Caveat

This version of the SDTK manual is a preliminary version. Please pardon the poor layout and omittances in the material.

Table of Contents

1:Index

The GSS Software Developper Toolkit (SDTK) is a set of software applications, libraries, documentation and other information that will accelerate application support for the GSS-compliant sound cards.

The Developer Toolkit covers the following areas:

Software Development Libraries

This section explains how your applications can interface with the Software Development libraries. It also contains a complete function directory for each of the library modules. Sample source code is supplied on diskette to provide a better understanding of the use of the libraries.

Low-Level Programming
Details the I/O map of each of the hardware sections of the GSS Cards. This section is intended for programmers who want to directly access the hardware, instead of using the software drivers.

Appendices
The appendices provide additional information on the GSS Sound Standard Interface definition.

1.1

The GSS Software Developper Toolkit (SDTK) is a set of libraries that you can use to accelerate the developpement of applications that will support Gold Sound Standard compliant sound cards. Full source code to the libraries, as well as example source code on how to use these libraries is provided.

The SDTK can be normally distributed by electronic means. It sometimes will be distributed in two separate parts, which can be in two separate compressed files on BBS. In this case, the first file contains this User's Manual, and the second file contains the software source code and examples.

Directory structure of the libraries

The main directory of the SDTK contains source code of the libraries. You can freely use this source code to write your own device drivers or other applications.

The main directory also contains the MAKE files necessary to create the libraries. You may need to alter the MAKE files to customize the libraries to your specific compiler or memory model. This is explained later on in this chapter.

A subdirectory OBJECT contains the object files resulting from the compiling of the library source code.

Subdirectory SAMPLES contains sample source code that shows simple uses of the various library modules. It contains the source code and executable versions of each of the examples. Directory SAMPLES has its own OBJECT subdirectory to contain the object code of the sample modules.

Customizing the MAKE file

It is relatively easy to customize the MAKE file to your environment, your compiler and memory model. Most of those variables on the MAKE file are using MACROS, which can easily be redefined. Since creating a MAKE file that would take in consideration all possible options would be very difficult, and would render the MAKE file very difficult to read, we strongly urge you to take a close look to the file and to alter it to suit your own personnal needs.

Operating directories

In the macro named "BaseDir", you can put the name given to the SDTK root directory. A number of subsequent macros are used to define other associated directories. You would not need to alter them if you have kept the original subdirectory structure.

The macros "Compile", "Assemble", "Link" and "Lib" can be altered to specify the path of your compiler and associated tools.

Compiler: Models and Version

The SDTK source code can be compiled using the Microsoft C6.0 or Microsoft C7.0 compiler, as well as the Borland C and C++ compiler version 2.0.

To compile with Microsoft C compilers, you need to include, in the compile line, the following compiler option:

    /DMICROSOFT

and to compile under Borland compilers, you need to include the option:

-DTURBO

These defines are used in the source code to generate compiler-specific function calls.

The MAKE file contains a macro "Compile" where you can define the command line for your compiler.

Compiling for GSS Compatibility Level 1 and Level 2

The SDTK Control Driver automatically detects for a GSS Level 1 or GSS Level 2 card when you call *InitControlDriver()*. However, some functions are not useable under GSS Level 1 card.

InitControlDriver() sets global variable, called *gssLevel*, that you can use to determine the GSS compatibility level of the card used. It can take one of three predefined values (defined in Control.h):

levelNoCard:	no card found
level1:	GSS Level 1 card
level2:	GSS Level 2 card

Here are some areas to be careful about when using this toolkit while operating on level1 cards:

Mixer functions will have no effect on some Level 1 cards

Address relocation is not available on level 1 cards

OPL3 timers are not available.

Functionality

A GSS-compliant sound card is a multifunction card whose minimal functions include digital recording, playback of digitized and synthesized sounds, MIDI recording and playback and game port.

There are two different levels of compatibility of GSS hardware. GSS Level 1 cards offer the standard support of FM sounds, Digitized sound, timers, joystick and MIDI, through the MMA and OPL3 chips.

GSS-Level 2 sound cards also include a software programmable digital audio mixer, and programmable configuration of the card.

The SDTK software libraries offer support of both Levels of compatibility. However, some functions are available in Level 2, and are not under Level 1.

Digital Recording and Playback

GSS sound cards offer two seperate monophonic channels of digital recording and playback, at fixed rates of 5.5Khz, 7.3Khz, 11Khz, 22Khz and 44.1Khz, through the MMA chip. It can also record and play a single channel of stereophonic data at the same rates.

Although the MMA DAC is a 12-bit DAC, the MMA chip supports 8-bit 12-bit and 16-bit data formats, providing for upgradability in the future. A 4-bit ADPCM format is also available, giving high-quality sound with reasonable memory consumption.

The 8-bit format is a signed-integer format, with null speaker displacement at 0x0 and maximal speaker displacement at 0x7F and 0xFF. This contrasts with the unsigned integer format, which is also widely used, that places null speaker displacement at 0x80 and maximal speaker displacement at 0x0 and 0xFF.

To convert from one format to another each sample simply needs to be XORed with 0x80.

Each of the digitized sound channels support interrupt-mode and DMA-mode transfers. Each channel also has a 128 byte FIFO buffer. The buffers can generate interrupts at programmable levels, to facilitate programming and improve programming flexibility.

When performing DMA transfers, DMA data is put or read directly in the channel FIFO. Progamming for DMA transfer mode is then quite similar to programming for interrupt transfer mode.

FM Sound Playback

The OPL3 chip provides for a variable configuration of 4-operator FM voices and 2-operator FM voices, giving up to 20 2-operator FM voices.

Each of the separate voices can be panned left, right or center, for stereophonic effect.

The number of operator waveforms was improved to 6 basic waveforms, giving richer sounds.

MIDI Recording and Playback

The MMA provides a MIDI (Musical Instrument Digital Interface) interface. Separate MIDI input and output 16-byte FIFO buffers and interrupt-driven interface facilitate the programming tasks.

Game Port

The MMA also provides a standard IBM compatible game-port interface.

Differences between GSS-Level 1 and GSS Level 2 hardware

Additionnal features of GSS-Level 2 hardware include a standard on-board programmable mixer, which also is used for software configuration of GSS cards.

This mixer enables the independent programming of each audio sources volume, and a global bass and treble control.

The applications can also read from GSS-Level 2 cards, the DMA channel and interrupt line assignments used on the card.

GSS-Level 2 cards share 1 single interrupt line for interrupts coming from OPL3 and MMA.

Because of these differences, functions in the SDTK libraries that refer to the OPL3 timer interrupts or to the mixer capabilities are disabled in Level-1 code.

2:Index

2.1

Overview

The SDTK libraries are written in C language and are conceived to be directly linked into your application.

Different libraries are provided to support the various memory-model options offered by compilers. All these libraries are functionnaly equivalent.

Source code for the various modules of the libraries is also supplied. You can alter the source code if you wish.

A makefile is supplied, which is based on the BorlandC environement. To customize for your specific compiler needs, you only need to alter the MAKEFILE and DRIVERS.LNK files. Make sure that the compiler options used when making the library match the options used in your application.

The SAMPLE directory provides sample code which can be used to test specific parts of the drivers. Each of the sample applications is described in more detail further in this section.

The function nomenclature refers to each of the librarie's modules as "drivers". This nomenclature was kept for historcal reasons, although no memory-resident drivers are involved.

Module Interaction

The library is composed of 5 separate functionnal entities (called here by the misnomer "drivers"):

Control Driver

Manages the mixer and configuration features of the cards, and also centralizes interrupt-handling for each of the other drivers.

FM Driver

Manages all of the FM-Synthesis functions of the card.

Timer Driver

Provides functions to program the OPL3 and MMA timers, and hook-up to the interrupts generated by the timers.

MIDI Driver

Provides functions to control input and output of MIDI data through the MMA MIDI port.

Wave Driver

Provides functions to play sampled data from memory and to record sampled data to memory.

All drivers are dependent on the Control Driver to handle the interrupts, therefore, the Control driver should always be the first initialized (InitControlDriver()) and the last closed (CloseControlDriver()).

SetControlRegister

Syntax

`int SetControlRegister`**`(int reg, WORD val)`**

Sets register 'reg' of Control Chip to 'val'.

Parameters

`int` **`reg`**

Which register to write to.

`WORD` **`val`**

Which value to write in register.

Return value

If no error 0, otherwise 1.

Comments

This low-level routine handles the details related to accessing the Control Chip, like interrupt disabling and reenabling. It also verifies that no access is made while the Control Chip's RB & SB bits are set.

CtStoreConfiglnPermMem

Syntax

```
WORD     CtStoreConfigInPermMem()
```

This causes all control chip registers, in their current state, to be written to permanent memory.

Parameters

None

Return value

1 if ok. 0 if a problem occured.

Comments

None

CtRestoreConfigFromPermMem

Syntax

```
WORD     CtRestoreConfigFromPermMem()
```

Restores the card configuration from permanent memory.

Parameters

None

Return value

1 if ok. 0 if a poblem occured

Comments

None

CtSetChannel0SampGain

CtSetChannel1SampGain
CtGetChannel0SampGain
CtGetChannel1SampGain

Syntax

```
WORD    CtSetChannel0SampGain(WORD value)
WORD    CtSetChannel1SampGain(WORD value)
WORD    CtGetChannel0SampGain(WORD value)
WORD    CtGetChannel1SampGain(WORD value)
```

Sets the gain of sampling channels.

Parameters

WORD **value**

Gain value from 0 to 255.

256 different values possible giving a range from approximately 0.04 to 10 times the input value. The exact gain is given by the equation:

Gain = (registerValue * 10) / 256 Linear gain.

Return value

1 if ok.

Comments

None

CtSetChannelFilter0Mode

CtSetChannel1FilterMode

Syntax

```
WORD     CtSetChannel0FilterMode(WORD value)
WORD     CtSetChannel1FilterMode(WORD value)
```

Sets the antialiasing fiters in the proper mode for the channel.

Parameters

WORD **value**

0 = playback mode, 1 = sample mode

Return Value

1 if ok.

Comments

This filter MUST be set in sample mode before sampling.

This filter MUST be set in playback mode before playback.

GSS cards use the same antialiasing filters during sampling and playback. The appropriate filter mode must be set before any sampling or playback operation.

CtGetChannelFilter0Mode

CtGetChannel1FilterMode

Syntax

```
WORD     CtGetChannel0FilterMode(void)
WORD     CtGetChannel1FilterMode(void)
```

Returns the current antialisaing filter mode for the channel.

Parameters

None

Return Value

0: playback mode. 1: Sampling mode

Comments

None

CtStereoMonoAuxSamp

Syntax

`WORD      CtStereoMonoAuxSamp`**`(WORD value)`**

Forces auxiliary inputs to work monophonically or sterophonically.

Parameters

`WORD      `**`value`**

0 = auxiliary input is stereo, 1 = auxiliary input is mono

Return Value

1 if ok.

Comments

The microphone and telephone inputs are monophonic sources and can only be sampled monophonically on channel 0. However, the auxiliary inputs are normally sampled in stereo on both channel 0 and 1 at the same time. This stereo audio input can be turned monophonic and sampled on channel 0 using this function.

CtGetStereoMonoAuxSamp

Syntax

```
WORD     CtGetStereoMonoAuxSamp(void)
```

Returns whether the auxiliary inputs are used for monophonic sampling or stereophonic sampling.

Parameters

None

Return Value

0 = auxiliary input is stereo, 1 = auxiliary input is mono

Comments

None

CtEnabDisabMicroOutput

Syntax

`WORD CtEnabDisabMicroOutput` **(WORD value)**

Enables/disables microphone output.

Parameters

WORD **value**

0 = Microphone output enabled, 1 = Microphone output disabled

Return Value

1 if ok.

Comments

When using the microphone input and the normal loudspeaker outputs of the audio card, audio feedback could result. In normal mode, microphone output is enabledd When disabled, the microphone signal is cut from the output of the card but sent to the telephone output, eliminating possible causes of feedback.

CtGetEnabDisabMicroOutput

Syntax

```
WORD     CtGetEnabDisabMicroOutput()
```

When using the microphone input and the normal loudspeaker outputs of the audio card, audio feedback could result. In normal mode, this bit is set to 0. When set to 1, the microphone signal is cut from the output of the card and only sent to the telephone output, eliminating possible causes of feedback.

Parameters

None

Return Value

0 = Microphone output enabled, 1 = Microphone output disabled

Comments

See CtEnabDisabMicroOutput()

CtEnabDisabInternPcSpeak

Syntax

WORD CtEnabDisabInternPcSpeak**(WORD value)**

Enables/Disables redirection of the PC internal speaker output to to the GSS mixer.output

Parameters

WORD **value**

0 = Disconnect internal PC speaker,

1 = Connect internal PC speaker

Return Value

1 if ok.

Comments

This can enable the PC internal speaker signal to be mixed with the audio signals of a GSS card (directly, without any mixer volume control).

CtGetEnabDisabInternPcSpeaker

Syntax

```
WORD     CtGetEnabDisabInternPcSpeaker()
```

Returns the state of redirection of the PC speaker.

Parameters

None

Return Value

0 = Internal PC speaker not redirected.

1 = Internal PC speaker redirected

Comments

None

CtSelectInterruptLineNbr

Syntax

`WORD CtSelectInterruptLineNbr(WORD value)`

Selects the interrupt request line used by the audio portion of the GSS hardware.

Parameters

WORD **value**

0 = IRQ3, 1 = IRQ4, 2 = IRQ5, 3 = IRQ7

4 = IRQ10, 5 = IRQ11, 6 = IRQ12, 7 = IRQ15

Return Value

1 if ok.

Comments

The interrupt line is used by OPL3, MMA and telephone hardware. Valid interrupt lines on an XT are IRQ3, IRQ4, IRQ5 and IRQ7. Valid interrupt lines on an AT are IRQ3, IRQ4, IRQ5, IRQ7, IRQ10, IRQ11, IRQ12 and IRQ15.

