IN_SAMPLESPERSEC, *SamplingRate*) 2. ctvmInputCM(*wIOHandle*, *lpBuf*, *BufLen*)

ctvm_inputxms(*XMBHandle*, *SamplingRate*, *XMBOffset*, *XMBSize*)	1a. ctvmSetIOParam(*wIOHandle*, CTVOC_IN_FORMAT,*FormatTag*) 1b. ctvmSetIOParam(*wIOHandle*, CTVOC_IN_BITSPERSAMPLE, *SamplingRate*) 1c. ctvmSetIOParam(*wIOHandle*, CTVOC_IN_NCHANNELS,*NumChannels*) 1d. ctvmSetIOParam(*wIOHandle*, CTVOC_IN_SAMPLESPERSEC, *SamplingRate*) 2. ctvmInputXM(*wIOHandle*, *XMBHandle*, *XMBOffset*,*XMBSize*)
ctvd_output(*FileHandle*)	ctvdOutput(*wIOHandle*, *FileHandle*)
ctvm_output(*lpBuf*)	ctvmOutputCM(*wIOHandle*, *lpBuf*)
ctvm_outputxms(*XMBHandle*, *XMBOffset*)	ctvmOutputXM(*wIOHandle*, *XMBHandle*, *XMBOffset*)
ctv*?*_pause()	ctv*?*Pause(*wIOHandle)*
ctv*?*_set_input_source(*source*)	a. ctv*?*SetIOParam(*wIOHandle*, CTVOC_IN_LEFTINPUTS, *LeftInputSwitches*) b. ctv*?*SetIOParam(*wIOHandle*, CTVOC_IN_RIGHTINPUTS, *RightInputSwitches*)
ctv*?*_set_stereo(*mode*)	ctv*?*SetIOParam(*wIOHandle*, CTVOC_IN_NCHANNELS, *NumChannels*)
ctv*?*_speaker(*fOnOff*)	ctv*?*SetSpeaker(*fOnOff*)
ctv*?*_stop()	ctv*?*Stop(*wIOHandle*)
ctv*?*_terminate()	ctv*?*Terminate()
ctv*?*_version()	ctv*?*GetParam(CTVOC_DRIVERVERSION, &*dwDriverVersion*)

Mixer

All the functions to the auxiliary driver are retained. You only need to make the following changes to use the new libraries:

1. Remove the call to the **GetEnvSetting** function in the driver initialization sequence; instead, call the **ctadGetEnvSettings** function before **ctadInit**.

2. The **ctadInit** function no longer sets the addresses of the global variables **CTFadeStatus** and **CTPanStatus**. Therefore, your application is responsible to explicitly call the **ctadSetFadeStAddx** and **ctadSetPanStAddx** to inform the driver as to the whereabouts of the fade and pan status words.

3. For Turbo Pascal users, the global far pointer to the driver entry point has been changed from **_CTAuxDrv** to **CTAuxDrv** (the leading underscore is removed). For Microsoft Basic users, the function to set the driver entry address has been changed from **AUXADDX** to **ctadSetDriverEntry**.

4. The required include file names have been changed from AUXDRV.??? to SBKAUX.???, where the ??? is the file extension depending on the programming languages you use.

FM Music

The FM music support files have been changed from .CMF files to .MID files. To cater for the existing applications which were written to play .CMF files, we have retained all the SBFMDRV.COM driver interface functions in the new libraries. Refer to the previous edition manual for these functions' descriptions. For new application, you should use the CTMIDI driver interface function to play .CMF files if your application plays FM music.

You should use the include file SBCMUSIC.??? in your previous edition package when you are using the SBFMDRV driver interface functions.

The table below lists the SBFMDRV driver interface functions that are retained in the new libraries:

Function Name
sbfm_init
sbfm_instrument
sbfm_pause_music
sbfm_play_music
sbfm_read_status
sbfm_reset
sbfm_resume_music
sbfm_set_channel
sbfm_song_speed
sbfm_stop_music
sbfm_sys_speed
sbfm_terminate
sbfm_transpose
sbfm_version

MIDI Interface

The MIDI Interface functions have been changed from embedded library functions to CTMIDI.DRV loadable driver. Like other loadable drivers, you need to load it into memory before you can use any of its functions. Follow the guidelines below to convert your application:

1. Remove the call to the **GetEnvSetting** function. Instead call the **ctmdGetEnvSettings** function to process the BLASTER environment settings.

2. Get rid of the functions **sbc_check_card**, **sbc_test_int** and **sbc_test_dma** calls for the hardware testing. The drivers' **ctmdInit** function takes care of all the hardware testing.

3. You must call the **ctmdInit** function before any MIDI Interface functions. Also, you must call the **ctmdTerminate** before the program terminates.

