InterWave™ IC
Am78C201/202

Programmer's Guide

Rev. 2, 1996

A D V A N C E D M I C R O D E V I C E S

Trademarks

TABLE OF CONTENTS

Chapter 8. Local Memory Control

Chapter 20. System Control DDK Functions

LIST OF TABLES

LIST OF FIGURES

The InterWave™ Audio Integrated Circuit (IC) features a stereo synthesizer, a stereo coder/decoder (codec) and mixer, a processor-controlled interface, and standard game and Musical Instrument Digital Interface (MIDI) ports. It is designed for the personal computer (PC) market, and is specifically aimed at systems that are built to run the MS-DOS® and Microsoft® Windows® operating systems. It is intended to be used on system boards and add-in sound cards for desktop and portable computers.

This chapter gives an overview of this book and lists the typographical conventions used in it.

How To Use This Book

The purpose of this book is twofold. First, it is a teaching tool—it explains the functions of the InterWave audio IC and how to program it. Second, this book is a reference tool—it provides quick access to the facts required for programming the IC.

The best tool for programming applications to use the InterWave IC is the InterWave Game API, discussed in ###. You can also use custom APIs such as the AIL driver from John Miles. For educational purposes and for writing simple applications, you can use the InterWave Driver Developer's Kit (DDK).

If you are programming in Windows, you do not need this book. However, the information may still be useful to help you understand how the InterWave IC operates.

To assist the programmer in understanding the InterWave IC, this book includes an ample supply of code examples taken from the DDK function library. The code examples are incorporated into the chapters on programming the InterWave found in Part 2. The descriptions of the functions in the DDK API are in Part , "InterWave Game API and Driver Developer's Kit Reference." The DDK source code is available from AMD.

The book is organized as follows:

Part 1. Introducing the InterWave IC

Chapter 1, "Introduction," provides an brief functional description of the InterWave IC.

Chapter 2, "Software Environment," discusses the software tools available with the InterWave IC and the programming environment in which the InterWave IC is likely to be used.

Chapter 3, "Compatibility With Advanced Gravis UltraSound and Enhancements," discusses the compatibility of the InterWave IC with respect to the UltraSound products marketed by Advanced Gravis. InterWave technology is a backward-compatible enhancement of the technology used in the Gravis UltraSound (GUS) products.

Part 2. Programming the InterWave IC

Chapter 4, "Global Programming Topics," discusses general programming issues that relate to all functional areas of the IC.

Chapter , "Typographical Conventions," describes the System Bus Interface (SBI) and other system control functions, including the Plug and Play ISA capabilities of the InterWave IC.

Chapter 6, "Codec/Mixer," describes the codec and mixer module and how to program it.

Chapter 7, "Synthesizer," describes the synthesizer and how to program it.

Chapter 8, "Local Memory Control," describes the local memory control capabilities and how to program them.

Chapter 9, "Game and MIDI Ports," describes the game and MIDI ports and how to program them.

Chapter 10, "Legacy Sound Card Compatibility and Emulation," describes how to program the InterWave to support software written for other sound cards.

Part 3. InterWave Registers Reference

These chapters provide summary and detailed information about the programmable registers in the InterWave IC.

Chapter 11, "Register Summary," describes the register naming conventions and contains two tables listing all of the user-accessible registers in the InterWave IC, one ordered sequentially by I/O address and one ordered alphabetically by register mnemonic.

Chapter 12, "System Control Registers," describes the system control registers, which includes all registers that deal with the System Bus Interface (SBI), PNP functions, and compatibility with or emulation of other sound board products.

Chapter 13, "Codec/Mixer Registers," describes the codec module registers, which includes all registers that control inputs to and outputs from the codec and the mixer.

Chapter 14, "Synthesizer Registers," describes the synthesizer registers, which includes all registers for voice selection and control.

Chapter 15, "Local Memory Control Registers," describes the local memory control registers, which includes all registers used to write data to or read data from the InterWave IC's local memory.

Chapter 16, "Game Port and MIDI Port Registers," describes the game and MIDI port registers.

Part 4. InterWave Game API and Driver Developer's Kit Reference

These chapters contain information about the InterWave Game API and the InterWave DDK.

Chapter 17, "InterWave Game API Reference," discusses the InterWave Game API.

Chapter 18, "Programming With The Driver Developer's Kit," discusses general topics the programmer needs to know when using the InterWave DDK and lists the DDK functions.

Chapter 19, "DDK Quick Reference," briefly describes and lists all of the DDK functions.

Chapter 20, "System Control DDK Functions," contains detailed reference information about the DDK system control functions.

Chapter 21, "Codec/Mixer DDK Functions," contains detailed reference information about the DDK codec/mixer functions.

Chapter 22, "Synthesizer DDK Functions," contains detailed reference information about the DDK synthesizer functions.

Chapter 23, "Local Memory Control DDK Functions," contains detailed reference information about the DDK local memory control functions.

Appendixes

Appendix A, "Packaging and Pin Designations" lists and describes the pins in the InterWave IC.

Appendix B, "Sample Plug and Play Resource Map" provides a sample resource map for programming a serial EEPROM for use as a PNP ROM.

Typographical Conventions

To help you locate and interpret information easily, this book uses consistent visual cues and document conventions.

- When referring to registers and addresses, this book uses certain conventions when describing bit values and signals. These conventions are as follows:
  - — "High" refers to the logical value 1; "Low" refers to the logical value 0.
  - — When a bit should be a logical 1, the text says "set the bit High" or "if the bit is High."
  - — When a bit should be a logical 0, the text says "set the bit Low" or "if the bit is Low."
  - — When the text says a bit is set without specifying High or Low, it means the bit is set to a logical 1.
  - — When the text says a bit is cleared, it means the bit is set to a logical 0.
- When a register name appears in text, the name is spelled out with initial caps in the standard text font and is followed by the register mnemonic in parentheses. Subsequent references to the register in the same or immediately following paragraphs may use the register mnemonic only:

  Three bits from the PNP Power Mode register (PPWRI) must be set High before certain paths in the codec module can be used.

- Individual bits and bit ranges within registers are referred to by bit field name, spelled out with initial caps and printed in italics. Where necessary to prevent any ambiguity, the name of the register containing the bit field follows the bit field name. The register mnemonic, including the bit position for the bit field, follows the bit field name or the register name:

  If the *Auto Increment* bit of the LMC Control register (LMCI[0]) is set High, then the I/O address counter value automatically increments by one with each access through this port.

- Register and address bits are numbered from zero starting with the least significant bit. Bit ranges in register mnemonics appear between brackets. Within the brackets, the bit position values are separated by a colon, with the most significant bit first. Bit ranges in text are preceded with the word “bits.” The bit position values are separated by an en dash, with the most significant bit first:

  When SMSI[5] is zero, bits 11–8 of the Synthesizer Right Offset register (SROI[11:8]) provide a pan value that determines the stereo position of the voice.

- Programming examples appear in Courier font

```
...
OUT   279h, 00h     ; set PIDXR to 00h to select PSRPAI
OUT   0A79h, 80h    ; set address in PSRPAI
...
```

- Equations appear in Times font under a numbered caption as referenced in Equation 0-1 on page -xx

---

Equation 0-1 Audio Channel 1

$$\text{Channel\_1\_IRQ} = \text{PUACTI}[0] \bullet \text{UMCR}[3] \bullet \text{IDECI}[6] \bullet$$
$$(((\overline{\text{IDECI}[7]} \bullet \text{CIRQ}) + \text{SIRQ}) \bullet \overline{\text{UMCR}[4]}$$
$$+ \text{SBIRQ} \bullet \overline{\text{UICI}[7]} + \text{MIRQ} \bullet \text{UICI}[6])$$

- In text, function and program names appear in bold type:

  The **IwaveMemSize** function returns the size, in kilobytes, of local DRAM.

Part 1

Introducing the InterWave IC

Part 1 provides a description of the functional areas of the InterWave IC, describes the naming conventions used for the programmable registers in the IC, and provides a complete list of those registers. It also discusses the likely programming environment for the IC as well as compatibility concerns when using it with software intended for the Advanced Gravis UltraSound card.

CHAPTER

1 INTRODUCTION

This chapter lists the features of the InterWave IC and describes the various components and functional areas of the InterWave IC.

The InterWave IC is available in two versions: The 160-pin Am78C201 has an ISA Plug and Play interface, while the 144-pin Am78C202 has a non-Plug and Play interface.