CtGetInterruptLineNbr

Syntax

```
WORD     CtGetInterruptLineNbr()
```

Returns a number indicating the interrupt line used by the audio portion of the GSS hardware..

Parameters

None

Return Value

0 = IRQ3, 1 = IRQ4, 2 = IRQ5, 3 = IRQ7

4 = IRQ10, 5 = IRQ11, 6 = IRQ12, 7 = IRQ15

Comments

None

CtSelectDMA0ChannelSampChan

CtSelectDMA1ChannelSampChan

Syntax

```
WORD     CtSelectDMA0ChannelSampChan(WORD value)
WORD     CtSelectDMA1ChannelSampChan(WORD value)
```

Allocates DMA channel for the specified MMA sampling channel.

Parameters

WORD **value**

0 = DMA 0
1 = DMA 1
2 = DMA 2
3 = DMA 3

Return Value

1 if ok.

Comments

Only DMA channels 1, 2 and 3 are available on 8-bit bus GSS cards. All listed DMA channels are available on 16-bit bus GSS cards.

CtGetDMA0ChannelSampChan

CtGetDMA1ChannelSampChan

Syntax

```
WORD     CtGetDMA0ChannelSampChan()
WORD     CtGetDMA1ChannelSampChan()
```

Returns a number indicating the DMA channel used by the specified sampling channel.

Parameters

None

Return Value

The sampling channel used.

0 = DMA 0
1 = DMA 1
2 = DMA 2
3 = DMA 3

Comments

None

CtEnabDisabDMA0SampChan

CtEnabDisabDMA1SampChan

Syntax

```
WORD     CtEnabDisabDMA0SampChan(WORD value)
WORD     CtEnabDisabDMA1SampChan(WORD value)
```

Disables or enables use of DMA channel for sampling channel.

Parameters

WORD **value**

0 = disable, 1 = enable

Return Value

1 if ok.

Comments

None

CtGetEnabDisabDMA0SampChan

CtGetEnabDisabDMA1SampChan

Syntax

```
WORD     CtGetEnabDisabDMA0SampChan()
WORD     CtGetEnabDisabDMA1SampChan()
```

Tells if the DMA channel is disabled or enabled for the specified sampling channel.

Parameters

None

Return Value

0 = disabled, 1 = enabled

Comments

None

CtSetRelocationAddress

Syntax

`WORD      CtSetRelocationAddress`**`(value)`**

Set s the base ports address for MMA, OPL3 and control chip.

Parameters

`WORD`   **`value`**

New I/O address. Must be a multiple of 8.

Return Value

`1 if ok.`

Comments

`None`

CtGetRelocationAddress

Syntax

```
WORD     CtGetRelocationAddress()
```

Returns the base port addresses for MMA, OPL3 and control chip.

Parameters

None

Return Value

New base I/O address.

Comments

None

CtSetMixerLevelForFMLeft

CtSetMixerLevelForFMRight
CtSetMixerLevelForLeftSamplePb
CtSetMixerLevelForRightSamplePb
CtSetMixerLevelForAuxLeft
CtSetMixerLevelForAuxRight
CtSetMixerLevelForMicrophone
CtSetMixerLevelForTelephone

Syntax

```
WORD    CtSetMixerLevelForFMLeft(WORD value)
WORD    CtSetMixerLevelForFMRight(WORD value)
WORD    CtSetMixerLevelForLeftSamplePb(WORD value)
WORD    CtSetMixerLevelForRightSamplePb(WORD value)
WORD    CtSetMixerLevelForAuxLeft(WORD value)
WORD    CtSetMixerLevelForAuxRight(WORD value)
WORD    CtSetMixerLevelForMicrophone(WORD value)
WORD    CtSetMixerLevelForTelephone(WORD value)
```

Sets the volume for the specified device

Parameters

WORD **value**

Volume level from 128 to 255 whereis 128 is the minimum, 255 the maximum.

Return Value

1 if ok.

Comments

Writing a value less than 128 will result in a signal with negative polarity and should be avoided because the resulting signal may cancel out another signal of opposite polarity.

CtGetMixerLevelForFMLeft

CtGetMixerLevelForFMRight
CtGetMixerLevelForLeftSamplePb
CtGetMixerLevelForRightSamplePb
CtGetMixerLevelForAuxLeft
CtGetMixerLevelForAuxRight
CtGetMixerLevelForMicrophone
CtGetMixerLevelForTelephone

Syntax

```
WORD     CtGetMixerLevelForFMLeft()
WORD     CtGetMixerLevelForFMRight()
WORD     CtGetMixerLevelForLeftSamplePb()
WORD     CtGetMixerLevelForRightSamplePb()
WORD     CtGetMixerLevelForAuxLeft()
WORD     CtGetMixerLevelForAuxRight()
WORD     CtGetMixerLevelForMicrophone()
WORD     CtGetMixerLevelForTelephone()
```

Returns the volume of the specified device.

Parameters

None

Return Value

Volume level from 128 to 255 whereis 128 is the minimum, 255 the maximum.

Comments

None

CtSetOutputVolumeLeft

CtSetOutputVolumeRight

Syntax

```
WORD      CtSetOutputVolumeLeft(WORD value)
WORD      CtSetOutputVolumeRight(WORD value)
```

Sets the final output volume

Parameters

WORD **value**

Volume level from 0 to 255

Return Value

1 if ok.

Comments

There are actually 64 final volume levels. The driver divides the specified value by 4.

CtGetOutputVolumeLeft

CtGetOutputVolumeRight

Syntax

```
WORD     CtGetOutputVolumeLeft()
WORD     CtGetOutputVolumeRight()
```

Returns the the final output volume

Parameters

None

Return Value

Final output volumefrom 0 to 255

Comments

There are actually 64 final volume levels. The driver multiplies the specified value by 4 in the return value.the return value may not correspond exactly to the value specified with CTSetOutputVolumeXXX().

CtSetOutputBassLevel

CtSetOutputTrebleLevel

Syntax

```
WORD     CtSetOutputBassLevel(WORD value)
WORD     CtSetOutputTrebleLevel(WORD value)
```

Sets the output bass and treble level.

Parameters

WORD **value**

Range from -128 to 127.

Return Value

1 if ok.

Comments

Negative values decreases trebleor bass, positive numbers, increase treble or bass. 0 does not alter sound.

CtGetOutputBassLevel

CtGetOutputTrebleLevel

Syntax

```
WORD     CtGetOutputBassLevel()
WORD     CtGetOutputTrebleLevel()
```

Returns the bass or treble level setting.

Parameters

None

Return Value

Bass or treble setting, from -127 to 127

Comments

Since only 4 bits are actually used in the control Chip, the result obtained can differ with the value written using the CtSetOutputBassLevel() and CtSetOutputTrebleLevel function, due to rounding errors.

CtEnabDisabOutputMuting

Syntax

WORD `CtEnabDisabOutputMuting`**(value)**

Disables or enables output muting.

Parameters

WORD **value**

0 = disable, 1 = enable

Return Value

1 if ok.

Comments

None

CtGetEnabDisabOutputMuting

Syntax

```
WORD     CtGetEnabDisabOutputMuting()
```

Returns a value indicating if output muting is disabled or enabled.

Parameters

None

Return Value

0: disabled, 1: enabled

Comments

None

CtSelectSCSIInterruptNumber

Syntax

`WORD      CtSelectSCSIInterruptNumber`**`(WORD value)`**

Selects an interrupt request line for the SCSI hardware on the Goldcard.

Parameters

`WORD` **`value`**

0 = IRQ3
1 = IRQ4
2 = IRQ5
3 = IRQ7
4 = IRQ10
5 = IRQ11
6 = IRQ12
7 = IRQ15

Return Value

1 if ok.

Comments

**Valid interrupt lines on an XT are IRQ3, IRQ4, IRQ5 and, IRQ7.
Valid interrupt lines on an AT are IRQ3, IRQ4, IRQ5, IRQ7, IRQ10, IRQ11, IRQ12 and IRQ15.**

CtGetSCSIInterruptNumber

Syntax

`WORD      CtGetSCSIInterruptNumber()`

Returns a number indicating the interrupt request line used by the optionnal SCSI hardware on the GSS card.

Parameters

None

Return Value

Interrupt request line:

0 = IRQ3
1 = IRQ4
2 = IRQ5
3 = IRQ7
4 = IRQ10
5 = IRQ11
6 = IRQ12
7 = IRQ15

Comments

None

CtEnabDisabSCSIInterrupt

Syntax

`WORD CtEnabDisabSCSIInterrupt`**(value)**

Disables or enables interrupt from SCSII.

Parameters

`WORD` **value**

0 = disable, 1 = enable

Return Value

1 if ok.

Comments

None

CtEnabDisabSCSIDMA

Syntax

`WORD CtEnabDisabSCSIDMA`**`(value)`**

Disables or enables DMA transfers on optionnal SCSI hardware.

Parameters

`WORD` **`value`**

0 = disable, 1 = enable

Return Value

1 if ok.

Comments

None

CtGetEnabDisabSCSIInterrupt

Syntax

```
WORD     CtGetEnabDisabSCSIInterrupt()
```

Returns 1 if interrupts are enabled on optional SCSI hardware.

Parameters

None

Return Value

0: Interrupts are disabled
1: Interrupts are enabled

Comments

None

CtGetEnabDisabSCSIDMA

Syntax

```
WORD     CtGetEnabDisabSCSIDMA()
```

Returns 1 if DMA transfers are enabled on the optional SCSI hardware.

Parameters

None

Return Value

0: DMA is disabled
1: DMA is enabled

Comments

None

CtSelectSCSIDMAChannel

Syntax

```
WORD     CtSelectSCSIDMAChannel(WORD value)
```

Assigns a DMA channel to the optional SCSI hardware of the GSS Card.

Parameters

WORD **value**

0 = DMA 0
1 = DMA 1
2 = DMA 2
3 = DMA 3

Return Value

1 if ok.

Comments

Valid DMA channels are 0 - 3. Other channel numbers are reserved for future extensions.

CtGetSCSIDMAChannel

Syntax

```
WORD     CtGetSCSIDMAChannel()
```

Returns the number of the DMA channel Assigned to the optional SCSI hardware of the GSS card.

Parameters

None

Return Value

0 = DMA 0
1 = DMA 1
2 = DMA 2
3 = DMA 3

Comments

None

CtSetSCSIRelocationAddress

Syntax

`WORD CtSetSCSIRelocationAddress` **(value)**

Sets the base port address addresses for optional SCSI controller.

Parameters

WORD **value**

New base I/O address divided by 8.
Range from 0 to 127.

Return Value

1 if ok.

Comments

None

CtGetSCSIRelocationAddress

Syntax

```
WORD     CtGetSCSIRelocationAddress()
```

Returns the base port address for SCSI controller.

Parameters

None

Return Value

New base I/O address divided by 8.
Range from 0 to 127.

Comments

None

CtSetHangUpPickUpTelephoneLine

Syntax

`WORD CtSetHangUpPickUpTelephoneLine(WORD value)`

Hangs up or picks up telephone.

Parameters

WORD **value**

0 = Disconnect telephone line,
1 = Connect telephone line

Return Value

1 if ok.

Comments

None

CtGetHangUpPickUpTelephoneLine

Syntax

```
WORD     CtGetHangUpPickUpTelephoneLine()
```

Returns a value telling if the telephone line is on-hook or off-hook.

Parameters

None

Return Value

0: telephone line is on-hook (not connected)
1: telephone line is off-hook (connected)

Comments

None

CtSelectOutputSources

Syntax

`WORD      CtSelectOutputSources`**`(value)`**

Selects final output mixing redirection.

Parameters

`WORD      `**`value`**

0 = left mixer channel to left output & right mixer channel to right output,
1 = left mixer channel to both left and right outputs,
2 = right mixer channel to both left and right outputs.

Return Value

1 if ok.

Comments

On the Adlib GSS cards, mixing and volume control is performed in two stages. First, all sources are sent to a stereo mixer. Then, the stereo output of the mixer is fed into the final volume control circuitry. The final left and right outputs can be mixed in the fashion described above.

CtGetOutputSources

Syntax

`WORD     CtGetOutputSources()`

Returns the final mixer redirection mode.

Parameters

None

Return Value

0 = left mixer channel to left output & right mixer channel to right output,
1 = left mixer channel to both left and right outputs,
2 = right mixer channel to both left and right outputs.

Comments

None

CtSelectOutputMode

Syntax

`WORD CtSelectOutputMode`**`(value)`**

Controls the effect applied to the final output .

Parameters

`WORD` **`value`**

0 = Forced mono,
1 = linear stereo,
2 = pseudo stereo,
3 = spatial stereo.

Return value

1 if ok.

Comments

Linear stereo is ordinary, with no effects added. The spatial and pseudo-stereo effects will be useful primarily when the original source is monophonic.

CtGetOutputMode

Syntax

```
WORD     CtGetOutputMode()
```

Returns the effect applied to the final output .

Parameters

None

Return value

0 = Forced mono,
1 = linear stereo,
2 = pseudo stereo,
3 = spatial stereo.

Comments

None

GetControlRegister

Syntax

`WORD GetControlRegister`**(reg)**

Returns value stored on register 'reg' of Ad Lib Control Chip.

Parameters

`int` **reg**

Which register to read from. If reg is -1, this reads the control chip status register.

Return value

Returns the WORD at the register position.

Comments

None

CtGetBoardIdentificationCode

Syntax

```
WORD     CtGetBoardIdentificationCode()
```

Returns the board identification code.

Parameters

None

Return value

Board identification code:

0 - 8 bit bus GSS card,
1 - 16 bit bus GSS card
2 - (to be defined)

Comments

None

CtGetBoardOptions

Syntax

```
WORD     CtGetBoardOptions()
```

Returns a bit pattern indicating the options present on boardpresent

Parameters

None

Return value

Bit 0-3 (0 = not present, 1 = installed)

bit 0 - Telephone,
bit 1 - Surround,
bit 2 - SCSI,
bit 3 - Currently unused

Comments

None

CtGetControllerStatus

Syntax

`WORD CtGetControllerStatus()`

Returns the interrupt controller status.