4. The required include file names have been changed from SBMIDI.??? to SBKMIDI.???, where the ??? is the file extension depending on the programming languages you use.

The table below lists the successors to the old functions for sending MIDI data:

Old Functions	New Functions
sbmidi_out_shortmsg	ctmdSendShortMessage
sbmidi_out_longmsg	ctmdSendLongMessage

For receiving MIDI data, the handling have slight changes particularly on the MIDI data retrieval. The previous approach had an embedded 8KB circular buffer for storing MIDI in-bound data. The function **sbmidi_get_input** was used to retrieve MIDI data in the embedded buffer. The new approach requires the user to pass a buffer as storage for MIDI in-bound data to the driver. As such, user can access the MIDI buffer for the in-bound data.

The table below lists the successors to the old functions for receiving MIDI data. Where more than one function is placed in the right-hand column, they will be numbered in ascending order, possibly with an alphabetic suffix. This has a certain significance: function #1a and #1b must be called before function #2, but function #1b may be called before function #1a.

Old Functions	New Functions
sbmidi_start_input()	1a. ctmdSetTimeStampMode(ELAPSED_MODE) 1b. ctmdSetMidiInputBuffer(*lpBuf*, *dwBufSize*) 2. ctmdStartMidiInput()
sbmidi_stop_input()	ctmdStopMidiInput()

For the new API, MIDI input will stop automatically when input MIDI buffer is full.

CD-ROM Audio Interface

The CD-ROM audio functions remain the same except of the naming conventions. To use the new libraries, you need to change the name of the CD-ROM audio functions used in your application to the corresponding new function names, then recompile and relink.

The required include file names have been changed from SBCD.??? to SBKCD.???, where the ??? is the file extension depending on the programming languages you use.

The table below lists the old function names and their corresponding new function names:

Old Function Name	New Function Name
sbcd_continue	sbcdContinue
sbcd_fastforward	sbcdFastForward
sbcd_get_aud_status	sbcdGetAudioStatus
sbcd_get_disc_info	sbcdGetDiscInfo
sbcd_get_loc_info	sbcdGetLocInfo
sbcd_get_volume	sbcdGetVolume
sbcd_init	sbcdInit
sbcd_media_changed	sbcdMediaChanged
sbcd_next_track	sbcdNextTrack
sbcd_pause	sbcdPause
sbcd_play	sbcdPlay
sbcd_prev_track	sbcdPrevTrack
sbcd_read_toc	sbcdReadTOC
sbcd_rewind	sbcdRewind
sbcd_select_drv	sbcdSelectDrive
sbcd_stop	sbcdStop

Appendix B
Relevant Information

If you need more information on the digital audio, MIDI, CD-ROM, Extended Memory Specifications, you may refer to the following sources:

Digital Audio

For more information on digital audio, see the following books:

Principles of Digital Audio
Ken C. Pohlmann
Howard W. Sams & Company

Digital Audio Engineering, An Anthology
Strawn John F.
William Kaufmann, Inc.

MIDI

For more information on MIDI, see the following book:

MIDI: A Comprehensive introduction
Joseph Rothstein
A-R Editions, Inc

CD-ROM

For more information on the CD-ROM, see the following books:

Guide to CD-ROM
Dana Parker & Bob Starrett
New Riders Publishing

The Compact Disc: A handbook of theory and use
Ken C. Pohlmann
A-R Editions, Inc.
Madison, Wisconsin

Microsoft MS-DOS CD-ROM Extensions (MSCDEX) Specification is available from Microsoft Corporation.

Extended Memory Specification

For more information on the Extended Memory Specification, see the following sources:

Extending DOS
Edited by Ray Duncan
Addison-Wesley Publishing Company, Inc.

Extended Memory Specification version 2.0 is available from Microsoft Corporation.

Glossary

A

Advanced Signal Processor A component of Sound Blaster 16 Advanced Signal Processing that performs high speed mathematical processing on the in-bound and out-bound digitized sound data.

Auto-initialize DMA mode A DMA transfer mode where the DMA controller automatically reloads the transfer address, and counter at the moment it decrements from zero to FFFF hex.

AUXDRV.DRV Creative high-level loadable auxiliary driver that controls the mixer chip.

B

Basic MIDI map A type of MIDI mapping used for low-end synthesizers. It uses MIDI channels 13 to 16 where channel 16 is the percussion channel.

BLASTER environment string An environment string identifying the current Sound Blaster card settings.

Block Type A block header that identifies the type of data contained in the Creative Voice File (.VOC).

C

Callback function A function invoked by a driver to notify an application when certain events occur.

CCITT A-Law An audio-compression technique.