InterWave IC Features

Major features of the InterWave audio IC include

- Support for Sound Blaster, AdLib, and MPU-401 software and compatibility with Advanced Gravis Ultrasound hardware
- Glueless, Plug and Play Industry Standard Architecture (PNP ISA) compliant system bus interface
- Wavetable-based stereo synthesizer
- Local memory control support for
  - — Up to four 4-Mbyte DRAM banks
  - — Up to four 2Mx16 EPROM banks
  - — 8-bit linear, 8-bit µ-law, or 16-bit linear, 44.1-kHz samples through the synthesizer
- Synthesizer support for up to 32 simultaneous voices
- Envelope control, tremolo, and vibrato for each voice
- Synthesizer support for up to eight delay-based accumulators which can provide effects such as echo, reverb, and flange
- Built-in stereo coder/decoder (codec), designed to be Crystal CS4231 compatible:
  - — Independent record and playback sample rates
  - — Sample rates up to 48 kHz
  - — 8-bit and 16-bit linear, µ-law, A-law, ADPCM, mono and stereo data formats
- Mixer with the following I/O:
  - — Four sets of stereo external inputs
  - — One set of stereo synthesizer inputs
  - — One set of stereo system-bus-sourced DAC inputs
  - — One set of stereo destined-for-system-bus ADC outputs
  - — One set of stereo external outputs
  - — One external mono input and one external mono output
- Playback (DAC) and record (ADC) FIFOs in the codec
- Sample counters and a timer in the codec

- Support for DMA transfer between system memory and local memory and between system memory and the codec record and playback FIFOs
- MPU-401-compatible MIDI port
- Game port support for up to two PC industry-standard joysticks or one joystick with four buttons
- Operating temperature 0°C–70°C
- Operating voltages 3.0 V–3.6 V and 4.75 V–5.25 V
- 160-pin PQFP package
- 144-pin TQFP package

General Description

The InterWave IC provides a complete audio subsystem that meets all major business and entertainment audio standards. The Am78C201 and 202 devices integrate a 32-voice stereo wavetable synthesizer, a 16-bit stereo audiophile codec and audio mixer with MIDI and game ports, and legacy sound card emulation hardware into a single device.

Synthesizer

The wavetable synthesizer offers 32 16-bit stereo voices, all running at a 44.1-kHz frame rate. Each voice supports frequency interpolation, envelope generation, tremolo, vibrato, panning, and volume control.

Integrated Effects Processing

An on-chip effects processor provides up to eight channels of delay-based accumulators to simulate effects such as reverb, echo, chorus, and flange. Effects can be assigned to individual voices or to any combination of voices.

Wavetable Data in Local Memory

The local memory can be either DRAM or ROM, or a combination. When DRAM is used, musical instrument patches can be swapped in and out as needed—this allows for a smaller, lower-cost, local memory. The IC supports up to 16 Mbyte of DRAM and 16 Mbyte of ROM. The InterWave Game API requires enough local memory to hold a complete patch set.

Digital Mixer

Use any or all of the 32 synthesizer voices to play and mix digital audio files. Additionally, all of the voice processing power can be applied to modifying signals, including volume, pan, frequency shift, and reverb.

Low Frequency Oscillators (LFOs)

Sixty-four LFOs provide tremolo and vibrato effects.

On-Chip 16-Bit Synthesizer Digital-to-Analog Converters (DACs)

The InterWave IC converts the stereo digital output of the synthesizer into analog form with on-chip DACs.

Patch Formats

The IC supports wavetable patches in 8-bit PCM, 16-bit PCM, or 8-bit µ-law compressed formats.

Codec

The full-duplex 16-bit audiophile stereo codec/filter is a register-compatible superset of the popular CS4231 device.

Flexible Sample Rates

You can independently select the sample rates for the record and playback paths, which range up to 44.1 kHz or 48 kHz, depending on the crystals used with the IC. A mode is provided that allows the playback sample rate to be continuously varied from 3.5 kHz–22 kHz or 5.0 kHz–32 kHz (256 steps).

Data Formats

The codec operates with a variety of data types, including 8-bit and 16-bit linear, 8-bit A-law and µ-law compressed, and 4-bit IMA ADPCM compressed.

FIFOs

Sixteen-sample record and playback FIFOs move data to and from the DACs and ADCs. Additionally, in revision C and later ICs, you can allocate very large FIFOs, from 8 bytes to 256 Kbytes, using the local memory (DRAM). These very large FIFOs are useful in non-DMA applications such as PCMCIA cards.

Serial DSP Port

The serial port allows an external digital signal processor (DSP) to connect directly to the codec record and playback paths.

Mixer

The mixer provides four external stereo input pairs, two internal stereo input pairs (codec DAC and synthesizer DAC), and one external mono input. It also provides separate stereo and mono outputs.

Bus Interface Options

The InterWave IC provides several interface options.

ISA Plug and Play Interface

The 160-pin package is a fully ISA-compliant, glueless, Plug and Play version with a selectable 8-bit or 16-bit data bus interface.

External Device Pass-Through

The InterWave IC supports connecting an external device such as a CD-ROM interface to the ISA bus through the Plug and Play interface. For applications that do not require a CD-ROM interface, use this port to supply Plug and Play support for other functions.

Reduced Pin Count ISA Bus Interface

A 144-pin ISA bus version of the device is available for applications such as PCMCIA card and laptop motherboards where Plug and Play support is not required.

Other Sound Card Support

The InterWave sound processing technology is compatible with the UltraSound audio board from Advanced Gravis (commonly referred to as GUS). Through a combination of hardware and software, the InterWave IC can emulate legacy FM sound cards. The hardware includes register access traps, status register updates, and timers. Thus, all software written for legacy sound cards and for *native-mode* UltraSound can use the InterWave IC as the hardware target. Native-mode UltraSound is the mode in which independent software vendors (ISVs) have written software to take advantage of the high-quality UltraSound synthesis capabilities.

Game and MIDI Ports

The InterWave IC includes interfaces for connecting analog game controllers (joysticks) and MIDI devices. The game port supports two joysticks and offers programmable analog scaling. The MIDI port is built around a UART with a 16-byte receive FIFO. This UART can be programmed to behave like a Motorola MC6850 UART or a MPU-401 UART. In addition, the IC contains MPU-401 emulation registers with interrupt indicators. The IC can be programmed to generate interrupts when application software writes to the MIDI UART. These interrupts can be captured by MPU-401 emulation software, which would in turn read the data written by the application and translate it into commands appropriate for the InterWave IC.

CHAPTER

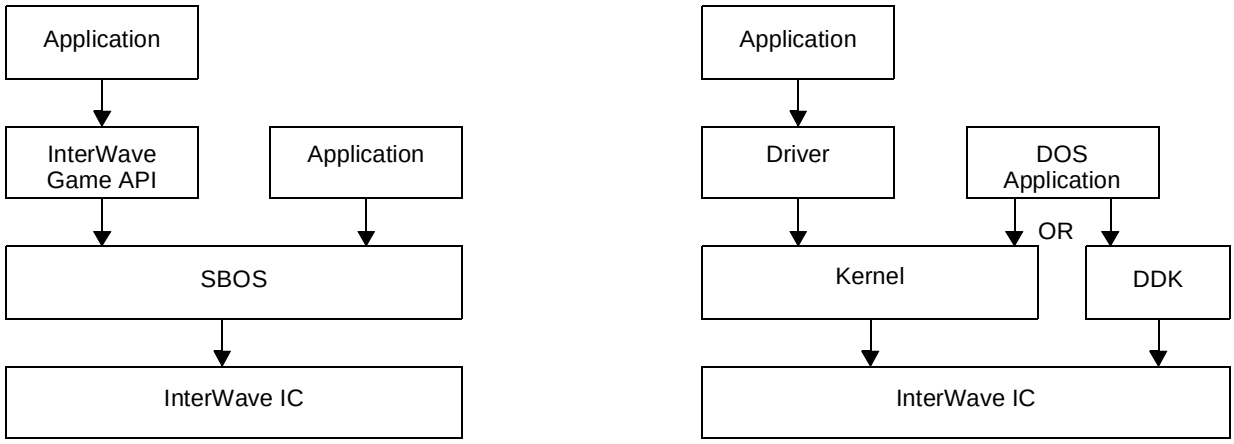
2 SOFTWARE ENVIRONMENT

The InterWave IC is designed for use in IBM-compatible personal computer systems. A number of software tools are available to assist the programmer in preparing drivers and applications for the IC.

Software Hierarchy

Figure 2-1 illustrates the basic hierarchy of an InterWave IC application.

Figure 2-1 Software Hierarchies



InterWave Game API

The InterWave Game API provides a library of functions that allow an application to communicate with resident or background applications, such as SBOS.

InterWave Driver Developer's Kit (DDK)

The InterWave DDK is an extensive library of basic functions that perform such tasks as: allocating local memory, setting mixer volume, or playing digital audio. Use these functions to build basic applications, or to create low-level device drivers.

Part 4 of this book provides a complete reference to the InterWave Game API and the InterWave DDK.

Plug and Play Support

The **iwinit.exe** program provided with the complete InterWave software package, initializes the InterWave IC-based hardware. As part of the initialization process, in the DOS and Windows 3.x environments, **iwinit.exe** configures the PNP facilities of the IC. In Windows 95, the virtual device driver (VxD) handles PNP configuration.

InterWave Kernel

The InterWave Kernel is a highly optimized set of functions designed to support the full complement of InterWave hardware functions. These functions reside between the InterWave IC and the API layer, providing a common, efficient means of supporting both standard and custom application interfaces. The Kernel functions reside in modules, one for each of the basic functions of the IC. The modules currently available are:

MIDI Plays MIDI files through the InterWave synthesizer

Digital Audio Provides high-level streaming of digital audio data. The Digital Audio module is independent of operating system, disk format, or filetypes.