Parameters

None

Return value

bit 0 - equals 1 when an OPL3 interrupt is pending
bit 1 - equals 1 when an MMA interrupt is pending
bit 2 - equals 1 when an telephone interrupt is pending
bit 3 - equals 1 when a SCSI interrupt is pending
bit 6 - equals 1 when the Control Chip is currently occupied writing a value to the Mixer Chip or the Volume Control Chip.
bit 7 Set to 1 when the Control Chip is busy writing its internal registers to the external EEPROM chip.
This bit must be polled after activating the "Store configuration" sequence to make sure that the Control Chip is free to proceed with another operation.

Comments

Bit 7 and Bit 6 are polled by all set functions, prior to writing to the registers, to make sure that the Control Chip is free to proceed with another operation.

CtGetRingTelephoneStatus

Syntax

`WORD CtGetRingTelephoneStatus()`

Gets telephone status.

Parameters

None

Return value

bit 0: "Ring signal" (0 = no ring, 1 = ring)

Comments

None

CtGetInterruptRoutine

Syntax

```
WORD     CtGetInterruptRoutine()
```

This routine returns the corresponding interrupt number associated with the interrupt request line used by the audio section.

Parameters

None

Return value

Corresponding interrupt number

Comments

Useful utility mostly used when setting interrupt vectors.

CtGetGoldCardPresence

Syntax

```
WORD     CtGetGoldCardPresence()
```

Checks for GSS card presence.

Parameters

None

Return value

1 if any GSS card is found. 0 if no GSS card is found.

Comments

None

2.3

Introduction

The Ad Lib GSS FM Synthesis Driver offers services to access features of the OPL3 FM Chip.

Voice Allocation Structure

The OPL3 chip contains 36 operators which can be combined in various ways to create 1-, 2- or 4-operator voices. (You may wish to refer to the "FM Driver Voices" table on the next page for the purposes of this discussion.)

The 4-operator voices offer the richest sound. Up to six 4-operator voices can be used simultaneously. In the FM Driver, the 4-operator voices are numbered 0, 2, 4, 6, 8 and 10. By default, all six 4-operator voices are enabled. They may be selectively disabled, thus creating two 2-operator voices.

In the FM Driver, when 4-operator voice *x* is disabled, the two 2-operator voices are numbered *x* and *x*+1. For example, if 4-operator voice #2 was disabled, the resulting 2-operator voices will be numbered 2 and 3.

Use Set4OpMaskOPL3() to determine the grouping of the units in either 2 operator or 4 operator voices.

Six of the chip's operators can only be used as three 2-operator voices. These three voices are numbered 12, 13 and 14.

The configuration of the remaining 6 operators depends on whether the card is in melodic or percussive mode. In melodic mode, these 6 operators are configured as three 2-operator voices: driver voice numbers 15, 16 and 18. In percussive mode, the 6 operators are used to create one 2-operator voice (the bass drum) and four 1-operator voices (the remaining drum sounds). The percussive voices are driver voice numbers 15 through 19.

Use SetPercModeOPL3() to configure this section in the melodic or percussive mode.

4 operator voice number	**2 operator voice number**	**Percussive voice number**
0	0, 1	-
2	2,3	-
4	4,5	-
6	6,7	-
8	8,9	-
10	10,11	-
-	12	-
-	13	-
-	14	-
-	15	15 (BD)
-	16	16 (HH)
-	-	17 (SD)
-	18	18 (TOM)
-	-	19 (CYMB)

FM Driver Voices

Function Directory

The following section is an alphabetically arranged definition of all the functions available in the FM Synthesis Driver.

InitFMDriver

Syntax

void InitFMTimer(void)

Initializes the FM Chip.

Parameters

None

Comments

After initialization, percussion voices are available and all 4 op-voices are enabled.

LeftRightOPL3

Syntax

void LeftRightOPL3(voiceNum, leftRight)

Modifies the stereo position of the voice.

Parameters

int voiceNums

VoiceNumber between 0 and 19.

int leftRight

Position of the specified voice:

0: Center.
1: Left.
2: Right.

LevelOPL3

Syntax

void LevelOPL3(voiceNum, level)

Specify the individual volume for a voice.

Parameters

int voiceNum

Voice number between 0 and 19

int level

Volume for the voice.
This in an integer number between 0 and 127.
Volume scaling is linear.

Comments

The volume is scaled linearly by the driver software.

NoteOffOPL3

Syntax

void NoteOffOPL3(voiceNum)

Starts the decay of the timbre currently playing on the voice.

Parameters

int voiceNum

VoiceNumber between 0 and 19.

NoteOnOPL3

Syntax

```
void NoteOnOPL3(voiceNum, note)
```

Starts playing a note on the specified voice.

Parameters

int voiceNum

VoiceNumber between 0 and 19.

int note

MIDI value for the note played, in the range 12-107.

Comments

If a note is already playing on the specified voice, the frequency of the voice will be modified. However, the attack for the timbre will not be heard. To reattack the timbre on the specified voice, a NoteOffOPL3 must be issued.

PitchbendOPL3

Syntax

void PitchBendOPL3(voiceNum, pitchBend)

Modifies the pitch bend scaling factor for the melodic voice.

Parameters

int voiceNum

Melodic voiceNumber between 0 and 15.

WORD pitchBend

Pitch bend scaling factor within the range set in SetGlobalOPL3().

The pitch bend scaling factor is a 14 bit unsigned value. 0 is the maximum negative pitch bend, 0x2000 is no bend and 0x3FFF is the maximum positive pitch bend.

Comments

Percussive voices cannot be bent.

PresetOPL3

Syntax

void PresetOPL3(voiceNum, timbrePtr)

Assigns a patch to the specified voice.

Parameters

int voiceNum

voiceNumber between 0 and 19

struct TIMBRE *timbrePtr

pointer to a description (28 bytes) of the patch assigned to the voice.

Comments

If a 4 operator description is sent to a 2-op voice, only the first two operators are considered.

Appendix A: FM Patch format further describes the structure pointed to by timbrePtr.

QuitFMDriver

Syntax

void QuitFMDriver()

Resets the FM chip in the compatible mode.

Parameters

None.

Comments

This should be called by all applications prior to leaving, in order to put the OPL3 chip back in the Ad Lib compatible mode.

Set4OpMaskOPL3

Syntax

void Set4OpMaskOPL3(mask)

Enables or disables 4-op voices.

Parameters

WORD mask

Bit mask of enabled 4-op voices (in bits 0-5).

Bits 0-5 of mask specify whether the corresponding voice is in 4-op mode (bit set to 1) or in 2-op mode (bit cleared to 0).

Bit 0 corresponds to voice 0 (0-1 in 2 op), bit 1 to voice 2 (2-3 in 2 op) etc. (See to table 1 in the Voice Allocation section of this document).

Comments

There is a maximum of 6 4-op voices.

SetGlobalOPL3

Syntax

```
void SetGlobalOPL3   (noteSelectEnable, amplitudeModEnable,
                       vibDepthEnable, pitchBendRange)
```

Modifies global operating parameters of the OPL3.

Parameters

BOOL `noteSelectEnable`

For future use. Set to 0 for now.

BOOL `amplitudeModEnable`

When non-zero, enables amplitude modulation for all timbres that have an amplitude modulation defined.

BOOL `vibDepthEnable`

When non-zero, enables vibrato for all timbres that have a vibrato depth defined.

int `pitchBendRange`

Range of the pitch bend in semitones.
Integer between 0-12.

SetPercModeOPL3

Syntax

void SetPercModeOPL3(newState)

Sets the OPL3 in melodic or percussive mode.

Parameters

BOOL newState

True for percussive mode, false for melodic mode.

Comments

If newState is true, disables melodic voices 15-18 and enables percussive voices 15-19 instead.

If newState is false, melodic voices 15-18 are enabled in place of percussive voices 15-19.

The Wave Driver is a high level software interface to the sampling hardware of the GSS Card. Its interface is inspired by the Microsoft Multimedia Wave Driver specifications. But in order to support the target hardware and software more efficiently, some adaptations were necessary. The main differences are:

The support of ADPCM as well as PCM formats.

The support of a stereo sample format.

The control of multiple transfer modes from memory to hardware (polling, interrupt, DMA). (This implies an extension of the WaveFormat structure to include the new parameters.)

The use of a callback function as a message-passing mechanism between the application and the driver during waveform recording and playback.

Some syntactical differences were introduced in the naming of functions and structures, in order to respect the naming conventions already in use in other modules.

The Wave Driver allows to queue multible memory blocks of data for playback, in interrupt or DMA mode. The blocks are returned to the application once thay have been processed, by the means of a callback mechanism. The callback routine is specified by the application in the WaveOutOpen() or WaveInOpen() calls.

DOS Wave Driver Function Directory

The following section is definition of all the functions available in the Wave Driver.

InitWaveDriver

Syntax

```
void  InitWaveDriver()
```

Initializes the wave driver.
It is to be called only once by the application.

Parameters

None

Return value

None

Comments

You must call InitControlDriver() prior to this calling this function.

QuitWaveDriver

Syntax

Word QuitWaveDriver ()

This function resets the driver.

IMPORTANT: This must be called before returning to the DOS.

Parameters

None

Return value

This function should be called before CloseControlDriver().

WaveInAddBuffer

Syntax

Word WaveInAddBuffer (hWaveIn, lpWaveInHdr, wSize)

Sends a buffer to a waveform input device. When the buffer is full, the application is notified.

Parameters

HWaveIn hWaveIn

Specifies a handle to the waveform device which is to receive the buffer.

LpWaveHdr lpWaveInHdr

Specifies a far pointer to a **WaveHdr** structure that identifies the buffer.

Word wSize

Specifies the size of the WaveHdr structure.

Return value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid

WaveInClose

Syntax

Word **WaveInClose**(hWaveIn)

Closes the specified waveform input device.

Parameters

HWaveIn hWaveIn

Specifies a handle to the waveform input device to be closed.
If the function is successful, the handle is no longer valid after this call.

Return value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid

WERR_STILLPLAYING

There are still buffers in the queue

Comments

If there are input buffers that have been sent with **WaveInAddBuffer**, and have not been used, the close operation will fail. Call in **WaveInReset** to mark all pending buffers as done.

WaveInGetNumDevs

Syntax

Word WaveInGetNumDevs()

Retrieves the number of waveform input devices present in the system.

Parameters

None

Returns value

Returns the number of waveform input devices in the system.

WaveInOpen

Syntax

```
Word  WaveInOpen      (lphWaveIn, wDeviceID, lpFormat, dwCallBack,
                       dwCallBackData, dwFlags)
```

Opens the specified waveform input device for recording.

Parameters

HWaveIn far *lpWaveIn

Specifies a pointer to a HWaveIn handle. This location is filled with a handle identifying the opened waveform input device. Use this handle to identify the device when calling other waveform input functions.

This parameter may be NULL if the WAVE_FORMAT_QUERY flag is specified for the dwFlags.

Word wDeviceID

Identifies the waveform input device that is to be opened.

LpWaveFormat lpFormat

Specifies a far pointer to a WaveFormat data structure that identifies the desired format for recording the waveform data.

```
int (far * dwCallBack)      (HWaveIn dev, LpWaveHdr block,
                         DWord dwCallBackData)
```

Specifies the address of a callback function. The callback function is called by the driver during recording to process messages related to the progress of the recording.

Specify NULL for this parameter if no callback is desired.

DWord dwCallbackData

Specifies 32 bits of user defined data that is passed to the callback function.

DWord dwFlags

Specifies flags for opening the device.

WAVE_FORMAT_QUERY
If this flag is specified, the device driver will determine if it supports the given format, but will not actually open the device.

Return value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_ALLOCATED
Specified resource is already allocated.

WERR_BADDEVICEID

Specified device is out of range.

WERR_BADTRANSFERMODE

Specified transfer mode is unsupported or unavailable.

WERR_STEREOBADCHANNEL

Invalid channel for stereo output (stereo output is only possible on channel 0).

WERR_STEREONEED2FREECHNL

Could not allocate two consecutive channels for stereo output.

WERR_UNSUPPORTEDFORMAT

Attempted to open with an unsupported wave format.
(This error code not currently supported).

Comments

Use **WaveInGetNumDevs** to determine the number of input devices present in the system. The device ID specified by wDeviceID varies from 0 to one less than the specified number of devices present.

The application should make sure that the transfer mode specified in the lpFormat variable is supported by the hardware configuration. The wave driver does NOT validate a DMA or interrupt transfer. This can be done by calling the appropriate functions in the control chip driver.

WaveInReset

Syntax

Word **WaveInReset**(hWaveIn)

Stops input on a given waveform device and resets the current position to 0. All pending buffers are marked as done.

Parameters

HWaveIn hWaveIn

Specifies a handle to the input device that is to be reset.

Return value

Returns zero if the function is successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid.

WaveInStart

Syntax

Word WaveInStart(hWaveIn)

Starts input on a given waveform input device.

Parameters

HWaveIn hWaveIn

Specifies a handle to the input device to be started.

Return value

Returns zero if the function is successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid.

Comments

Buffers are returned to the client when full or when WaveInReset is called (the dwBytesRecorded field in the header will contain the actual length of the data). If there are no buffers available, the data is thrown away without notification to the client and input will continue.

Calling this function when input is already started will have no effect and 0 will be returned.

WaveOutBreakLoop

Syntax

Word **WaveOutReset**(hWaveOut)

Breaks a loop on a given waveform device and allows playback to continue with the next block in the driver list.

Parameters

HWaveOut hWaveOut

Specifies a handle to the waveform output device to receive the command.

Return value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid

Comments

Waveform looping is controlled by the dwLoops and dwFlags fields in the **WaveHdr** structures passed to the device with **WaveOutWrite.** Use the **WHDR_BEGINLOOP** and **WHDR_ENDLOOP** flags in the **WaveHdr** structure to specify the beginning and ending data blocks for looping. To loop on a single block, specify both flags for the same block. Use the dwLoops field in the **WaveHdr** structure for the first block in the loop to specify the number of loops.