CCITT μ-Law An audio-compression technique.

CD-ROM Acronym for Compact Disc-Read Only Memory. It is an optical data storage technology that allows large quantities of data to be stored on a compact disc.

Creative ADPCM Acronym for Creative Adaptive Differential Pulse Code Modulation. It is an audio-compression technique.

CSP.SYS A device driver that mediates access to the Creative Advanced Signal Processor chip.

CTMIDI.DRV Creative loadable MIDI driver that handles MIDI operations.

CTMMSYS.SYS Creative Multimedia System driver that controls digitized sound I/O, auxiliary devices and Advanced Signal Processor chip.

CTSOUND.SYS A device driver that communicates with Sound Blaster Digital Sound Processor and Mixer.

CTVDSK.DRV Creative high-level loadable digitized sound driver that handles the recording and playback of digitized sound data (from .VOC file), when using disk as the working space.

CT-VOICE.DRV Creative high-level loadable digitized sound driver that handles the recording and playback of digitized sound data (from .VOC file), when using memory as the working space.

CTWDSK.DRV Creative high-level loadable digitized sound driver that handles the recording and playback of waveform data (from .WAV file), when using disk as the working space.

CTWMEM.DRV Creative high-level loadable digitized sound driver that handles the recording and playback of waveform data (from .WAV file), when using memory as the working space.

D

Digital Sound Processor (DSP) A component of Sound Blaster card that interprets commands sent to the card. It also handles digitized sound and MIDI port I/O.

Disk buffer A temporary memory buffer to store data for transfer between disk and DMA transfer buffer.

DMA transfer buffer A memory buffer used to perform auto-initialize DMA mode digitized sound I/O. DMA transfer buffer must not straddle a 64KB physical page boundary.

Device-level drivers Hardware-dependent drivers that communicate directly with Sound Blaster hardware.

E

Extended MIDI map A type of MIDI mapping used for high-end synthesizers. It uses MIDI channels 1 to 10 where channel 10 is the percussion channel.

F

FM synthesizer A synthesizer that creates sounds by combining the output of digital oscillators using frequency modulation technique.

G

General MIDI map A type of MIDI mapping that uses all MIDI channels 1 to 16. Usually, percussion channel will be on channel 10.

H

High-level digitized sound driver A collective term that refers to CT-VOICE.DRV, CTVDSK.DRV, CTWMEM and CTWDSK.DRV drivers.

I

I/O handle A parameter that uniquely identifies I/O process.

L

Loadable driver A binary image of driver code that must be loaded into memory before it can be invoked. An application invokes a loadable driver via an inter-segment call to the driver entry-point.

Low-level driver A term that refers to Creative Multimedia System driver, CTMMSYS.

M

.MID file A file format for storing MIDI music commonly used in Windows environment.

MIDI Acronym for Musical Instrument Digital Interface. It is a standard communications protocol between musical instruments and other musical devices such as computers or synthesizers.

MIDI channel A logical representation in MIDI that identifies a way to send messages to an individual device.

MIDI environment string An environment string identifying the MIDI mapper type and MIDI device to use in an operation.

MIDI mapper type A mapping that translates and redirects MIDI messages according to the setup in the MIDI map. Generally, mapper types are General, Basic and Extended.

MIDI sequencer A program that creates or plays music stored as MIDI files.

MIDI system-exclusive message A special MIDI message understood only by MIDI devices from a specific manufacturer. The standard MIDI specification defines only a framework for system-exclusive messages.

MSCDEX.EXE A driver distributed by Microsoft Corporation that makes CD-ROM drives appear to MS-DOS as network drives. MSCDEX uses hardware-dependent drivers to communicate with a CD-ROM drive.

Mixer A component of Sound Blaster card providing volume control of various input and output sources. It also controls the recording source selection.

R

Red book audio An optical data-storage format for storing high-quality digital-audio data on a compact disc.

S

Sampling rate The number of samples per second.

SBCD.SYS Creative hardware-dependent CD-ROM driver. MSCDEX uses this driver to communicate with a Creative CD-ROM drive.

SOUND environment string An environment string identifying the path where Sound Blaster software and drivers are located.

T

Tempo The speed of a musical piece.

Time stamp A time information tagged with the in-bound MIDI data so that a sequencer can replay the data at the proper tempo.

Track With a CD-ROM audio, this refers to a sequence of audio pieces on a red book audio disc. A track usually corresponds to a song. With a MIDI file, tracks can correspond to MIDI channels, or can correspond to parts of a song.

Transposing A process of changing the tune of a musical piece.

V

.VOC file Creative Voice File format for storing digitized sound data.

W

.WAV file A file format for storing waveform data commonly used in Windows environment.

Index

T

W