- Synthesizer (playback)
- Codec (playback or record)

Digital Music Provides lower-level access to the synthesizer for sound effects or digital music. A MOD player application would use these functions.

Mixer Provides information to the application on the content and configuration of the sound equipment

UART Allows either polling or interrupts on either transmit to or receive from the MIDI UART.

Init/DeInit Initializes or deinitializes the synthesizer, codec and mixer, and kernel

Local Memory Management
Functions for accessing InterWave local memory

DMA and Programmed I/O
Digital Audio Codec uses DMA. Digital Music and Digital Audio Synthesizer use Programmed I/O.

Interrupts Functions for allocating and deallocating handlers

Voice Allocation
Management of up to 32 voices. May be allocated in any number up to the number available

AdLib-Compatible Timers
Allocate and deallocate one 80-microsecond and one 320-microsecond clock timers.

The Kernel includes the following miscellaneous modules:

- Delay
- Error Reporting
- Configuration
- PNP
- Patch library and maintenance

The Kernel also includes the following operating system and compiler dependent modules:

- DOS Real: Borland C/C++ and Microsoft C
- DOS Protected: Watcom 10.0
- OS/2—This module is still in development
- Windows 3.1 and Windows 95

Although some compiler dependent code is scattered throughout the kernel, the greatest amount of operating system and compiler dependent code is located in the OS dependent modules whose functions all begin with the **os**_ prefix. These functions handle:

- File I/O
- DMA
- Interrupt vector tables
- IRQs
- I/O to the InterWave IC

Supported Compilers

The Kernel is written in C and 80x86 assembly language. It can be used with almost all high-level language compilers, assemblers, and linkers available for the IBM-compatible PC platform. It has been tested with the following compilers:

- Borland® C++
- Microsoft C++
- Watcom C/C++

For version numbers of the compilers used, see the *InterWave Kernel Reference Manual*.

SBOS

The Sound Board Operating System (SBOS) is a device driver which emulates the functionality of legacy sound systems. It contains a Sound Blaster DSP emulator, an AdLib emulator, and a MPU-401 UART emulator. Because the MIDI sounds are actually played by the InterWave wavetable synthesizer, rather than a conventional FM synthesizer, the sound quality may be superior to that of the hardware emulated.

Operating System Support

Although the InterWave Kernel greatly simplifies the interface to the InterWave IC, the majority of application developers probably want a simpler interface than this for the following reasons:

- The complexity of code using the Kernel Interface could be unacceptably large.
- In DOS mode, static links to Kernel functions are required, necessitating relinking and redistribution of all software each time the InterWave Kernel is updated. Therefore, the InterWave Game API or one of the custom APIs should be used, when possible, for DOS programming.

For these reasons, a driver or wrapper is provided for each of the operating system environments supported by the InterWave IC:

- DOS (Real or Protected)

- Win 3.x (V86 and Windows 3.x applications)
- Win 95, (V86, Windows 3.x, and Windows 95 applications)

The application developer should write exclusively to these layers to insulate code from changes in the Kernel and the InterWave IC. The remainder of this section describes the wrappers and drivers provided with the Kernel. The interface to a Windows 3.x or Windows 95 driver is described in the appropriate Microsoft publications.

DOS

Figure 2-1 illustrates the relationship between the components of a DOS application, the Interwave Wrapper, Kernel, and SBOS. The InterWave Kernel contains a robust set of functions which require large amounts of memory by the standards of DOS conventional memory. For this reason, the DOS wrapper interface to the kernel is split into two components:

- A Real mode stub TSR occupying only 8 Kbytes of conventional memory. This TSR contains the InterWave Kernel Interface, which is used for relaying calls while making a mode transition to:
- The Protected mode TSR occupying 300 Kbytes of extended memory containing all of the functions of the Kernel and SBOS, which are accessed through static links.

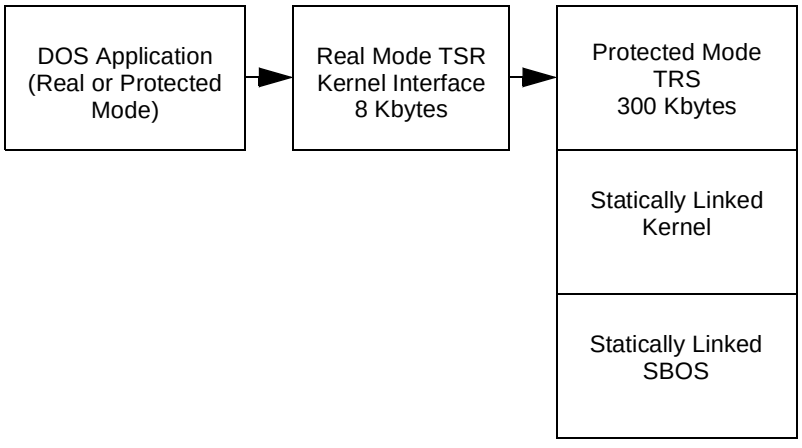
This innovative split mode TSR wrapper allows the InterWave IC to have the best of both worlds:

- State-of-the-art audio software requiring large amounts of RAM
- Operation within DOS using minimal amounts of conventional RAM

Both Real and Protected mode applications are supported by this method. The only difference is that transitions are made between Protected mode and Real mode when moving between a Protected mode application and the Real mode stub.

Whether a Protected mode application's DOS extender uses DPMI or VCPI is transparent to the InterWave split mode TSR.

Figure 2-2 DOS Split Mode TSR

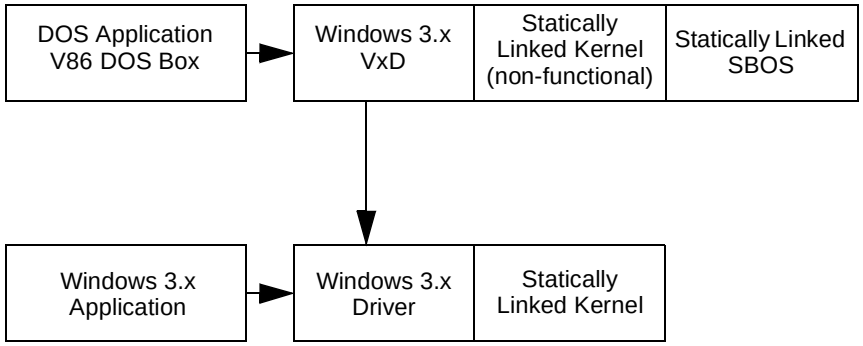


Windows 3.x

Figure 2-3 shows the interaction under Windows 3.x between Windows and DOS applications, the Windows driver, and the VxD for V86 DOS boxes. The VxD is identical to the Protected mode TSR portion of the DOS Split Mode TSR Wrapper. In DOS, it functions as a wrapper. In Windows 3.x, it functions as a VxD capable of entertaining one client V86 DOS box at a time. The kernel portion of the VxD/TSR is non-functional, the kernel functions being performed by the Windows driver. SBOS is not available to the Windows driver, but is still a functioning component of the VxD/TSR under Windows 3.x.

The InterWave Kernel's voice allocation module supports both a Windows 3.X application and a single DOS application using the InterWave IC at the same time. When a DOS application wishes to use a device, it communicates to the chip through the VxD, which in turn makes a request for the device through the Windows 3.x driver. The Windows 3.x driver then makes a request to the InterWave Kernel for the number of voices needed to *virtualize* the device. Similarly, if a Windows 3.x application attempts to use the IC, it makes a request for the device to the Windows 3.x driver, which then obtains the appropriate number of voices. The Windows 3.x driver continues to ask the InterWave Kernel for voices until the Kernel finds that it does not have enough to allocate (a limit of 32 voices). When not enough voices are available, the Windows 3.x driver tells the requestor—a VxD or Windows 3.x application—that the requested device is not available.

Figure 2-3 Windows 3.x



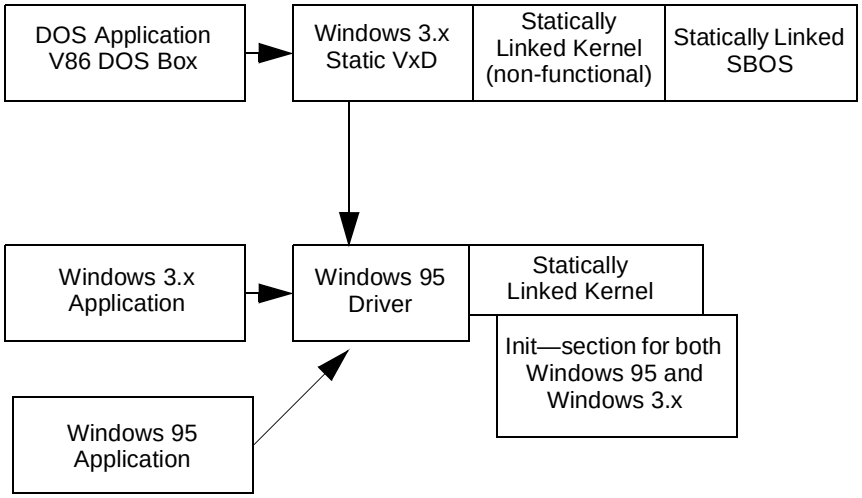
Windows 95

Figure 2-4 illustrates the interaction between DOS applications and Windows under Windows 95. The interaction is nearly identical to that under Windows 3.x. The principal difference is that the Windows 3.x driver has been replaced with a Windows 95 driver. The Init section of this driver contains code which recognizes the calling application as either a Windows 95 application or a Windows 3.x application and configures the driver accordingly.