Calling this function when nothing is playing or looping will have no effect and 0 will be returned.

WaveOutClose

Syntax

Word **WaveOutClose**(hWaveOut)

This function closes the specified waveform output device.

Parameters

HWaveOut hWaveOut

Specifies a handle to the waveform output device to be closed. If the function is successful, the handle is no longer valid after the call.

Return value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid.

WERR_STILLPLAYING

There are still buffers in the device queue.

Comments

If the device is still playing a waveform, the close operation will fail. Use **WaveOutReset** to terminate playback before calling **WaveOutClose**.

WaveOutGetNumDevs

Syntax

Word WaveOutGetNumDevs()

Retrieves the number of waveform output devices present in the system.

Parameters

None

Returns value

Returns the number of waveform output devices in the system.

WaveOutGetVolume

Syntax

Word WaveOutGetVolume(hWaveOut, lpdwVolume)

This function queries the current volume setting of a waveform output device.

Parameters

HWaveOut hWaveOut

Identifies the wave output device.

LPDWord lpdwVolume

Specifies a far pointer to a location that will be filled with the current volume setting.

The high-order word contains the left channel volume and the low-order word contains the right channel volume.

If a device does not support volume control on both left and right channels (if the device is opened in mono), only the right channel value is used.

A value of 0xFFFF specifies full volume and a value of 0x0000 is silence.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid.

Comments

Volume control is supported on the left and right channels only if the device was opened specifying 2 in the nChannel field of the **lpWaveFormat** structure of **WaveInOpen.**

WaveOutOpen

Syntax

Word **WaveOutOpen** (lphWaveOut, wDeviceId, lpFormat, dwCallBack, dwCallBackData, dwFlags)

Opens a specified waveform output device for playback.

Parameters

HWaveOut far *lphWaveOut

Specifies a pointer to an HWAVEOUT handle. This location is filled with a handle identifying the opened waveform output device.
Use the handle to identify the device when calling other wave ouput functions. This parameter may be NULL if WAVE_FORMAT_QUERY is specified in dwFlags.

Word wDeviceID

Identifies the waveform output device that is to be opened.

LpWaveFormat lpFormat

Specifies a pointer to a WaveFormat structure that identifies the format of the waveform that will be sent to the output device.
The WaveFormat structure is also used to specify the "mode" by which the data will be sent to the hardware (WAVE_TRANF_POLLING, WAVE_TRANSF_INTERRUPT, WAVE_TRANSF_ DMA).

int (far * dwCallBack) (**HWaveOut** dev, **LpWaveHdr** block, **DWord** dwCallBackData)

Specifies the address of a callback function. The callback function is called by the driver during playback to process messages related to the progress of the playback.

Specify NULL for this parameter if no callback is desired.

DWord dwCallbackData

Specifies 32 bits of user defined data that is passed to the callback.

DWord dwFlags

Specifies flags for opening the device.

WAVE_FORMAT_QUERY

If this flag is specified, the device driver will determine if it supports the given format, but will not actually open the device.

Return value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_ALLOCATED

Specified resource is already allocated.

WERR_BADDEVICEID

Specified device is out of range.

WERR_BADTRANSFERMODE

Specified transfer mode is unsupported or unavailable.

WERR_STEREOBADCHANNEL

Invalid channel for stereo output (stereo output is only possible on channel 0).

WERR_STEREONEED2FREECHNL

Could not allocate two consecutive channels for stereo output.

WERR_UNSUPPORTEDFORMAT

Attempted to open with an unsupported wave format.
(This error code not currently supported).

Comments

Use **WaveOutGetNumDevs** to determine the number of output devices present in the system. The device ID specified by wDeviceID varies from 0 to one less than the specified number of devices present.

The application should make sure that the transfer mode specified in the **lpFormat** structure is supported by the hardware configuration. The wave driver does NOT validate a DMA or interrupt transfer. This can be made by calling the appropriate functions in the control chip driver. The wave driver uses information stored in the control chip to determine which interrupt and which DMA line it will use.

WaveOutPause

Syntax

Word **WaveOutPause**(hWaveOut)

Pauses playback on a specified waveform output device. The current playback position is saved. Use **WaveOutRestart** to resume playback from the current playback position.

Parameters

HWaveOut hWaveOut

Specifies a handle to the waveform output device to be paused.

Return value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid.

Comments

Calling this function when output is already paused will have no effect and 0 will be returned.

WaveOutReset

Syntax

Word **WaveOutReset**(hWaveOut)

Stops playback on a given waveform output device and resets the current position to 0. All pending playback buffers are marked as done.

Parameters

HWaveOut hWaveOut

Specifies a handle to the waveform output device that is to be reset.

Return value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid.

WaveOutRestart

Syntax

Word **WaveOutRestart**(hWaveOut)

This function restarts a paused waveform output device.

Parameters

HWaveOut hWaveOut

Specifies a handle to the waveform output device that is to be restarted.

Return value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid.

Comments

Calling this function when the output is not paused will have no effect and 0 will be returned.

WaveOutSetLeftRight

Syntax

Word `WaveOutSetLeftRight(hWaveOut, leftRight)`

Selects which sides the output will be directed to.

Parameters

HWaveOut `hWaveOut`

Specifies a handle to the waveform output device that is to be restarted.

Word `leftRight`

Flags specifying the output direction:

WAVE_STEREO_LEFT
WAVE_STEREO_CENTER
WAVE_STEREO_RIGHT

Return value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid

Comments

This function is useful only when the channel is monophonic. Stereophonic channels are always output left and right.

WaveOutSetVolume

Syntax

Word WaveOutSetVolume(hWaveOut, dwVolume)

Sets the volume of a waveform output device.

Parameters

HWaveOut hWaveOut

Identifies the wave output device.

Dword dwVolume

Specifies the volume setting.

The high-order word contains the left channel volume and the low-order word contains the right channel volume.

If a device does not support volume control on both left and right channels (if the device is opened in mono), only the right channel value is used.

A value of 0xFFFF specifies full volume and a value of 0x0000 is silence.

Return value

Returns zero if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid.

Comments

Volume control is supported on the left and right channels only if the device was opened specifying 2 in the nChannel field of the **lpWaveFormat** structure specified in **WaveOutOpen.**

Note that this controls output volume only.

WaveOutWrite

Syntax

Word **WaveOutWrite**(hWaveOut, lpWaveOutHdr, wSize)

Sends a data block to the specified waveform output device.

Parameters

HWaveOut hWaveOut

Specifies a handle to the waveform device that the data is to be sent to.

LpWaveHdr lpWaveOutHdr

Specifies a far pointer to a **WaveHdr** structure containing information about the data block.

Word wSize

Specifies the size of the **WaveHdr** structure.

Return value

Returns 0 if the function was successful. Otherwise, it returns an error code. Possible error codes are:

WERR_INVALIDHANDLE

Specified device handle is invalid.

Comments

Unless playback is paused by **WaveoutPause**, playback begins when the first data block is sent to the device.

When writing to a device opened using the WAVE_TRANSF_POLLING mode, control will be returned to the application only when the buffer has been completely played. Using this transfer mode, wave output must be paused with **WaveOutPause** prior to calling **WaveOutWrite** if the application must write more than one buffer.

The GSS cards offers to developers 5 multi-purpose timers. They are physically located on two different chips but their implementation are similar.

All timers have their own base clock (time resolution) and counter size (maximum period). The controls available for all timers are:

- o Write access in their register of different count values (divider).
- o Stop and start (decrementing the initial stored count until it reach zero and re-writing the original count, again and again).
- o Enable/disable interrupts to occur on zero count crossing.
- o Read the interrupt status (access on the zero count crossing).

Some differences exist and need to be noticed:

- o The timer 2 from the MMA chip is the only timer whose current count can be read.
- o Yamaha in its own documentation use the terms timer 1 and 2 for the timers physically located in the OPL3 chip and timers located in the MMA chip.
- o A base counter (another timer) is used in the MMA chip as an input clock for the timers 1 and 2. Those last two timers are decremented each time the base counter reaches zero. This means that the software must initialized the base counter with an appropriate value then the timer 1 or 2.
- o OPL3 timers are NOT available on Level 1 implementation of the drivers.

Here is a table that illustrates the specifications of all timers:

	OPL3 chip		MMA chip			
	Tim. 1	Tim. 2	Tim. 0	B. C.	Tim. 1	Tim. 2
time resolution in µsec	80	320	1.89	1.89	1.89	1.89
max period length in msec	20.4	81.6	123.83	7.738	116.07	507116
counter size in bits	8	8	16	12	4+12	16+12

Table 1: Hardware specifications of timers

Remember that the MMA timer 1 and 2 are combined with the MMA base counter and that their combined specifications gives for the timer 1 a size of 16 bits and for the timer 2 a size of 28 bits.

The timer's function can be access directly or by the TimerDrvService functions which is a dispatcher.

Each timer function is presented in the following pages.

LoadStartOPL3Timer1

LoadStartOPL3Timer2
LoadStartMMATimer0
LoadStartMMATimer1
LoadStartMMATimer2

Syntax

```
WORD     LoadStartOPL3Timer1(void)
WORD     LoadStartOPL3Timer2(void)
WORD     LoadStartMMATimer0(void)
WORD     LoadStartMMATimer1(void)
WORD     LoadStartMMATimer2(void)
```

This will load the physical counter with the count associated and start the counter.

Parameters

None

Return value

TIMER_NO_ERROR

If the function was sucessful.

TIMER_FUNCTION_ERROR

If a problem occured when loading.

Comments

None

StopOPL3Timer1

StopOPL3Timer2
StopMMATimer0
StopMMATimer1
StopMMATimer2

Syntax

```
WORD     StopOPL3Timer1(void)
WORD     StopOPL3Timer2(void)
WORD     StopMMATimer0(void)
WORD     StopMMATimer1(void)
WORD     StopMMATimer2(void)
```

Stop the associated timer.

Parameters

None

Return value

TIMER_NO_ERROR

If the function was sucessful.

TIMER_FUNCTION_ERROR

If a problem occured when stoping.

Comments

None

SetOPL3Timer1Counter

SetOPL3Timer2Counter
SetMMATimer0Counter
SetMMATimer1Counter
SetMMATimer2Counter
SetMMABaseCounterCounter

Syntax

```
WORD     SetOPL3Timer1Counter(BYTE count)
WORD     SetOPL3Timer2Counter(BYTE count)
WORD     SetMMATimer0Counter(WORD count)
WORD     SetMMATimer1Counter(BYTE count)
WORD     SetMMATimer2Counter(WORD count)
WORD     SetMMABaseCounterCounter(WORD count)
```

Set the OPL3 and MMA timer with the count value. Base clock periods are the following:

OPL3Timer1: 79.9682 us
OPL3Timer2: 319.873 us
MMATimer0: 1.89 us
MMATimer1: 1.89 us
MMATimer2: 1.89 us
MMATimerBaseCounter: 1.89 us

See table xx for more information the capacity of each timer.

Parameters

BYTE **count**
WORD **count**

The parameters count specified the number of cycle the timer is supposed to do. Depending of timer count is BYTE or WORD parameter.

Return value

TIMER_NO_ERROR

If the function was successful.

TIMER_FUNCTION_ERROR

If a problem occured when setting.

Comments

It is important to check the table xx because each timer don't use all of the bits in the count parameters.

SetOPL3Timer1Period

SetOPL3Timer2Period
SetMMATimer0Period
SetMMATimer1Period
SetMMATimer2Period
SetMMABaseCounterPeriod

Syntax

```
WORD     SetOPL3Timer1Period(DWORD lPeriod)
WORD     SetOPL3Timer2Period(DWORD lPeriod)
WORD     SetMMATimer0Period(DWORD lPeriod)
WORD     SetMMATimer1Period(DWORD lPeriod)
WORD     SetMMATimer2Period(DWORD lPeriod)
WORD     SetMMABaseCounterPeriod(DWORD lPeriod)
```

This set of functions offer another way to set the count of a timer. The period of a cycle is passed instead of passing the divider. It becomes more easy for the programmer to think in terms of period rather than in terms of a divider to associate with the required period.

Parameters

DWORD lPeriod

Period in usec to be passed to the timer.

Return value

TIMER_NO_ERROR

If the function was sucessful.

TIMER_FUNCTION_ERROR

If a problem occured when setting.

Comments

Check the table xx to be sure to respect the maximum capacity of the timer.
The period will be round to the precision of the timer.

EnableOPL3Timer1

EnableOPL3Timer2
EnableMMATimer0
EnableMMATimer1
EnableMMATimer2

Syntax

```
WORD     EnableOPL3Timer1(void)
WORD     EnableOPL3Timer2(void)
WORD     EnableMMATimer0(void)
WORD     EnableMMATimer1(void)
WORD     EnableMMATimer2(void)
```

This will set the mask bit associated with the timer interrupt.

Parameters

None

Return value

TIMER_NO_ERROR

If the function was sucessful.

TIMER_FUNCTION_ERROR

If a problem occured when enabling.

Comments

None

DisableOPL3Timer1

DisableOPL3Timer2
DisableMMATimer0
DisableMMATimer1
DisableMMATimer2

Syntax

```
WORD     DisableOPL3Timer1(void)
WORD     DisableOPL3Timer2(void)
WORD     DisableMMATimer0(void)
WORD     DisableMMATimer1(void)
WORD     DisableMMATimer2(void)
```

This will reset the mask bit associated with the timer interrupt.

Parameters

None

Return value

TIMER_NO_ERROR

If the function was sucessful.

TIMER_FUNCTION_ERROR

If a problem occured when disabling.

Comments

None

GetOPL3TimerIntStatus

GetMMATimerIntStatus

Syntax

```
WORD     GetOPL3TimerIntStatus(void)
WORD     GetMMATimerIntStatus(void)
```

These functions will return the state of timer interrupt of the OPL3 and MMA.