Note: Only one Windows application, either Windows 95 or Windows 3.x, may use the InterWave IC at any given time. As before, a DOS application in a V86 box may use the chip concurrently with a Windows application.

For the purposes of the InterWave IC, the Windows 95 Stand Alone mode is functionally identical to DOS Real mode and the software functions as illustrated in Figure 2-2.

Figure 2-4 Windows 95



DRAM versus ROM Local Memory Trade-offs

When choosing how much DRAM or ROM to put on an InterWave IC-based sound card, the following questions must be addressed:

- What sound quality is required? More DRAM and ROM results in potentially better sound quality.
- How much time delay is tolerable in loading sound effects or instruments? DRAM must be loaded with sound patches. ROM, of course, is preloaded.
- Will new instruments be created? DRAM holds the new instrument patches.

Table 2-1 describes and lists the ROM patch sets available for the InterWave IC.

Table 2-1 Available ROM Patch Sets

Size (bytes)	Contents	Sample Size	How Made	Time to Load Same DRAM Set (seconds)
1M	GM Instrument set, SBOS set	8 bits	Removal of least significant bits from 16-bit 2M set	2
2M	GM Instrument set, SBOS set	16 bits	Original recording with shortening of sustain loops and other methods of decreasing the data size	4
2M	GM Instrument set, GS percussion set, SBOS set	8 bits	Removal of least significant bits from 16-bit 2M set	4
4M	GM Instrument set, GS percussion set, SBOS set	16 bits	Original recording	8

Table 2-2 lists some suggested DRAM and ROM combinations for an InterWave IC-based sound system.

Table 2-2 TDRAM and ROM Choice Space

Price and Quality	ROM (Bytes)	DRAM (Bytes)	Comments
Least expensive	1M	256K	DRAM for LFO and short, repetitious sound effects
Less expensive and Ultrasound compatible	1M	512K	Supports eight melodic and 20 percussive instruments
Less expensive and Ultrasound compatible	1M	1M	
Expensive	2M	512K	GS percussion
Expensive with good quality	4M	512K	
Expensive with excellent quality and the best UltraSound compatibility	4M	1M	
Most expensive with excellent quality	4M	2M-4M	Useful in all applications

There are four strategies for loading DRAM patch sets, each of which has its advantagesand disadvantages. Table 2-3 explores these methods.

Table 2-3 Strategies for Loading DRAM Patch Sets

Method	Advantages	Disadvantages
Pre-load the entire General MIDI set.	All instruments in the patch set are available immediately.	This method requires the maximum amount of memory.
Choose an optimal set of instruments which fit the available DRAM.	Requires no loading.	It is difficult to choose an optimal set.
Load patch sets between songs or scores.	Loading generally occurs when an application is getting other information from the hard disk, so no degradation is noticeable.	Few applications provide information about which patch sets are required for a score.
Load a patch when it is needed.	Guarantees correct instrument is played.	Degradation in performance may be noticeable—missed notes and stagger.

Sizes of Software Modules

Table 2-4 lists the amount of system memory required by each of the software modules and drivers mentioned in this chapter.

Table 2-4 Memory Requirements for Software Modules

Module Name	Size in Kbytes
DOS Split Mode TSR—conventional memory	8
DOS Split Mode TSR—extended memory	300
Windows 3.x Driver (the **interwav.drv** file)	140
IWINIT	Not resident
MIXER	Not resident
Miles Driver—MIDI	4
Miles Driver—Digital Wave	12
HMI Driver—MIDI Real Mode	100 (**interwav.com** = 0.5)
HMI Driver—Digital Wave Real Mode	190
HMI Driver—MIDI Protected Mode	120 (**interwav.com** = 0.7)
HMI Driver—Digital Wave Protected Mode	260
SBOS	20
SBOS—IWSB1024 patch set	1,024 (Local Memory)
SBOS—IWSB512 patch set	512 (Local Memory)

Note:

Memory sizes are for system memory unless otherwise noted.

CHAPTER

3 COMPATIBILITY WITH ADVANCED GRAVIS ULTRASOUND AND ENHANCEMENTS

The InterWave IC is backward compatible with the UltraSound audio board from Advanced Gravis (commonly referred to as the Gravis UltraSound or GUS) if it has local memory DRAM available. Backward compatibility means that software written for the GUS will work with a sound card based on the InterWave IC. However, the InterWave IC also includes many enhancements to the capabilities of the GUS. To run software written for the GUS, the InterWave IC must be initialized to operate in *GUS-Compatibility mode*. When the InterWave IC is configured to utilize its enhanced capabilities, it is operating in *Enhanced mode*. To enable the enhanced features added to GUS operation, set High the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]). The **iwinit.exe** program turns on Enhanced mode operation.

This chapter discusses the GUS-compatibility facilities of the InterWave IC and the differences between GUS-Compatibility mode and Enhanced mode, including the following topics:

- Frame expansion
- Auto-increment mode
- Local memory addressing
- PCM operation
- Volume and Frequency LFOs
- Voice data in ROM
- µ-law voice data format
- Separate left and right stereo offset registers
- Voice deactivation
- Effects processing
- DMA control
- GUS-Compatibility mode reset

Frame Expansion

The InterWave IC can handle up to 32 voices at the full 44.1-kHz sampling rate. The GUS can maintain the 44.1-kHz sampling rate for only up to 14 voices. In GUS-Compatibility mode, approximately 1.6 µs are added to the sample period for each active voice over 14, which reduces the sampling rate accordingly. This increase of the sample period is called *frame expansion*. For more information, see “GUS Frame Expansion” on page 7-29.

Auto-Increment Mode

To make programming of synthesizer voices easier, the InterWave IC provides a method of writing to all of the voice-specific registers for a particular voice without having to write a new index value to the General Index register (IGIDXR) for each register. This mode,

called *auto-increment* mode, is available only in Enhanced mode. For information about auto-increment mode, see “Programming Voice-Specific Registers” on page 7-30.

Local Memory Addressing

The GUS synthesizer supports up to 1 Mbyte of local memory for storing wavetable data. The InterWave synthesizer can address up to 4 Mbytes of ROM for wavetable data and up to 16 Mbytes of DRAM for wavetable data and local memory FIFOs. To maintain compatibility with software written for the GUS, the two most significant bits of the synthesizer address in all synthesizer registers (address bits 21–20) and the four most significant bits of the local memory address in all local memory address registers (address bits 23–20) are held Low (0) when the InterWave IC is in GUS-Compatibility mode.

For information about how the InterWave synthesizer accesses local memory, see “Addressing Wavetable Data” on page 7-9. For information about the synthesizer address registers, see the register reference pages in Chapter 14, “Synthesizer Registers.”

For information about how the InterWave IC accesses local memory for data transfers, see “Accessing Local Memory” on page 8-8 and “DMA Data Transfers” on page 8-11. For information about the local memory address registers, see the register reference pages in Chapter 15, “Local Memory Control Registers.”

PCM Operation

To support the continuous playback of PCM data, the InterWave IC provides a method by which sample interpolation can take place between the data addressed by the synthesizer end address registers and the data addressed by the synthesizer start address registers. This capability is available only in Enhanced mode. For information about PCM operation, see “Digital Audio Files and PCM Operation Mode” on page 7-28.

Volume and Frequency LFOs

The InterWave IC contains 64 low frequency oscillators (LFOs), two for each of the 32 synthesizer voices. These LFOs can be used to produce tremolo (amplitude modulation) and vibrato (frequency modulation) effects. Adding LFO values to the synthesizer voice can be done only when the InterWave IC is in Enhanced mode. For more information about the LFOs, see “LFOs for Tremolo and Vibrato” on page 7-21.

Voice Data in ROM

The InterWave synthesizer can access voice data from external ROM. Enable the reading of voice data from ROM by setting high the *ROM* bit of the Synthesizer Mode Select register (SMSI[7]) for a particular voice. Features controlled through SMSI are available only in Enhanced mode.

µ-Law Voice Data Format

The InterWave synthesizer can decompress voice data stored in local memory in µ-law compressed format. Enable the handling of µ-law compressed voice data by setting high the *µ-Law* bit of the Synthesizer Mode Select register (SMSI[6]) for a particular voice. Features controlled through SMSI are available only in Enhanced mode. For more information about reading µ-law voice data, see “SMSI—Synthesizer Mode Select” on page 14-14.

Separate Left And Right Stereo Offset Registers

The GUS determines a voice's position in the stereo field by a single 4-bit pan value, with 0 being full left and 15 being full right. The InterWave IC provides an *offset mode* that uses two 12-bit values, one each for the left and right offset, to place a voice more precisely in the stereo field. The registers that contain these offset values also serve as an additional volume control because they operate by attenuating the left and right synthesizer outputs. Enable this offset mode by setting High the *Offset Enable* bit of the Synthesizer Mode Select register (SMSI[5]) for a particular voice. Features controlled through SMSI are available only in Enhanced mode. For more information about the offset mode, see "Synthesizer Offset Registers" on page 14-12.

Voice Deactivation

The InterWave IC provides the ability to deactivate a voice. When a voice is deactivated, the voice does not consume memory cycles for processing, thus allowing more local memory cycles for other activities. Deactivating unused voices also reduces noise. Deactivate a voice by setting the *Deactivate Voice* bit of the Synthesizer Mode Select register (SMSI[5]) High for a particular voice. Features controlled through SMSI are available only in Enhanced mode.

Effects Processing

The InterWave synthesizer can utilize a voice as an effects processor. The InterWave IC accomplishes such effects processing by writing the synthesizer output to local memory and then reading it some number of frames later. Select a voice as an effects processor by setting the *Effects Processor Enable* bit of the Synthesizer Mode Select register (SMSI[0]) High for a particular voice. Features controlled through SMSI are available only in Enhanced mode. For more information on effects processing, see "Delay-Based Effects" on page 7-26.

DMA Control

The InterWave IC supports two kinds of DMA transfers between system and local memory: GUS-compatible DMA and interleaved DMA. Use the LMC DMA Control register (LDMACI) to control GUS-compatible DMA and the LMC DMA Start Address High and LMC DMA Start Address Low registers (LDSAHI and LDSALI) to specify the DMA address. For more information about DMA transfers, see "DMA Data Transfers" on page 8-11. For information about the DMA registers, see the register reference pages in Chapter 15, "Local Memory Control Registers."

GUS-Compatibility Mode Reset

After the InterWave IC has been operating in Enhanced mode, the various GUS-compatibility features should be reset before attempting to operate the IC in GUS-Compatibility mode. For information about resetting the GUS features, see "URSTI—GUS Reset" on page 12-14.

Part 2

Programming the InterWave IC

Programming the InterWave IC consists of writing to and reading from user-accessible registers. This part first discusses topics that relate to all components of the IC and then describes how to program each of the major components:

- System control
- Codec
- Synthesizer
- Local memory control
- Game and MIDI ports
- GUS Compatibility

CHAPTER

4 GLOBAL PROGRAMMING TOPICS

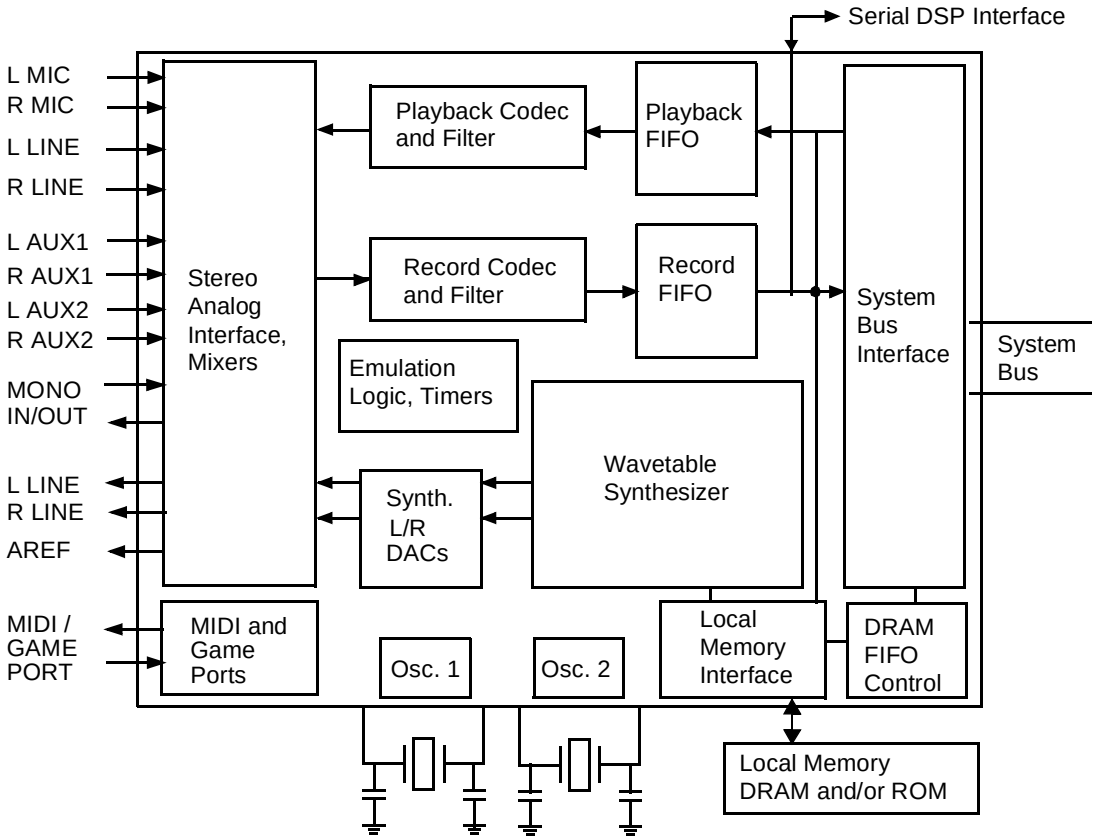
This chapter deals with programming topics that apply to the InterWave IC as a whole:

- Data paths
- Accessing InterWave registers
- Initialization
- Power modes
- Interrupt structure
- Clocks

Data Paths

Figure 4-1 illustrates the flow of data in the InterWave IC.

Figure 4-1 InterWave Data Paths



Accessing InterWave Registers

To control the InterWave IC, software must write to or read from certain I/O addresses associated with programmable registers in the InterWave IC. Depending on how the InterWave IC-based hardware is configured, there are two ways to address the InterWave registers:

- Normal, or internal, decoding mode
- External decoding mode

See “Accessing InterWave Registers with the DDK” on page 18-13 for information about two DDK functions that simplify the process of accessing registers.

Normal or Internal Decoding

Internal decoding mode allows software to access InterWave registers using base I/O addresses programmed into the PNP address registers. Internal decoding mode is further separated into system mode and card mode.

Note: The following modes are* not *programmable. The hardware may be set up to work in a specific mode or to switch between these modes using jumper settings.

System Mode

When InterWave IC-based sound hardware has no Plug and Play serial EEPROM available, or has one that has not yet been programmed, the Plug and Play isolation protocol must be bypassed. In this mode, called *system* mode, software must configure the PNP registers using the following procedure:

Place the IC into isolation mode.

1. Write the card select number (CSN) to I/O address 201h (fixed).
2. Configure the logical device.

Card Mode

In *card* mode, a serial EEPROM is available and has been programmed to report the resource requirements of the InterWave IC-based sound hardware. The IC is fully PNP-compliant and is expected to participate in the PNP isolation process. Standard PNP software should be able to configure the hardware. Board-specific (non-PNP) configuration requires vendor-supplied software (device drivers or boot-time initialization programs).

I/O Address Spaces

The I/O addresses of most of the InterWave IC registers depend on one of several base address values. Table 4-1 lists the address spaces, along with references to detailed information about the registers used to set the base addresses. Software can relocate all of these address spaces except the PNP Index Address (PIDXR) and the PNP Write Data Port (PNPWRP).

Note: In internal decoding mode, set the base addresses of these address spaces before attempting to write to or read from any InterWave register.

Each of the relocatable address spaces has one or two registers associated with it for setting and reading the base address of the space. Access the address-setting registers for all but two of the relocatable address spaces by first writing an index value to the Plug and Play Index register (PIDXR) at I/O address 279h, then writing the address value to the Plug and Play Write Data Port (PNPWRP) at address A79h or reading from the Plug and

Play Read Data Port (PNPRDP). These address registers are read through PNPRDP, whose address is set during the PNP isolation process.

The remaining two relocatable addresses, those for the General Purpose Register 1 (UGP1I) and General Purpose Register 2 (UGP2I), get their addresses from a combination of the Compatibility register (ICMPTI) and either the General Purpose Register 1 Addressregister (UGPA1I) or the General Purpose Register 2 Address register (UGPA2I).