Parameters

None

Return value

OPL3

return 0 if no timer has interrupted.
return 2 if timer 1 has interrupted.
return 1 if timer 2 has interrupted.
return 3 if timer 1 and 2 has interrupted.MMA
return 0 if no timer has interrupted.
return 1 if timer 0 has interrupted.
return 2 if timer 1 has interrupted.
return 4 if tmer 2 has interrupted.
or any combination of 1,2 and 4 if multiple timer has interrupted.

Comments

The MMA chip has a special behavior: it will reset the interrupt bit after a status register reading. Note that this routine is automatically called by the main interrupt handler from the Control Chip Driver. Using GetOPL3TimerIntStatus will not reset the OPL3 status register bits.

AssignOPL3Timer1IntService

AssignOPL3Timer2IntService
AssignMMATimer0IntService
AssignMMATimer1IntService
AssignMMATimer2IntService

Syntax

```
WORD     AssignOPL3Timer1IntService(void (*function)(void))
WORD     AssignOPL3Timer2IntService(void (*function)(void))
WORD     AssignMMATimer0IntService(void (*function)(void))
WORD     AssignMMATimer1IntService(void (*function)(void))
WORD     AssignMMATimer2IntService(void (*function)(void))
```

Use by applications to assign their callback function on a specific interrupt.

Parameters

void **(*function)(void)**

The parameter is the callback prototype.

Return value

TIMER_NO_ERROR

If the function was sucessful.

TIMER_FUNCTION_ERROR

If a problem occured with the assign procedure.

Comments

The application user must specifie a callback routine that will automatically be called when the interrupt occurs. This callback function must be very short to execute because this is a timer interrupt that may occurs at a very high rate. At initialisation the default service hooked on each timer interrupt is a local DoNothing function that must be replaced by the application user.

RestoreOPL3Timer1IntService

RestoreOPL3Timer2IntService
RestoreMMATimer0IntService
RestoreMMATimer1IntService
RestoreMMATimer2IntService

Syntax

```
WORD     RestoreOPL3Timer1IntService(void)
WORD     RestoreOPL3Timer2IntService(void)
WORD     RestoreMMATimer0IntService(void)
WORD     RestoreMMATimer1IntService(void)
WORD     RestoreMMATimer2IntService(void)
```

Use by applications to remove their callback function from the interrupt process.

Parameters

None

Return value

TIMER_NO_ERROR

If the function was sucessful.

TIMER_FUNCTION_ERROR

If a problem occured with the restore procedure.

Comments

None

ExecOPL3Timer1IntService

ExecOPL3Timer2IntService
ExecMMATimer0IntService
ExecMMATimer1IntService
ExecMMATimer2IntService

Syntax

```
void     ExecOPL3Timer1IntService(void)
void     ExecOPL3Timer2IntService(void)
void     ExecMMATimer0IntService(void)
void     ExecMMATimer1IntService(void)
void     ExecMMATimer2IntService(void)
```

Those routines will execute the function associated with each interrupt.

Parameters

None

Return value
None

Comments
None

ResetOPL3LastTimerInt

Syntax

`WORD ResetOPL3LastTimerInt`**(void)**

This will reset the IRQ signal generated by timers 1 and 2.

Parameters

None

Return value

TIMER_NO_ERROR

If the function was sucessful.

TIMER_FUNCTION_ERROR

If a problem occured with the reset procedure.

Comments

This function does not exist for the MMA because the MMA clear the status after each reading of the status register.

AllocateOPL3Timer1

AllocateOPL3Timer2
AllocateMMATimer0
AllocateMMATimer1
AllocateMMATimer2
AllocateMMABaseCounter

Syntax

```
WORD     AllocateOPL3Timer1(void)
WORD     AllocateOPL3Timer2(void)
WORD     AllocateMMATimer0(void)
WORD     AllocateMMATimer1(void)
WORD     AllocateMMATimer2(void)
WORD     AllocateMMABaseCounter(void)
```

This procedure will reserve and from then denied any external application access to this timer.

Parameters

None

Return value

1: if available
0: if not available

Comments

Any application who wants to use the service of any timers should ask the Timer Driver for its disponibility using an allocation routine. The application should free the timer after use.

FreeOPL3Timer1

FreeOPL3Timer2
FreeMMATimer0
FreeMMATimer1
FreeMMATimer2
FreeMMABaseCounter

Syntax

```
WORD      FreeOPL3Timer1(void)
WORD      FreeOPL3Timer2(void)
WORD      FreeMMATimer0(void)
WORD      FreeMMATimer1(void)
WORD      FreeMMATimer2(void)
WORD      FreeMMABaseCounter(void)
```

Free the the timer.

Parameters

None

Return value

1: if operation succed
0: if operation not succed

Comments

None

GetMMATimer2Content

Syntax

```
WORD     GetMMATimer2Content(void)
```

This routine returns the content of the MMA timer 2.

Parameters

None

Return value

16 bit content of MMA timer 2

Comments

This is the only timer that can be read. These timers respect the specification of Windows Multi-Media.

GetOPL3Timer1Caps

GetOPL3Timer2Caps
GetMMATimer0Caps
GetMMATimer1Caps
GetMMATimer2Caps

Syntax

```
WORD     GetOPL3Timer1Caps
         (DWORD far *lPeriodMin, DWORD far *lPeriodMax)
WORD     GetOPL3Timer2Caps
         (DWORD far *lPeriodMin, DWORD far *lPeriodMax)
WORD     GetMMATimer0Caps
         (DWORD far *lPeriodMin, DWORD far *lPeriodMax)
WORD     GetMMATimer1Caps
         (DWORD far *lPeriodMin, DWORD far *lPeriodMax)
WORD     GetMMATimer2Caps
         (DWORD far *lPeriodMin, DWORD far *lPeriodMax)
```

Used by external modules to query the driver on physical limits of each timer. It returns the minimum and maximum period covered by the timer in micro seconds.

Parameters

DWORD far *lPeriodMin
DWORD far *lPeriodMax

These two address will receve the minimum and the maximum period capacity respectively of the timer.

Return value

TIMER_NO_ERROR

If the function was sucessful.

TIMER_FUNCTION_ERROR

If a problem occured with the procedure.

Comments

None

InitTimerDriver

Syntax

`WORD InitTimerDriver`**`(WORD base)`**

This procedure initialize the Timer Driver structure with default values. This procedure should be used the first time the driver is called.

Parameters

`WORD base`

Actual address of the Ad Lib control chip.

Return value

`TIMER_NO_ERROR`

If the function was sucessful.

`TIMER_FUNCTION_ERROR`

If a problem occured with the procedure.

Comments

None

TimerDrvService

Syntax

```
WORD     far   TimerDrvService(WORD segm, WORD offs)
```

Entry point for the AdLib timer dispatcher. The segment and offset of the argument structure are passed as argument.

Parameters

WORD segm

WORD offs

These two parameters specify the segment and the offset of the following structure which is used to pass parameters to the TimerDrvService routine.

struct TimerArgum {

```
WORD     controlID;          which service to be used
WORD     timerDv;            on which timer
DWORD    param;              optionnal based on service used
DWORD    param2;             optionnal based on service used
void     (interrupt far *function)();  optionnal based on service used }
```

Return value

Service result if any.

Comments

See TimerDrv.h for all ID of services.

3:Index

Register Access

The control chip registers are implemented as a set of phantom registers to the second bank of FM registers. Access to the the control chip is triggered by writing 0FFh to the address register of the second FM bank (38Ah). Thereafter, all reads/writes will access the control chip. Access to the second FM bank is returned by writing 0FEh to the same address register.

As with the FM and sampling chips, the control chip uses two port addresses. The first address, 38Ah, is the address register and writing a register number to this address selects a given data register. The second address, 38Bh, is the data address. Values written to this address are directed to the register number specified by the previous write to the address register. There are delays that must be respected when writing to certain registers. These delays are explained in detail in the *Status Register* section.

By default, the control chip is located at 38Ah and 38Bh. However, the chip may be relocated (as explained in the section *Audio Relocalization*). Regardless of where the chip is located, the data register port address is always one greater than the address register port address.

All data registers on the control chip are read/write. Reading a register will return its current value. The only execption to this are registers 0 and 1. All registers are explained below in detail.

The GSS level 2 cards contain permanent memory (EEPROM) in which the boot-up values for all registers are stored.

Disabling Interrupts when accessing the hardware

In order to avoid possible conflicts between applications that try to access the same hardware at the same time, it is recommended that interrupts be disabled when accessing the OPL3, the Control Chip or the MMA. This will avoid conflicts between applications, TSR programs and drivers that will be supplied with the GSS Cards card in the future.

This procedure should be strictly adhered to for all software developed for the GSS Cards card.

To insure that the interrupt flag status is not destroyed when re-enabling interrupts, the following procedure is recommended:

To disable interrupts:

```
pushf          ; push flags, include interrupt flags
cli            ; clear interrupts
```

To re-enable interrupts:

```
popf           ; pop flag, includes interrupt flags
```

Status Register

Reading the address port (38Ah by default) when the control chip access has been triggered returns the following information:

D7	D6	D5	D4	D3	D2	D1	D0
RB	SB	X	X	SCSI	TEL	SMP	FM

The 4 least significant bits indicate interrupt status. Reading this register does not reset the interrupt status. A zeroed bit indicates which section of the board has generated an interrupt. FM indicates the FM section has generated an interrupt; SMP, the sampling section; TEL, the telephone section; SCSI, the SCSI section. SB set indicates that the card is busy writing to a register. RB set indicates that the card is busy writing its registers to memory.

A delay of approximately 450 µsec is required after writing to any of registers 4 to 8. A delay of approximately 5 µsec is required after writing to any of registers 9 through 16. As well, the chip must not be accessed while the chip is saving its registers to memory. In order to respect these delays, the SB and RB bits should be polled until they become zero. As a general rule, always poll the SB and RB bits before writing anything to the chip.

As well, the chip must not be accessed while it is restoring its registers from memory. This process takes a bit less than 2.5 milliseconds. As there is no status bit for this action, the timing must be done in software.

IMPORTANT**: Before returning access to the FM chip (writing FEh to 38Ah), all delays must have expired. Results will be unpredictable otherwise.**

Register Map

The diagram on the following page is a summary of the control chip registers. When writing to registers which contain undesignated bits, these bits must be set to zero. Locations where certain bits must be set are indicated by a "1" in the register map.

Register Map, Control Chip

<table>
<tr><th>REG</th><th>D7</th><th>D6</th><th>D5</th><th>D4</th><th>D3</th><th>D2</th><th>D1</th><th>D0</th></tr>
<tr><td>00</td><td colspan="6"></td><td>ST</td><td>RT</td></tr>
<tr><td>01</td><td colspan="6"></td><td>RING</td><td>TC</td></tr>
<tr><td>02</td><td colspan="8">SAMPLING GAIN - LEFT</td></tr>
<tr><td>03</td><td colspan="8">SAMPLING GAIN - RIGHT</td></tr>
<tr><td>04</td><td>1</td><td>1</td><td colspan="6">FINAL OUTPUT VOLUME - LEFT</td></tr>
<tr><td>05</td><td>1</td><td>1</td><td colspan="6">FINAL OUTPUT VOLUME -RIGHT</td></tr>
<tr><td>06</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="4">BASS</td></tr>
<tr><td>07</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="4">TREBLE</td></tr>
<tr><td>08</td><td>1</td><td>1</td><td>MU</td><td colspan="2">ST-MONO</td><td colspan="3">SOURCE</td></tr>
<tr><td>09</td><td colspan="8">FM VOLUME - LEFT</td></tr>
<tr><td>0A</td><td colspan="8">FM VOLUME - RIGHT</td></tr>
<tr><td>0B</td><td colspan="8">SAMPLING VOLUME - LEFT</td></tr>
<tr><td>0C</td><td colspan="8">SAMPLING VOLUME - RIGHT</td></tr>
<tr><td>0D</td><td colspan="8">AUX VOLUME - LEFT</td></tr>
<tr><td>0E</td><td colspan="8">AUX VOLUME - RIGHT</td></tr>
<tr><td>0F</td><td colspan="8">MICROPHONE VOLUME</td></tr>
<tr><td>10</td><td colspan="8">TELEPHONE VOLUME</td></tr>
<tr><td>11</td><td colspan="2"></td><td>SPKR</td><td></td><td>MFB</td><td>XMO</td><td>FLT0</td><td>FLT1</td></tr>
<tr><td>12</td><td colspan="8"></td></tr>
<tr><td>13</td><td>DEN0</td><td colspan="3">DMA SEL 0</td><td>AEN</td><td colspan="3">INT SEL A</td></tr>
<tr><td>14</td><td>DEN1</td><td colspan="3">DMA SEL 1</td><td colspan="4"></td></tr>
<tr><td>15</td><td></td><td colspan="7">AUDIO RELOCATE</td></tr>
<tr><td>16</td><td>DENS</td><td colspan="3">DMA SEL S</td><td>SIEN</td><td colspan="3">INT SEL S</td></tr>
<tr><td>17</td><td></td><td colspan="7">SCSI RELOCATE</td></tr>
<tr><td>18</td><td colspan="8">SURROUND</td></tr>
</table>

Control/ID

D7	D6	D5	D4	D3	D2	D1	D0
X	X	X	X	X	X	ST	RT

Register #0: Write

Writing to the Control/ID byte with the ST bit set will cause all control chip registers, in their current state, to be written to memory. If RT is set, then all registers will be restored from memory. When the operation is finished, the control chip sets the appropriate bit back to zero. It is not necessary to manually clear the bit.

D7	D6	D5	D4	D3	D2	D1	D0
X	OP2	OP1	OP0	MODEL ID			

Register #0: Read

Reading this register gives information on the model of the card and which options are present. The currently defined MODEL ID's are:

ID	GSS Model
0	16 bit bus
1	8 Bit bus
2	MicroChannel

The OP0, OP1 and OP2 bits indicate which of the board options are present and are SET when the option is NOT present.