Table 4-1 InterWave Address Spaces

Mnemonic	Description	Base Address (SA11–SA0)†	Ref. Page
P2XR	**GUS-Compatibility**—A block of ten addresses within 16 spaces used primarily for compatibility with existing sound cards. The four least-significant bits of the address span the 16 spaces, but only P2XR+0, P2XR+6, and P2XR+8 through F are used.	0*, 0*, P2X0HI[1:0], P2X0LI[7:4], X, X, X, X	12-25
P3XR	**MIDI and Synthesizer**—A block of eight consecutive addresses used primarily to address the synthesizer and MIDI functions. The three least-significant bits span the eight spaces, but P3XR+6 is not used.	0*, 0*, P3X0HI[1:0], P3X0LI[7:3], X, X, X, X	12-25
PCODAR	**Codec**—A block of four consecutive addresses used to address the codec function.	0*, 0*, PHCAI[1:0], PLCAI[7:2], X, X	12-25
PCDRAR	**External Interface**—A block of eight consecutive addresses used for accesses to the external CD-ROM interface. When this I/O block is decoded, it causes the $\overline{\text{EX\_CS}}$ signal to become active.	0*, 0*, PRAHI[1:0], PRALI[7:3], X, X, X	12-25
PATAAR	**ATAPI**—A block of two consecutive addresses used in conjunction with PCDRAR to communicate with an ATAPI CD-ROM. When this I/O block is decoded, it causes the $\overline{\text{EX\_CS}}$ signal to become active.	0*, 0*, PATAHI[1:0], PATALI[7:1], X	12-25
PIDXR	**Plug and Play Index register**—The single-byte index to all of the standard PNP-ISA registers.	279h	12-21
PNPWRP	**Plug and Play Write Data Port**—The single-byte data port through which all of the standard PNP-ISA registers are written.	A79h	12-21
PNPRDP	**Plug and Play Read Data Port**—The single-byte data port through which all of the standard PNP-ISA registers are read.	0, 0, PSRPAI[7:0], 1, 1	12-21
UGPA1I, UGPA2I (P401AR)	**General Purpose Address Registers 1 and 2**—The general purpose registers (UGP1I and UGP2I) are single-byte registers used for compatibility with legacy sound cards. These registers are typically placed in two consecutive locations and pointed to by P401AR, a standard PNP register that specifies the location of these two emulation addresses. P401AR is necessary to communicate to the PNP BIOS and to application software that these addresses are being used.		12-11, 12-11, 12-25
P201AR	**Game Control (Joystick)**—Points to a single-byte block that is used for game control (the Game Control register (GCCR)). This port is typically set to address 201h to match the legacy location of this function.	0*, 0*, P201HI[1:0], P201LI[7:6], 0, 0, 0, 0, 0, 1	12-25
P388AR	**AdLib Emulation**—Points to a two-byte block that is used to capture activity for legacy AdLib registers; however, this block has little value unless it is set to the legacy position of 388h–389h.	0*, 0*, P388HI[1:0], P388LI[7:6], 0, 0, 1, 0, 0, X	12-25

Notes:

†SA refers to system address pins.

**SA11–SA10 may or may not be decoded based on the state of $\overline{EX\_CS}$ at power up.*

Direct Registers

Software can access some of the programmable registers in the InterWave IC directly through their I/O addresses. Table 4-2 on page 4-4 lists the eight groups of functions in the IC that contain such *direct* registers and their associated direct addresses.

Table 4-2 Direct Addresses

Function	Direct Addresses
Codec	PCODAR+0 through PCODAR+3
Game, MIDI port	P201AR, P3XR+0, P3XR+1
System control	P3XR+3, P3XR+(4, 5)
Local memory control	P3XR+7 (also contains registers in P3XR+(4, 5))
Plug and Play ISA	PIDXR, PNPWRP, PNPRDP
CD-ROM	PCDRAR+0 through PCDRAR+7, PATAAR+(0, 1)
Synthesizer	P3XR+2 (also contains registers in P3XR+(4, 5))
GUS and AdLib–Sound Blaster emulation	P2XR+0, P2XR+6, P2XR+8 through P2XR+0Fh, P388AR+(0, 1), UGPA1I, UGPA2I (also contains registers in P3XR+(4, 5))

The mnemonics for the registers that use these direct addresses end in R for register, or P for port for easy identification. In the register list tables of Chapter 11, "Register Summary," these registers contain no value in the *Index* column because it is not necessary to write an index value to some other register before reading from or writing to the register's I/O address location.

Indirect Registers

Most of the programmable registers in the InterWave IC must have an index value set before reading or writing the register's I/O address location. The register mnemonics for these *indirect* registers end in I. In the register list tables of Chapter 11, "Register Summary," these registers include a value in the *Index* column. This value indicates which register receives the index value and what the value is. Some of the registers have different index values and even different index registers for reading and writing. To access the indirect register, write the index value to the appropriate index register, then write to or read from the register's I/O address.

External Decoding

External decoding mode allows access to the InterWave registers without first having to configure the PNP registers. In this mode, the direct registers respond to preset 6-bit I/O addresses as specified in Table 4-3. In this mode, access the indirect registers by writing their index values to the appropriate direct register using the information in this table.

Note:

It is not valid for bits 5 and 4 to both be Low at the same time.

Table 4-3 External Decoding Mode I/O Addresses

Bit 5	Bit 4	Bits 3–0	Register	Equivalent Internal-Decoding-Mode Address
1	0	0h	UMCR	P2XR + 0h
1	0	1h	GGCR, PCSNBR	P201AR, 201h (fixed)
1	0	2h	PIDXR	279h (12-bit fixed)
1	0	3h	PNPWRP, PNPRDP	A79h (12-bit fixed), PNPRDP
1	0	4h	-	
1	0	5h	-	
1	0	6h	UISR, U2X6R	P2XR + 6h
1	0	7h	-	
1	0	8h	UACWR, UASRR	P2XR + 8h, 388h (fixed)
1	0	9h	UADR	P2XR + 9h, 389h (fixed)
1	0	Ah	UACRR, UASWR	P2XR + Ah
1	0	Bh	UHRDP	P2XR + Bh
1	0	Ch	UI2XCR	P2XR + Ch
1	0	Dh	U2XCR	P2XR + Dh
1	0	Eh	U2XER	P2XR + Eh
1	0	Fh	URCR, USRR	P2XR + Fh
0	1	0h	GMCR, GMSR	P3XR + 0h
0	1	1h	GMTDR, GMRDR	P3XR + 1h
0	1	2h	SVSR	P3XR + 2h
0	1	3h	IGIDXR	P3XR + 3h
0	1	4h	I16DP (low byte)	P3XR + 4h
0	1	5h	I16DP (high), I8DP	P3XR + (4-5)h, P3XR + 5h
0	1	6h	-	
0	1	7h	LMBDR	P3XR + 7h
0	1	8h	-	
0	1	9h	-	
0	1	Ah	-	
0	1	Bh	-	
0	1	Ch	CIDXR	PCODAR + 0h
0	1	Dh	CDATAP	PCODAR + 1h
0	1	Eh	CSR1R	PCODAR + 2h
0	1	Fh	CPDR, CRDR	PCODAR + 3h

Reset

The InterWave IC can be reset by hardware or by software. In addition, the IC can be put into and brought out of a low-power suspend mode. This section covers the following topics:

- Power-up and hardware reset
- Software reset
- Suspend mode

Power-Up and Hardware Reset

After power-up or hardware reset, all InterWave registers default to their initial states. Software drivers or applications must write appropriate values for the chip's hardware environment and software to the programmable registers.

Information about register default states can be found in the following places:

- The register list tables in Chapter 11, "Register Summary"
- The initialization section of each of the programming chapters in Part 2, "Programming the InterWave IC"
- The register details in Part , "InterWave Registers Reference"

There are no provisions to allow for recovery from power-off. If power is removed from the IC, it must be completely reinitialized.

Software Reset

When the IC is in the PNP configuration state, setting the *Reset* bit of the PNP Configuration Control Command register (PCCCI[0]) to High causes a software reset. The PNP Set PNPRDP Address register (PSRPAI), the PNP Card Select Number register (PCSNI), and the PNP state itself are not affected. Therefore, the PNP resources must be reprogrammed but there is no need to perform PNP isolation again.

Software reset works only when the IC is not in the PNP wait-for-key state. After a software reset, the software should wait about 10 ms before accessing the IC again.

Note: Software reset resets ALL PNP devices.

For more information about the PCCCI register, see "PCCCI—PNP Configuration Control Command" on page 12-22.

Suspend Mode

A hardware signal can place the InterWave IC in a low-power mode of operation called suspend mode. Suspend mode isolates, or effectively detaches, the IC from the ISA bus. Software cannot directly initiate this mode.

Local memory DRAM refreshing can be maintained during suspend mode. See "Returning from Suspend Mode" on page 8-5.

After coming back from suspend mode—if DRAM refreshing was maintained during the suspend—the synthesizer resumes operation with the next frame pointed to by its registers.

For more details about suspend mode, see the *InterWave IC Hardware Designer's Guide*, available from AMD.

Initialization

Software must initialize the InterWave IC when either of the following events occur:

- Power-up, or hardware reset
- Software reset, by way of PCCCI

When the IC is reset, the following process must take place before the IC can be used again:

1. Configure the standard PNP registers.

The PNP registers are common to all PNP-compliant ISA adapters or devices and they must be programmed by Plug and Play software before the device can operate correctly. Programming the PNP registers establishes the I/O space, selects DMA channels, and selects interrupt lines. The software used to program these registers can be in one of several forms: a Plug and Play BIOS, a Plug and Play manager, a Plug and Play operating system, or a vendor-supplied driver or program. AMD provides an initialization program with the InterWave DDK called **iwinit.exe** that can be used to program these registers at system boot time. For more information about the PNP initialization process, see “Plug and Play Functions” on page 5-13.

Note: A software reset does not change the PNP Read Data Port (PNPRDP), the PNP Read Data Address (PSRPAI), the PNP Card Select Number (PCSNI), or the PNP state (wait-for-key, isolation, or configuration).