Bit	Option
OP0	Telephone
OP1	Surround
OP2	CD-ROM

Reg.1: Telephone Control

D7	D6	D5	D4	D3	D2	D1	D0
X	X	X	X	X	X	X	TC

Register #1: Write

D7	D6	D5	D4	D3	D2	D1	D0
X	X	X	X	X	X	RING	TC

Register #1: Read

Setting TC engages the telephone line; clearing the bit hangs up. Reading this register returns the state of the telephone ring signal: RING set indicates that the line is NOT ringing and TC returns the status of the telephone line (i.e. the previously written value of TC).

Reg. 2-3: Sampling Gain

Registers 2 and 3 control the gain on sampling channels 0 (left) and 1 (right). 256 different gain values are possible, giving a range from approximately 0.04 to 10 times the input value. The exact gain is given by the equation:

Gain = (RegisterValue * 10) / 256

Reg. 4-5: Final Output Volume

These registers control the overall output volume of the card. They replace the potentiometer found on the original Ad Lib card. Adjusting for left and right channels separately allows the balance to be varied.

The volume ranges from +6 dB to -64 dB in steps of 2 dB. An additional step gives -80 dB (off). IMPORTANT: Bits D6 and D7 must be set to 1.

dB	D5-D0
6	3F
4	3E
o o o	o o o
-62	1D
-64	1C
-80	1B
o o o	o o o
-80	0

Registers #4 and #5

Reg. 6: Bass

The bass control has a range of +15dB to -12 dB in 3 dB steps. The bass is set using bits D0-D3. IMPORTANT: Bits D4 - D7 must be set to 1.

dB	D3-D0
15	F
o o o	o o o
15	B
12	A
o o o	o o o
0	6
o o o	o o o

-12	2
o o o	o o o
-12	0

Register #6

Reg. 7: Treble

The treble control has a range of +12dB to -12 dB in 3 dB steps. The treble is set using bits D0-D3. IMPORTANT: Bits D4 - D7 must be set to 1.

dB	D3-D0
12	F
o o o	o o o
12	A
o o o	o o o
0	6
o o o	o o o
-12	2
o o o	o o o
-12	0

Register #7

Reg. 8: Output Mode

D7	D6	D5	D4	D3	D2	D1	D0
1	1	MU	ST-MONO		SOURCE		

Register #8

This register controls the final output. This final output section takes as its input the output from the mixing section. SOURCE indicates which channels from the mixer are selected for final output. If only one input channel is selected, it is directed to both output channels. Stereo input results in stereo output.

SOURCE	Channels
6	Left and right
4	Right only
2	Left only

ST-MONO selects the type of effect applied to the final ouput:

ST-MONO	Effect
3	Spatial stereo
2	Pseudo stereo
1	Linear stereo
0	Forced mono

Linear stereo is ordinary, stereo output with no effects added. The spatial and pseudo stereo effects will be useful primarily when the original sources are monophonic. If the surround option is present, the output signal is modified after mixing and the attributes of this register are then applied.

Setting MU enables muting; clearing it disables muting.

IMPORTANT: Bits D6 and D7 must be set to 1.

Reg. 9-10: Mixing Volumes

Registers 9 through 10h are individual volume control registers and constitute the mixing section of the card. 128 different linear volume levels are possible, ranging from 128 (silent) to 255 (maximum gain). Note that writing values less than 128 will result in a signal with negative polarity and should be avoided because the resulting signal may cancel out another signal of opposite polarity.

Reg. 11: Audio Selection

D7	D6	D5	D4	D3	D2	D1	D0
X	X	SPKR	X	MFB	XMO	FLT0	FLT1

Register #11h

The GSS card uses antialiasing filters during sampling and playback to ensure maximum audio quality. Because these operations are mutually exclusive on a given channel, the same antialiasing filter is used for sampling and playback. When FLT0 is set, the filter for Channel 0 (left) is set for input (recording); clearing the bit sets the filter for output (playback). FLT1 operates similarly, but is applied to Channel 1 (right).

Normally, the Aux input on the card is sampled in stereo on both channels at the same time. This stereo input can be turned monophonic and sampled on Channel 0 by setting XMO. Clearing XMO returns Aux input to its normal state.

When the telephone option of the GSS card is present, microphone input is directed to both the loudspeaker output as well as the telephone when MFB is cleared. However, this could cause feedback to occur. When MFB is set, the microphone signal is not directed to the loudspeaker output, thus eliminating possible causes of feedback. Although this feature is intended for use with the telephone option, it is operational at all times so that setting MFB always removes the microphone from the final output.

The internal audio speaker from the PC can be mixed directly with the final audio signal of the GSS Card. When SPKR is cleared, the signal is disconnected; when set it is connected.

Register 12h

Register 12h is unused and should be ignored or set to 0 otherwise.

Reg. 13: Audio IRQ/DMA Select - Channel 0

D7	D6	D5	D4	D3	D2	D1	D0
DEN0	DMA SEL 0			AEN	INT SEL A		

Register #13h

Audio interrupts (FM, sampling and telephone) are enabled when AEN is set. The following values for INT SEL A select the corresponding interrupt line:

INT SEL A	IRQ
0	3
1	4
2	5
3	7
4	10
5	11
6	12
7	15

Only IRQ 3, 4, 5, and 7 are available on 8-bit bus models. All listed interrupts are available on the 16-bit bus and MicroChannel Bus..

DMA for sampling channel 0 is enabled when DEN0 is set. The following values for DMA SEL 0 select the corresponding DMA line:

DMA SEL 0	DMA Line
0	0
1	1
2	2
3	3

Only DMA 1, 2 and 3 are available on 8-bit bus models. All listed DMA lines are available otherwise.

Reg. 14: DMA Select - Channel 1

D7	D6	D5	D4	D3	D2	D1	D0
DEN1	DMA SEL 1			X	X	X	X

Register #14

DMA for sampling channel 1 is enabled when DEN1 is set. The following values for DMA SEL 1 select the corresponding DMA line:

DMA SEL 1	DMA Line
0	0
1	1
2	2
3	3

Only DMA 1, 2 and 3 are available on the 8 bit-bus models. All listed DMA lines are available otherwise

Reg. 15: Audio Relocalisation

D7	D6	D5	D4	D3	D2	D1	D0
X	AUDIO RELOCATE						

Register #15h

This register indicates the port address for the audio section (FM, sampling, control chip). Writing here immediately relocates the audio section to the specified address. The AUDIO RELOCATE value is the port address divided by eight. This forces the address to be on an 8-byte boundary.

The audio section uses 8 port addresses. It is the first of these 8 addresses which is used in this register. Note that the control chip address is considered to be part of the audio section, so that the address of the control chip changes as soon as this register is modified.

The following is the default configuration for the audio section:

Address	Section
388h, 389h	FM Bank 0
38Ah, 38Bh	FM Bank 1, Control Chip
38Ch, 38Dh	Sampling Channel 0
38Eh, 38Fh	Sampling Channel 1

Reg. 16: SCSI IRQ/DMA Select

D7	D6	D5	D4	D3	D2	D1	D0
DENS	DMA SEL S			SIEN	INT SEL S		

Register #16h

SCSI interrupts are enabled when SIEN is set. The following values for INT SEL S select the corresponding interrupt line:

INT SEL S	IRQ
0	3
1	4
2	5
3	7
4	10
5	11
6	12
7	15

Only IRQ 3, 4, 5, and 7 are available on 8-bit bus models. All listed interrupts are available otherwise.

SCSI DMA is enabled when DENS is set. The following values for DMA SEL S select the corresponding DMA line:

DMA SEL S	DMA Line
0	0
1	1
2	2
3	3

Only DMA 1, 2 and 3 are available on 8-bit bus models. All listed DMA lines are available otherwise.

Reg. 17: SCSI Relocalization

D7	D6	D5	D4	D3	D2	D1	D0
X	SCSI RELOCATE						

Register #17h

This register indicates the port address for the SCSI section. Writing here immediately relocates the SCSI section to the specified address. The SCSI RELOCATE value is the port address divided by eight. This forces the address to be on an 8-byte boundary. The SCSI section uses 8 port addresses. It is the first of these 8 addresses which is used in this register. The default configuration has the SCSI section at addresses 340h.

Reg. 18: Surround

D7	D6	D5	D4	D3	D2	D1	D0
SURROUND							

Register #18h

The surround sound option of the card is accessed via this register. It will be documented at a later date.

This chapter explains the features of the new FM synthesis chip, the YMF262, on the Ad Lib GSS cards. This chip is similar to the YM3812, the chip on the original Ad Lib card, and contains a compatibility mode to emulate the YM3812. Because of this similarity, the first part of this section discusses the features of the YM3812. Those of you who are already familiar with this chip may wish to skip this section and proceed to *Programming the YMF262*, which discusses the differences between the two chips.

(NOTE: This section is reproduced from the original Ad Lib Synthesizer Card Programmer`s Manual. It is necessary for understanding the functioning of the new FM chip, the YMF262. If you are already familiar with this material, you may wish to proceed to the following section which discusses the YMF262.)

This section provides information about the Ad Lib Music Synthesizer Card for advanced programmers who wish to program it directly. There is information on the components of the card, a technical description of the operators, the input / output map and a register reference section.

The Ad Lib Music Synthesizer Card

The card is equipped with a vibrato oscillator, an amplitude oscillator (tremolo), a noise generator which allows for the combination of a number of frequencies, two programmable timers, composite sine wave synthesis and 18 operators.

A white noise generator is used to create rhythm sounds. This white noise generator uses voices 7 and 8 (melodic voices), frequency information (Block, F-Number, Multi), and the proper phase output. Various rhythm sounds are produced by combining this output signal with white noise. The resulting signal is then sent to the operators. Experience has shown that the best ratio for the two frequencies is 3:1 (melodic voice 7 frequency = 3 times melodic voice 8 frequency). Finally, envelope information is multiplied with the wave table output. As the envelope is set for one operator which corresponds to a single rhythm instrument, the values which express that instrument's characteristics are set in the parameter registers in the same manner as for melody instruments.

Operators

The ALMSC uses pure sine waves that interact together to produce the full harmonic spectrum for any voice. Each digital sine wave oscillator is combined with its own envelope generator to form an "operator".

An operator has 2 inputs and 1 output. One input is the pitch oscillator frequency and the other is for the modulation data. The frequency and modulation data (phases) are added together and converted to a sine wave signal. The phase generator (PG) converts the frequency (w) into a phase by multiplying it by time (t). An envelope generator (EG) produces a time variant amplitude signal (ADSR). The EG's output is then multiplied by the sine wave and output to the outside world.

The operator output can be expressed as a mathematical expression:

$$F(t) = E(t) \sin(wt + \_)$$

E(t) is the output from the EG, w is the frequency, t is time and _ is the phase modulation.

The operators can be connected in three different ways: additive, frequency modulation and composite sine wave.

- **FM synthesis**
  FM synthesis uses two operators in series. The first operator, the modulator, modulates the second operator via its modulation input. The name given to the second operator is the carrier. The modulator can feed back its output into its modulation data input;

  $F_m(t) = E_m(t) \sin(w_m t + ßF_m(t))$ Modulator and feedback
  $F_c(t) = E_c(t) \sin(w_c t + F_m(t))$ Carrier and Modulator

- **Additive synthesis**
  Additive synthesis connects two operators in parallel, adding both outputs together. This method of synthesis is not as interesting as FM synthesis, but it can generate good organ type sounds.

  The simplified formula for the additive synthesis is:

  $F(t) = E_1(t) \sin(wt + \_{1}) + E_2(t) \sin(wt + \_{2})$

- **Composite sine wave synthesis**
  Composite sine wave synthesis (CSW) may be used to generate speech or other related sounds by playing all voices simultaneously. When using this mode the card cannot generate any other sounds. This mode is not used because other methods have proved to provide better quality speech.

ALMSC Input / Output Map

The ALMSC is located at address 388H in the i/o space. The card decodes two addresses: 388H and 389H. The first address is used for selecting the register address and the second is used for writing data to the selected register. There also exists the possibility of using three other addresses: 218H, 288H and 318H. The port address is currently hard-wired, but address jumpers may be added in the future so you may want to take into account the possibility of using different addresses when programming. Here is a register map of the ALMSC:

Because of the nature of the card, you must wait 3.3 µsec after a register select write and 23 µsec for a data write. Only the status register located at address 388H can be read.

For many parameters, there is one register per operator. However, there are holes in the address map so that the operator number cannot be used as an offset into the map. The operator offsets are as follows:

For example, the KSL/TL registers are at 40H-55H. If we wish to access the register for operator 8, we must write to register 49H (NOT 48H).

Test Register/WSE

This register must be initialized to zero before taking any action. The wave select enable/disable bit (WSE) is D5. If set to 1, the value in the WS register will be used to select the wave form used to generate sound. If the WSE is set to 0, the value in the WS register will be ignored and the chip will use a sine wave. (The available waveforms are detailed later in this section).

Timers

The timers are not wired on the card. However, the following information is included since the timers can be used to detect the presence of our card in the computer.

Timer-1 is an upward 8 bit counter with a resolution of 80 µsec. If an overflow occurs, the status register flag FT1 is set, and the preset value (address = 02) is loaded into Timer-1. Timer-2 (address = 03) is an upward 8 bit counter just like Timer-1 except that the resolution is 320 µsec.

$$T_{overflow}(ms) = (256-N) * K$$

N is the preset value and K is the timer constant equal to 0.08 for Timer-1 and 0.32 for Timer-2. Register address 04 controls the operation of both timers. ST1 and ST2 (start/stop T1 or T2) bits start or stop the timers. When the corresponding bit is 1 the counter is loaded and counting starts, but when 0 the counter is held.

The Mask bits are used to gate the status register timer flags. If a mask bit is 1 then the corresponding timer flag bit is kept low (0) and is active when the mask bit is cleared (0). The most significant bit (MSb) is called IRQ-RESET. It resets timer flags and IRQ flag in the status register to zero. All other bits in the control register are ignored when the IRQ-RESET bit is 1.

Status Register

Reading at address 388H yields the following byte of information:

- D0 - D4 are unused.
- D5 Timer 2 flag: Set to 1 when the preset time in Timer 2 has elapsed. The flag remains until reset.
- D6 Same as D5, except for Timer 1.
- D7 IRQ flag: set if D5 or D6 are 1.

As mentioned earlier, the timer interrupts are not connected, but the timers can be used to detect the presence of the board as follows:

1. Reset T1 and T2: write 60H to register 4.
2. Reset the IRQ: write 80H to register 4 (this step must NOT be combined with Step #1).
3. Read status register: read at 388H. Save the result.
4. Set timer-1 to FFH: write FFH to register 2.
5. Unmask and start timer-1: write 21H to register 4.
6. Wait (in a delay loop) for at least 80 µsec.
7. Read the status register and save the result.
8. Reset T1, T2 and IRQ as in steps #1 and #2.
9. Test the results of the two reads: the first should be 0, the second should be C0H. If either is incorrect, then an ALMSC board is not present. (NOTE: You should AND the result bytes with E0H as the unused bits are undefined.)

CSM/Keyboard Split

This register (address = 08) will determine if the card is to function in music mode (CSM = 0) or speech synthesis mode (CSM = 1) as well as the keyboard split point.

When using composite sine wave speech synthesis mode all voices should be in the KEY-OFF state. The bit NOTE-SEL (D6) is used to control the split point of the keyboard. When 0, the keyboard split is the second bit from the MSb (bit 8) of the F-Number. The MSb of the F-number is used when NOTE-SEL = 1. This is illustrated in the following table:

AM/VIB/EG-TYP/KSR/Multiple

This group of registers (addresses 20H to 35H), one per operator, controls the frequency conversion factor and modulating wave frequencies corresponding to the frequency components of music.

The MULTI 4-bit field determines the multiplication factor applied to the input pitch frequency in the PG section. That is, an operator's frequency will automatically be multiplied according to the value in this field. The multiplication factors are given in the following table:

The operator output can then be expressed, with "_" as the multiplication factor, as follows:

$$F(t) = E_C(t)\ \sin(\_{}_C w_C t + E_m \sin(\_{}_m w_m t))$$

The KSR bit (position = D4) changes the rates for the envelope generator (EG). This parameter makes it possible to gradually shorten envelope length (increase EG rates) as higher notes on the keyboard are played. This is particularly useful for simulating the sound of stringed instruments such as piano and guitar, in which the envelope of the higher notes is noticeably shorter than the lower notes. The actual rate is then equal to the ADSR value plus an offset:

Actual rate = 4*Rate + KSR offset

The KSR offset is specified in the following table:

The EG-Type activates the sustaining part of the envelope when the EG-Type is set (1). Once set, an operator's frequency will be held at its sustain level until a KEY-OFF is done.

The VIB parameter toggles the frequency vibrato (1 = on, 0 = off). The frequency of the vibrato is 6.4 Hz and the depth is determined by the DEP VIB bit in register 0BDH.

The AM parameter is similar to the VIB parameter except that it is an amplitude vibrato (tremolo) of frequency 3.7Hz. The amplitude vibrato depth is determined by the DEP AM bit in register 0BDH.

KSL/Total Level

These registers (addresses 40H to 55H, 1 per operator) control the attenuation of the operator's output signal. The KSL parameter produces a gradual decrease in note output level towards higher pitch notes. Many acoustic instruments exhibit this gradual decrease in output level. The KSL is expressed on 2 bits (value 0 through 3). The corresponding attenuation is given below:

D7	D6	Attenuation
0	0	0
1	0	1.5dB/oct
0	1	3.0dB/oct
1	1	6.0dB/oct

The Total Level (TL) attenuates the operator's output. In FM synthesis mode, varying the output level of an operator functioning as a carrier results in a change in the volume of that operator's voice. Attenuating the output from a modulator will change the frequency spectrum produced by the carrier. In additive synthesis, varying the output level of any operator varies the volume of its corresponding voice. The TL value has a range of 0 through 63 (6 bits). To convert this value into an output level, apply the following formula:

Output level = (63 - TL) * 0.75dB

ADSR

These values change the shape of the envelope for the specified operator by changing the rates or the levels. The attack (AR) and the decay (DR) rates are at addresses 60H to 75H (1 per operator). The Sustain Level (SL) and Release Rate (RR) are located at addresses 80H to 95H. All of these values are 4 bits in length (range 0 to 15). Refer to the diagram on page 11 for more information.

The attack rate (AR) determines the rising time for the sound. The higher the value in this register, the faster the attack.

The decay rate (DR) determines the diminishing time for the sound. The higher the value in the DR register, the shorter the decay.

The sustain level (SL) is the point at which the sound ceases to decay and changes to a sound having a constant level. The sustain level is expressed as a fraction of the maximum level. When all bits are set, the maximum level is reached. Note that the EG-Type bit must be set for this to have an effect.

The release rate (RR) determines the rate at which the sound disappears after a Key-Off. The higher the value in the RR register, the shorter the release time.

BLOCK/F-Number

These parameters determine the pitch of the note played. The Block parameter determines the octave while the F-Number (10 bits) further specifies the frequency. The following formula is used to determine the value of F-Number and Block:

$$\text{F-Num} = F_{mus} * 2^{(20-b)} / 49.716\text{ kHz}$$

In this formula, F_{mus} is the desired frequency (Hz) and "b" is the block value (0 to 7). Refer to Appendix C for a table of note frequencies.

The D5 bit in the register that contains the BLOCK information is called KEY-ON (KON) and determines if the specified voice (0 to 8) is enable (1) or disable (0). The lower bits of F-Number are at location A0H through A8H (1 per voice) and the 2 MSb are at positions D0 and D1 of addresses B0H to B8H.

Rhythm/AM Dep/VIB Dep

This register allows for control over AM and VIB depth, selection of rhythm mode and ON/OFF control for various rhythm instruments. Bit D5 (R) is used to change the mode from melodic (0) to percussive (1). When in percussive mode, bits D0 through D4 are the KEY-ON/KEY-OFF controls for the rhythm instruments listed below. The KEY-ON bit in registers B6H, B7H and B8H must always be 0 when in percussive mode.

D0	Hi-Hat
D1	Cymbal
D2	Tom-Tom
D3	Snare Drum
D4	Bass Drum

The AM Depth is 4.8dB when D7 is 1 and 1dB when 0. The VIB Depth is 14 cents when D6 is 1, and 7 cents when zero. (A "cent" is 1/100th of a semi-tone.)

FeedBack/Connection

These two parameters influence the way the operators are connected together and the ß factor in the feedback loop of the modulator. These parameters are assigned 1 per voice at locations C0H through C8H. The Connection bit (C) determines if the voice will be functioning in Additive synthesis mode (C = 1) of in Frequency modulation mode (C = 0). The other parameter, Feedback (FB), gives the modulation factor, ß, for the feedback loop:

Wave Select

The WS parameter enables the card to generate other kinds of wave shapes. This is done by changing the sine function of the specified operator. (Note that the WSE bit must be set in order to use this feature.) The addresses of this feature are E0H to F5H. The following figure gives the corresponding wave forms:

This section explains the differences between the Ad Lib GSS Sound Adapter and the original Ad Lib Music Synthesizer Card as regards FM synthesis. A previous knowledge of the original Ad Lib card is assumed. If you are unfamiliar with the original card, you should first read the following section: "Programming the Synthesizer", which is reproduced from the original Programmer's Manual .

You can see from the register map on the following page that the new FM section is quite similar to the original FM chip but with extra features added. Register Array 0 is accessed by writing to addresses x and x+1 (388H and 389H by default). Register Array 1 is accessed by writing to addresses x+2 and x+3 (38AH and 38BH by default). This scheme allows for complete compatibility with older software which recognizes only the original Ad Lib card.

All registers are cleared at reset. The TEST registers at 01 should be cleared or not accessed at all. Bits in the register map which are not designated should be left in their cleared state.

Register Array 0

Register Array 0 emulates the original chip and will be used as such by software written for the original card. However, there are several changes to be noted.

The Wave Select Enable bit (WSE, D5 at 01) no longer exists. Wave Select is now "on" permanently. Writing 1 to D5 at 01 has no effect so that compatiblity is thereby maintained.

The CSM bit (D7 at 08) found on the original chip is no longer present. Although this bit was documented on the original chip, it was non-functional. Compatibility is, therefore, not an issue.

The timers are now functional. How to program them is explained in the *Timers* section of *Programming the Synthesizer*.

Register Map, FM Array 0

<table>
<tr><th>REG</th><th>D7</th><th>D6</th><th>D5</th><th>D4</th><th>D3</th><th>D2</th><th>D1</th><th>D0</th></tr>
<tr><td>01</td><td colspan="8">TEST</td></tr>
<tr><td>02</td><td colspan="8">TIMER-1</td></tr>
<tr><td>03</td><td colspan="8">TIMER-2</td></tr>
<tr><td>04</td><td>RST</td><td colspan="2">mask
T1 | T2</td><td colspan="3"></td><td colspan="2">start/stop
T2 | T1</td></tr>
<tr><td>05</td><td colspan="8"></td></tr>
<tr><td>08</td><td></td><td>SEL</td><td colspan="6"></td></tr>
<tr><td>20-35</td><td>AM</td><td>VIB</td><td>EG</td><td>KSR</td><td colspan="4">MULTI</td></tr>
<tr><td>40-55</td><td colspan="2">KSL</td><td colspan="6">TL</td></tr>
<tr><td>60-75</td><td colspan="4">AR</td><td colspan="4">DR</td></tr>
<tr><td>80-95</td><td colspan="4">SL</td><td colspan="4">RR</td></tr>
<tr><td>A0-A8</td><td colspan="8">F-NUMBER (L)</td></tr>
<tr><td>B0-B8</td><td colspan="2"></td><td>KON</td><td colspan="3">BLOCK</td><td colspan="2">F-NUM (H)</td></tr>
<tr><td>BD</td><td>DEP
AM</td><td>DEP
VIB</td><td>R</td><td>BD</td><td>SD</td><td>TOM</td><td>TC</td><td>HH</td></tr>
<tr><td>C0-C8</td><td colspan="2"></td><td>SRL</td><td>STR</td><td colspan="3">FB</td><td>C</td></tr>
<tr><td>E0-F5</td><td colspan="5"></td><td colspan="3">WS</td></tr>
</table>

Register Map, FM Array 1

<table>
<tr><th>REG</th><th>D7</th><th>D6</th><th>D5</th><th>D4</th><th>D3</th><th>D2</th><th>D1</th><th>D0</th></tr>
<tr><td>01</td><td colspan="8">TEST</td></tr>
<tr><td>02</td><td colspan="8"></td></tr>
<tr><td>03</td><td colspan="8"></td></tr>
</table>

04			CONNECTION SELECT					
05								NEW
08								
20-35	AM	VIB	EG	KSR	MULTI			
40-55	KSL		TL					
60-75	AR				DR			
80-95	SL				RR			
A0-A8	F-NUMBER (L)							
B0-B8			KON	BLOCK			F-NUM (H)	
BD								
C0-C8			SRL	STR	FB			C
E0-F5						WS		

Each voice now has two bits which control stereo output: STL and STR (D5/D4 at C0-C8). Setting STL enables output to the left channel. Setting STR enables output to the right channel. Clearing both bits will result in no output for a given voice. However, for these bits to have effect, the NEW bit (explained in the next section) must be set. If NEW is not set (its default state), then the STL and STR bits are ignored and sound is output to both channels. This maintains compatibility with older software which ignores the existence of the stereo bits.

The stereo bits affect pairs of operators, which creates a particularity in percussive mode. The stereo bits in C7 simultaneously affect the Hi-Hat and Snare Drum; C8 affects the Tom-Tom and Cymbal similarly. The Bass Drum (C6) uses two operators and functions the same as a melodic voice.

The Wave Select has been expanded to 3 bits, thus allowing for a total of 8 different waveforms. The waveforms are shown below.

Register Array 1

Register Array 1 is similar to Register Array 0 with some omissions and additions. The timer registers are unused or are used for other purposes. Register Array 1 does not offer percussive voices, so the bits relating to percussive mode are not present.

The SEL, DEP AM and DEP VIB bits are globally affective and so are found only in the first register array. Setting any one of these three bits will affect both register arrays.

The NEW bit (D0 at 05) enables the new features of the new chip. If this bit is zero, then writes to any other register in Register Array 1 will be blocked. When NEW is zero, Register Array 0 functions as if it were the original chip: the stereo bits will be ignored and the high bit of the wave select will be ignored.

IMPORTANT: All software should enable the NEW bit during its initialization sequence. However, it should clear the NEW bit when exiting. This is so that if an older piece of software is subsequently run, the card will be in the mode which emulates the original card.

The CONNECTION SELECT bits control the 4-operator voice, as explained in detail in the next section.

4-Operator Voices

A significant new feature of the FM section of the Ad Lib GSS card is the presence of 4-operator voices, which are capable of creating a large variety of rich timbres. To enable a 4-operator voice, you must set the appropriate bit in the CONNECTION SELECT register. The following table shows which bit corresponds to which 4-operator voice and the pair of 2-operator voices which correspond to the 4-operator voice.

Connection Select (05H, Register Array 1):

	D5	D4	D3	D2	D1	D0
4-op voice	6	5	4	3	2	1
2-op voices	3,6	2,5	1,4	3,6	2,5	1,4
	Array 1		Array 0			

With 2-operator voices, the connection bit at C0-C8 specifies one of two possible methods for connecting the operators. With 4-operator voices, there are 4 methods of connecting the operators. This is done by using both connection bits of the pair of 2-operator voices involved. The following table shows the relationship between the 4-operator voice and its connection bits. The diagram on the next page illustrates the connection methods.

Connection bit (**C**) addresses for 4-operator voices:

4-op voice	1 2	3 4	5 6			
C addresses	C0,C3	C1,C4	C2,C5	C0,C3	C1,C4	C2,C5
	Array 0	Array 1				

Note that even if all six 4-operator voices are used, there are still three 2-operator voices available on Register Array 1 and three 2-operator or five percussive voices available on Register Array 0. The CONNECTION SELECT register allows you to selectively use 4-operator voices so that you can mix 2 and 4-operator voices as you wish.