2. Configure InterWave-specific functions.

After the standard PNP registers have been programmed and I/O space, IRQ, and DMA resources allocated, several InterWave-specific attributes must be initialized. These attributes include DRAM and ROM configuration, the mixer signal paths, and GUS-compatibility or enhanced mode operation. The **iwinit.exe** program takes care of all necessary InterWave-specific configuration.

Programmable Power Modes

Use the PNP Power Mode register (PPWRI) to reduce the power being consumed by various blocks of logic within the IC and to place it in shut-down mode. See “PPWRI—PNP Power Mode” on page 12-29 for details. These power modes differ from the suspend mode in that they are programmable and they do not detach the IC from the ISA bus.

The PPWRI register can place various sections of the IC in low-power mode and disable certain clocks. PPWRI controls the power mode for the following functions:

- 24.576 MHz oscillator
- Local memory control
- Synthesizer
- Codec playback signal path
- Codec record signal path
- Codec analog circuitry (inputs and outputs)
- Game and MIDI ports

The InterWave IC is placed in shut-down mode by putting all of the above functions in low-power mode.

Interrupt Structure

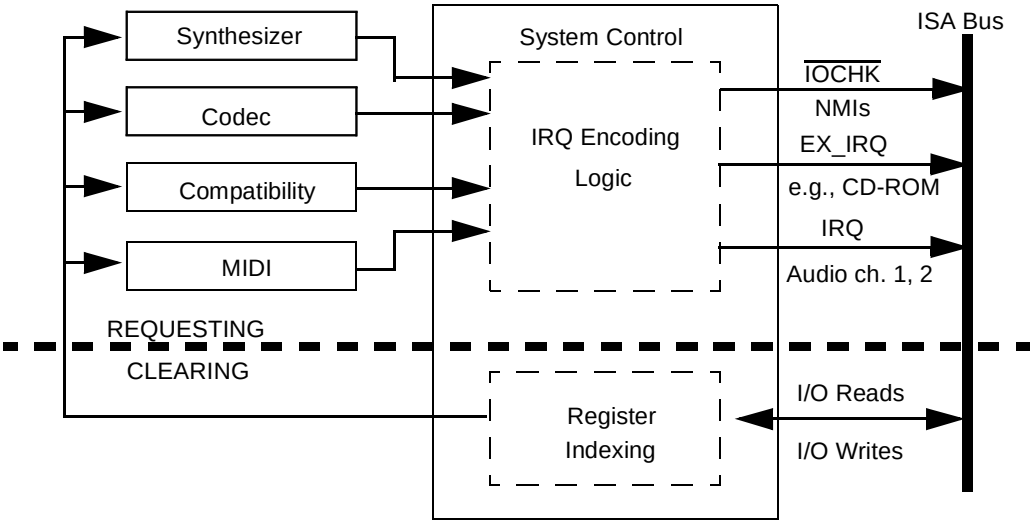
Certain functional modules can generate interrupts, which the system control module encodes into one of three IRQ channels chosen from system IRQs 2/9, 3, 4, 5, 7, 10, 11, 12, and 15.

The software clears InterWave interrupts by reading from or writing to the InterWave registers associated with the interrupt. The register reference information found in Part , “InterWave Registers Reference,” provides details on how to enable, read, and clear interrupts using the InterWave registers.

Figure 4-2 shows how the interrupt requests link to the ISA bus.

For a more detailed discussion of interrupt events, see “Interrupt Level Selection” on page 5-7.

Figure 4-2 Interrupt Structure



Clocks

The InterWave IC uses two external crystals: 24.576 MHz and 16.9344 MHz. If the 48-kHz family of codec sampling rates is not required, the 24.576 MHz crystal is not necessary. However, the 24.576 MHz crystal is required for GUS compatibility and for the full range of possible sampling rates.

The chapters in Part , “InterWave Registers Reference.” provide details on selecting clocks, setting timer load values, and handling clock interrupts using the InterWave registers.

CHAPTER

5 SYSTEM CONTROL FUNCTIONS

The InterWave system control module contains the interrupt and DMA channel selection logic, the interfaces to the ISA (system) bus, the Plug and Play (PNP) ISA logic, and the compatibility logic.

This chapter covers the following topics:

- System control basics—introduction to the system control concepts
- Data paths—InterWave signal flow
- Register overview—lists of system control functions available through the programmable registers
- Initialization—start-up procedures after power up, reset, or return from a low-power mode
- Interrupt and DMA channel selection—reporting and clearing
- System bus interface—ISA bus access
- Plug and Play functions—PNP ISA programming details
- Compatibility logic—logic and registers for compatibility with existing game-card software
- Programming tips and examples

System Control Basics

The following functional areas comprise the system control module:

Initialization: The InterWave circuits can be in various states of inactivity due to initial power-up, reset, or programmed low-power modes. The chip is initialized by writing to the proper registers.

Interrupt level selection:
Various InterWave modules request interrupts when service is needed from the host system. The system control module routes these requests to the proper IRQ (system-bus interrupt request) level.

DMA channel selection:
System Control combines DMA access requests from other InterWave modules into one of six DMA access request (DRQ) channels.

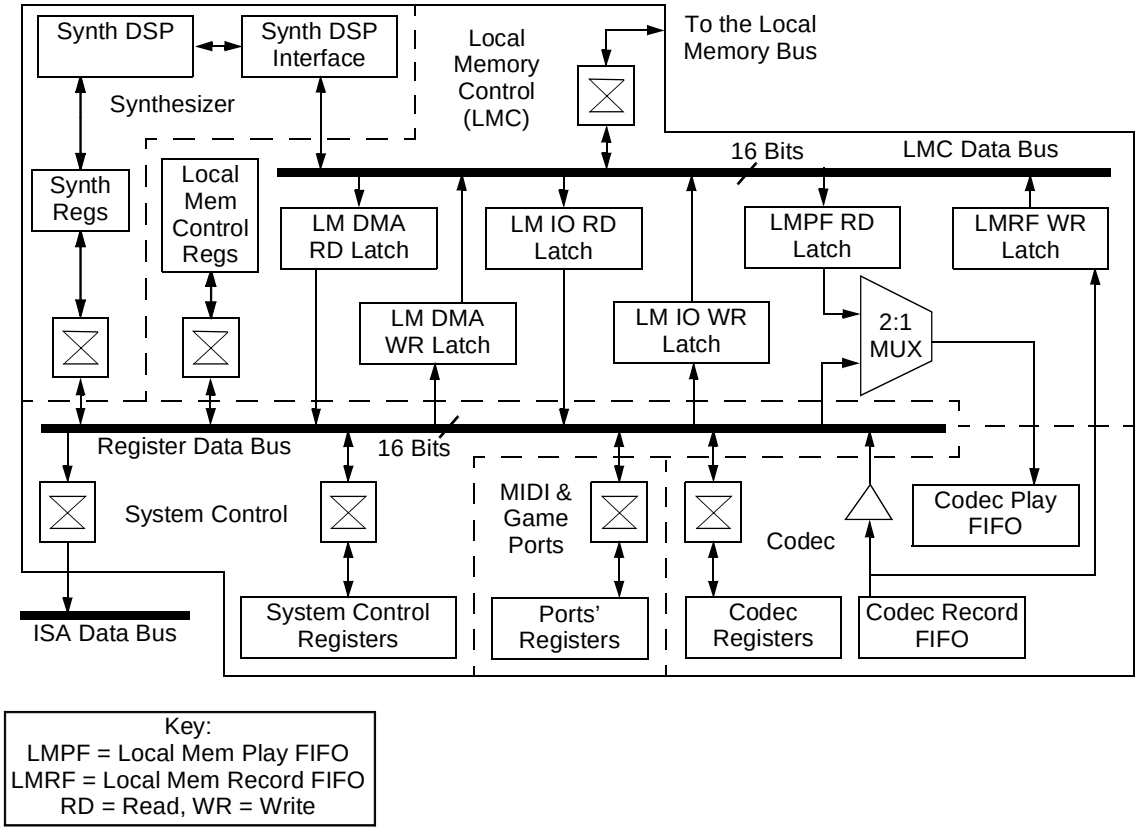
Plug and Play functions:
The Plug and Play logic implements the Plug and Play ISA specification.

Compatibility logic:
Compatibility logic provided support for existing game-card software.

System Control Data Paths

Figure 5-1 illustrates the flow of data between InterWave modules.

Figure 5-1 InterWave System Control Data Paths



Register Overview

The following tables outline the system control functions available through the InterWave programmable registers. For detailed information about a particular register, see the reference page listed in the last column of the table. These references point to the applicable section in Chapter 12, "System Control Registers."