The following table is a combination of the preceding two tables. You may find it useful for reference purposes.

Connect Sel	D5 D4	D3 D2	D1 D0			
4-op voice	6 5	4 3	2 1			
2-op voices	3,62,5	1,43,6	2,51,4			
C addresses	C2,C5	C1,C4	C0,C3	C2,C5	C1,C4	C0,C3
	Array 1	Array 0				

Feedback in a 4-operator voice is applied to the first operator only, as indicated by the loop around Operator 1 in the diagram on the following page. The feedback value is determined by the value written in the register for the first register pair (**C**x). The value in the second register pair (**C**x+3) is ignored.

Similarly, the F-NUMBER, KON, and BLOCK parameters for a 4-operator voice are determined by the values written in the registers for the first register pairs (Ax and Bx). The values in the second register pairs (Ax+3 and Bx+3) are ignored.

Note that the state of the STL and STR bits for a 4-operator voice must be the same for both register pairs (Cx and Cx+3) or else the output of all four operators will be disabled. For example, if STL at C0 is 1 and STL at C3 is 0, then this 4-operator voice will not be output to the left channel.

3.3

The digital I/O functions are handled by the YMZ263 chip, also known as the MMA. The MMA handles the following functions:

- 2 channels of digital audio input and ouput
- MIDI input and output
- Three high-speed timers

The digital I/O functions are accessed via three addresses. The first address is located four bytes past the address of FM Array 0 (38CH by default).

Accessing a MMA register is done in two steps:

1) write the index of the register to be accesed to the “register select” port, located at 38CH

2) write or read the desired value for the selected register, either in the channel 0 port, located at 38DH or in the Channel 1 port located at 38FH

A 470 nanosecond delay is necessary betwen read/write at any address of the MMA

REG		D7	D6	D5	D4	D3	D2	D1	D0
01	-	TEST							
02	W	TIMER-0 (L)							
03	W	TIMER-0 (H)							
04	W	BASE COUNTER (L)							
05	W	TIMER 1				BASE COUNTER (H)			
06	R/W	TIMER 2 (L)							
07	R/W	TIMER 2 (H)							
08	W	SBY	T2M	T1M	T0M	STB	ST2	ST1	ST0
09	W	RST	R	L	FREQ		PCM	P/R	GO
0A	W	VOLUME CONTROL							
0B	R/W	PCM DATA							
0C	W	ILV	DATA FMT		FIFO INT			MSK	ENB

0D	W			MSK POV	MSK MOV	MDI TRS RST	MSK TRQ	MDI RCV RST	MSK RRQ
0E	R/W	MIDI DATA							

Register Map, Channel 0

REG		D7	D6	D5	D4	D3	D2	D1	D0
01	-								
02	W								
03	W								
04	W								
05	W								
06	R/W								
07	R/W								
08	W								
09	W	RST	R	L	FREQ		PCM	P/R	GO
0A	W	VOLUME CONTROL							
0B	R/W	PCM DATA							
0C	W		DATA FMT		FIFO INT			MSK	ENB
0D	W								
0E	R/W								

Register Map, Channel 1

Register Reference

Status Register

Reading the port at address 38CH returns the following information:

D7	D6	D5	D4	D3	D2	D1	D0
OV	T2	T1	T0	TRQ	RRQ	FIF1	FIF0

Status Byte

OV becomes 1 when a MIDI receive overrun error or a PCM/ADPCM record or playback overrun error occurs.

TO, T1 and T2 become 1 when the specified time elapses in the corresponding timer.

TRQ becomes 1 when the MIDI transmit FIFO buffer is empty.

RRQ becomes 1 when the MIDI receive FIFO buffer has data in it.

FIF0 and FIF1 become 1 when the PCM/ADPCM FIFO reaches the status that was specified in FIFO INT. FIF0 corresponds to channel 0; FIF1 to channel 1.

Register 00H: Test Register

Register #1, Channel 0 is used for testing the LSI. It should not be accessed.

Registers 02H - 07H: Timer Counters

Timer 0 (Registers #1 and 2, Channel 0) is a 16-bit programmable down counter with 1.88964 usec resolution. This constant will be referred to as clockFreq. the the following examples. The interrupt is triggerred when the counter value reaches 0. The time **t0**, in usec, until IRQ is generated may be calculated as follows:

$$\mathbf{t0} = \mathrm{TIMER0(H)} * (256*\mathrm{baseFreq}) + \mathrm{TIMER0(L)} * \mathrm{baseFreq}$$

The BASE COUNTER (Register #4 and 5, Channel 0) is a 12-bit counter that supplies the period for each tick of TIMER1 and TIMER2. The base counter has a resolution of 1.89 usec. The period **bc**, in usec, may be calculated as follows:

$$\mathbf{bc} = \mathrm{BASE\ COUNTER(H)} * (256*\mathrm{baseFreq}) + \mathrm{BASE\ COUNTER(L)} * \mathrm{baseFreq}$$

Timer 1 (Register #5, Channel 0) is a 4-bit programmable down counter that is controlled by the base counter clock. . The 4-bit value is placed in the high nibble of the register. The interrupt is triggerred when the counter value reaches 0. The time **t1**, in usec, until IRQ is generated may be calculated as follows:

$$\mathbf{t1} = \text{TIMER1} * \mathbf{bc}$$

Timer 2 (Register #6 and 7, Channel 0) is a 16-bit programmable down counter that is controlled by the base counter clock. The interrupt is triggerred when the counter value reaches 0. The time **t2**, in usec, until IRQ is generated may be calculated as follows:

$$\mathbf{t2} = (\text{TIMER2(H)} * 256 + \text{TIMER0(L)} * \mathbf{bc}$$

TIMER2 may be read to determine the count value. When TIMER2(L) is read the 16-bit count value is latched and the latched value of TIMER2(L) is output. Subsequently, when TIMER2(H) is read, the latched value of TIMER2(H) is output. (Latching a value means taking a "snapshot" of that value at a given moment.) TIMER2(L) must be read first as it is this read which triggers the latching mechanism.

Register 08H: Timer Control

D7	D6	D5	D4	D3	D2	D1	D0
SBY	T2M	T1M	T0M	STB	ST2	ST1	ST0

Register #8: Channel 0

Stand-by Mode

Setting SBY to 1 reduces the internal clock frequency in order to minimize power consumption. This must be set to 0 when doing any I/O operations.

Timer Interrupt Masks

Setting T0M, T1M or T2M disables the interrupt generated by the corresponding timer. Hence, the bit must be cleared if you wish to use the interrupt timer.

Timer Controls

ST0, ST1, ST2 and STB (base counter) contol the start and stop of each timer. Setting a bit loads the reload value and starts counting down. Clearing the bit stops the timer.

Register 09H: Playback and Recording Control

D7	D6	D5	D4	D3	D2	D1	D0
RST	R	L	FREQ		PCM	P/R	GO

Register #9: Channels 0 & 1

Reset PCM/ADPCM

RST bit is used to reset PCM and ADPCM playback for the channel. Resetting a channel clears the FIFO buffers and resets the FIFO flags. In order for reset to operate properly, all other bits should be 0. The sequence for a channel reset should then be: 1) write 80H to register 9 2) write the desired values to register 9.

Select Output Channel

Setting L or R enables output from the left or right channel respectively. Clearing the bit disables output.

Select Frequency

FREQ selects the PCM/ADPCM frequency as indicated below:

FREQ	Sampling Frequency (KHz.)	
	PCM Mode	ADPCM Mode
0	44.1	22.05
1	22.05	11.025
2	11.025	7.35
3	7.35	5.5125

PCM/ADPCM Selection

Setting PCM selects PCM mode (data is not compressed). Clearing PCM selects ADPCM mode (data is compressed to 4-bits).

Select Record/Playback

Clear P/R to record; set it to playback.

Start/Stop Record/Playback

In playback, the FIFO buffers should never be empty when the GO bit is set. To start playback, the proper procedure is: 1) write data into the FIFO buffer for the channel. The FIFO should be filled to a level exceeding the FIFO interrupt level (see register 0CH description) 2) Set the GO bit to start playback.

Register 0AH: Output Volume Control

VOLUME CONTROL (Register #0Ah, both channels) sets the output attenuation value. A value of 0 is the minimum output volume, a value of FF is the maximum ouput volume.

Register 0BH: PCM/ADPCM Data

Register #0Bh (both channels) is used for writing data into the FIFO buffer and reading data from the FIFO buffer. . Each channel has its own buffer. Data written into this register is transferred into the FIFO buffer, and data transferred from the FIFO buffer is written into this register. In PCM mode, 12-bit data is accessed in one or two passes. The data format for this access follows the specification of the FORMAT register. In ADPCM mode, each access inputs or outputs two 4-bit data. The high 4 bits and the low 4 bits are each ADPCM data. The high data is followed immediately by the low data.

Register 0CH: Sampling Format and Control

D7	D6	D5	D4	D3	D2	D1	D0
ILV	DATA FORMAT		FIFO INT			MSK	ENB

Register #0Ch: Channels 0 & 1

Interleaving

Setting ILV (Channel 0 *only*) to 1 will cause the chip to do interleaving. Data will be alternately input/output from each channel. Channel 0 initiates the transfer. ENB must be 1 for both channels, otherwise the data transfer is not performed. Both channels operate in the same mode so that the P/R,FREQ and GO bits will be controlled by the values set for channel 0.

Set Data Format

There are 3 possible data formats for sampling input and output. The format is selected by writing 0, 1 or 2 to the DATA FORMAT register. "3" is an invalid format... This is ignored in ADPCM mode.

Format 0 is an 1-byte format which contains the 8 most significant bits of the sample.

Format 1 is a 2-byte format. The first byte contains the 8 least significant bits. The lower nibble of the second byte contains the 4 most significant bits of the sample. The MSB of the sample is repeated in all bits of the upper nibble.

Format 2 is a 2-byte format as well. The upper nibble of the first byte contains the 4 LSBs of the sample. The lower nibble is zero. The second byte contains the 8 MSB's.

FORMAT	PCM Data Byte 1	PCM Data Byte 2
0	MSB b10 b9 b8 b7 b6 b5 b4	There is no 2nd byte
1	b7 b6 b5 b4 b3 b2 b1 b0	MSB MSB MSB MSB MSB b10 b9 b8
2	b3 b2 b1 b0 0 0 0 0	MSB b10 b9 b8 b7 b6 b5 b4

PCM Data Formats

Set FIFO Interrupt

The FIFO INT register is used to specify when an interrupt will be generated while the 128-byte FIFO buffer is being filled or emptied. The following table documents the possible interrupt points.

FIFO INT	Interrupt Generation Point (bytes)
0	112
1	96
2	80
3	64
4	48
5	32
6	16
7	Prohibited

FIFO Interrupt Mask

Setting MSK disables the FIFO interrupt.

DMA Mode Specification

Set ENB to enable the DMA mode. Clear ENB when not using DMA to transfer data.

Register 0DH: MIDI and Interrupt Control

D7	D6	D5	D4	D3	D2	D1	D0
		MSK POV	MSK MOV	MDI TRS RST	MSK TRQ	MDI RCV RST	MSK RRQ

Register #0Dh: Channel 0

Mask Digital Overrun Error

Set POV to disable interrupt signals generated by overrun errors during PCM/ADPCM recording and playback.

Mask MIDI Overrun Error

Set MOV to disable interrupt signals generated by overrun errors during MIDI reception or transmission.

Reset MIDI transmit circuit

Set MDI TRS RST to 1 to reset the MIDI transmit circuit and clear the MIDI transmit FIFO buffer. Zero MDI TRS RST to terminate the reset status.

Mask MIDI transmit FIFO interrupts

Set MSK TRQ to disable interrupt signales generated by the MIDI transmit FIFO. When interrupts are enabled, an interrupt is generated when the MIDI transmit FIFO buffer is emptied.

Reset MIDI Receive Circuit

Set MDI RCV RST to 1 to reset the MIDI receive circuit and clear the MIDI receive FIFO buffer. Zero MDI RCV RST to terminate the reset status.

Mask MIDI Receive FIFO Interrupts

Set MSK RRQ to disable interrupt signals generated by the MIDI receive FIFO buffer. When interrupts are enabled, an interrupt is generated on reception of a MIDI byte.

Register 0EH: MIDI Data

This register is used for writing data into the MIDI FIFO buffer an reaing data from the MIDI FIFO bufer. Data written in this register is ransferred to the transmit FIFO buffer and data transferred from the receive FIFO buffer can be read from this register.

MMA Programming Tips

- o Reset a MMA channel after each sample (using the RST bit in register 9), after stopping the sample playback. This makes sure that the FIFO buffer for the channel is emptied.

- o In playback mode, when processing a FIFO interrupt, a situation occurs where your application is filling in the FIFO while the playback mechanism is emptying the FIFO at the same time. In some cases this can cause "false triggers" of the FIFO interrupt. In order to avoid this, a simple trick is to temporarily lower the FIFO level, while your application fills in the FIFO, and restore the original level before leaving the interrupt procedure.

- o A similar situation can occur in recording mode.

- o To avoid the same situation during playback and recording using DMA transfers, you can double-check if the interrupt is valid by reading the DMA controller's counters or status register. they should indicate that data transfer is over.

- o The MMA FIFO buffers should never be left to empty themselves during playback (tht is wen GO bit is set) This implies that the FIFO buffers should be filled to a level exceeding the FIFO interrupt level before the GO bit is set.

  Special care should be taken during high-speed transfers (44.1K, 12 bit stereo samples, for example) on slower computers.

- o All masks (mask T2, T1, T0, FIFO, POV, MOV, TRQ and RRQ) have no effect whatsoever on the status register. They are only used to disable the hardware interrupt.

- o Respect the 470ns delay between writes to the MMA registers.

:Index

WaveOutWrite, 2.93