Table 5-1 through Table 5-7 group the system control functions into the following areas:

- General control functions
- PNP functions
- DMA and non-emulation IRQ functions
- Emulation IRQ functions
- Emulation and compatibility control functions
- Game and MIDI port functions
- Audio I/O functions

Table 5-1 General Control Functions

Function	Register and Bit Field	Reference
Enable the synthesizer DAC	URSTI[1]	12-14
Select serial transfer mode—switches the codec FIFOs and synthesizer DSP data into various paths; switches the external-device pins to external serial port pins	ICMPTI[7:5]	12-15
Enable reading and writing of codec registers	IDECI[3]	12-16
Enable reading and writing of AdLib command, status, and data registers	IDECI[2]	12-16
Enable reading and writing and 2XE registers	IDECI[1]	12-16
Enable reading and writing of Sound Blaster 2X8, 2X9, and 2XA registers	IDECI[0]	12-16
Retrieve version number of die	IVERI[7:4]	12-17
Enable reading of UADR, URCR, and GMCR registers	IVERI[3]	12-17
Enable power to internal pull-up resistors	IVERI[2]	12-17
Enable MPU-401 emulation	IVERI[1]	12-17
Lock out hidden registers	IVERI[0]	12-17
Select GPOUT1–GPOUT0 codec flags	IEIRQI[7]	12-20
Read status of 16-bit I/O decoding	IEIRQI[6]	12-20

Table 5-2 PNP Functions

Function	Register and Bit Field	Reference
Write card select number to InterWave IC (system mode)	PCSNBR	12-21
Write to PNP registers (indexed by PIDXR)	PNPWRP	12-21
Read from PNP registers (indexed by PIDXR)	PNPRDP	12-21
Set read data port address	PSRPAI	12-22
Check PNP isolation state (See “Isolation State” on page 5-16)	PISOCI	12-22
Reset card select number	PCCCI[2]	12-22
Enter wait-for-key state	PCCCI[1]	12-22
Reset InterWave IC (if not in wait-for-key state)	PCCCI[0]	12-22
“Wake” the PNP device—enter isolation or configuration state	PWAKEI	12-23
Read PNP resource data from the serial EEPROM	PRESDI	12-23
Read status of PNP resource data register	PRESSI	12-23
Set the card select number (CSN) and enter configuration state	PCSNI	12-23
Index the PNP address space into logical devices	PLDNI	12-23
Activate audio functions	PUACTI	12-24
Activate external device functions	PRACTI	12-24
Activate game port functions	PGACTI	12-24
Activate AdLib–Sound Blaster functions	PSACTI	12-24
Activate MPU-401 functions	PMACTI	12-24
Check for audio I/O address conflicts	PURCI	12-24
Check for external device I/O address conflicts	PRRCI	12-24
Check for game port I/O address conflicts	PGRCI	12-24
Check for AdLib–Sound Blaster I/O address conflicts	PSRCI	12-24
Check for MPU-401 I/O address conflicts	PMRCI	12-24
Set I/O address for various address blocks	Table 12-3	12-25

Table 5-2 PNP Functions (Continued)

Function	Register and Bit Field	Reference
Select audio channel 1 IRQ number	PUI1SI	12-25
Select audio channel 1 IRQ type	PUI1TI	12-26
Select audio channel 2 IRQ number	PUI2SI	12-25
Select audio channel 2 IRQ type	PUI2TI	12-26
Select external function IRQ number	PRISI	12-25
Select external function IRQ type	PRITI	12-26
Select AdLib–Sound Blaster IRQ number	PSBISI	12-25
Select AdLib–Sound Blaster IRQ type	PSBITI	12-26
Select MPU-401 IRQ number	PMISI	12-25
Select MPU-401 IRQ type	PMITI	12-26
Select the DMA request number for audio DMA channel 1	PUD1SI	12-27
Select the DMA request number for audio DMA channel 2	PUD2SI	12-27
Select the DMA request number for the external device DMA channel	PRDSI	12-27
Specify the output-low drive capability of the ISA data bus	PSEENI[1]	12-28
Select the serial EEPROM control mode	PSEENI[0]	12-28
Set control parameters for PNP serial EEPROM	PSECI	12-28
Set low-power modes, disable clocks	PPWRI	12-29

Table 5-3 DMA and Non-Emulation IRQ Functions

Function	Register and Bit Field	Reference
Select between Interrupt Control register (UICI) and DMA Control register (UDCI)	UMCR[6]	12-1
OR all synthesizer and codec IRQs to channel 2 and mask channel 1	UMCR[4]	12-1
Enable audio device IRQ and DMA	UMCR[3]	12-1
Read status of DMA terminal count (TC) IRQ	UISR[7]	12-2
Read status of volume loop IRQ	UISR[6]	12-2
Read status of address loop IRQ	UISR[5]	12-2
Read status of IRQ and DMA Enable bit (UMCR[3])	USRR[1]	12-7
Select extra interrupt	UDCI[7]	12-8
Combine DMA channels	UDCI[6]	12-8
Select DMA channel 1	UDCI[2:0]	12-8
Select DMA channel 2	UDCI[5:3]	12-8
Combine IRQ channels	UICI[6]	12-9
Select IRQ channel 1	UICI[2:0]	12-9
Select IRQ channel 2	UICI[5:3]	12-9
Clear all IRQs in USRR	UCLRII	12-11
Enable synthesizer IRQs	URSTI[2]	12-14
Select codec IRQ channel	IDECI[7]	12-16
Enable IRQs on channel 1 or 2	IDECI[6:5]	12-16
Enable NMIs	IDECI[4]	12-16

Table 5-4 Emulation IRQ Functions

Function	Register and Bit Field	Reference
Enable AdLib timers 1 and 2 IRQs	UASBCI[3:2]	12-12
Enable interrupt caused by write to AdLib Data register (UADR)	UASBCI[1]	12-12
Enable Sound Blaster 2X6 and 2XC IRQs	UASBCI[5]	12-12
OR of AdLib–Sound Blaster register IRQs	UISR[4]	12-2
Read status of AdLib timer 1 IRQ	UISR[3]	12-2
Read status of AdLib timer 2 IRQ	UISR[2]	12-2
Read status of MIDI receive IRQ	UISR[1]	12-2
Read status of MIDI transmit IRQ	UISR[0]	12-2
Enable IRQs caused by reads of Sound Blaster register 2XE	URCR[7]	12-6
Enable IRQs caused by reads and writes of General Purpose registers 1 and 2	URCR[4:3]	12-6
Read status of 2XE IRQ	USRR[7]	12-7
Read status of General Purpose registers 1 and 2 read IRQs	USRR[4,6]	12-7
Read status of General Purpose registers 1 and 2 write IRQs	USRR[3,5]	12-7
Redirect all AdLib–Sound Blaster IRQs to $\overline{\text{IOCHK}}$ (NMI)	UICI[7]	12-9
Read status of AdLib timer 1 IRQ (maskable)	UASRR[6]	12-3
Read status of AdLib timer 2 IRQ (maskable)	UASRR[5]	12-3
Read status of AdLib timer 1 IRQ (non-maskable)	UASRR[2]	12-3
Read status of AdLib timer 2 IRQ (non-maskable)	UASRR[1]	12-3
OR of AdLib timers 1 and 2 IRQs (maskable)	UASRR[7]	12-3
Reset AdLib timers 1 and 2 IRQs (maskable)	UADR[7]	12-4
Mask AdLib timers 1 and 2 IRQs (maskable)	UADR[6:5]	12-4
Read status of write to 2XC IRQ	UASRR[4]	12-3
Read status of write to 2X6 IRQ	UASRR[3]	12-3
Read status of write to AdLib Data register (UADR) IRQ	UASRR[0]	12-3
Set MPU-401 IRQ	IEIRQI[1]	12-20
Set Sound Blaster IRQ	IEIRQI[0]	12-20

Table 5-5 Emulation and Compatibility Control Functions

Function	Register	Reference
Enable General Purpose register access	URCR[6]	12-6
Select which register is accessed through GUS Hidden Register Data Port (UHRDP)	URCR[2:0]	12-6
Enable toggling of UI2XCR[7]	URCR[5]	12-6
Enable compatibility (IRQ and DMA channel selection through UHRDP)	ICMPTI[4]	12-15
Read/write data for AdLib–Sound Blaster/MPU-401 compatibility	UGP1I	12-10
Read/write data for AdLib–Sound Blaster/MPU-401 compatibility	UGP2I	12-10
Specify emulation address for General Purpose Register 1	ICMPTI[1:0] & UGPA1I	12-15 & 12-11
Specify emulation address for General Purpose Register 2	ICMPTI[3:2] & UGPA2I	12-15 & 12-11
Enable AdLib timer test	UASBCI[4]	12-12
Enable AdLib auto-timer mode	UASBCI[0]	12-12
Set Sound Blaster IRQ bit in AdLib Status register (UASRR[3])	U2X6R	12-3
Access Sound Blaster 2XC register and trigger an interrupt	UI2XCR	12-5
Access Sound Blaster 2XC register without triggering an interrupt	U2XCR	12-6
Access Sound Blaster 2XE register and trigger an interrupt	U2XER	12-6
Read from the InterWave synthesizer by AdLib applications	UACRR	12-3
Write to the InterWave synthesizer by AdLib applications	UACWR	12-3
Start AdLib timers 1 and 2	UADR[1:0]	12-4
Reset GUS-compatibility features	URSTI[0]	12-14
Set load values for AdLib timers 1 and 2*	UAT1I, UAT2I	12-13
Enable reading of UART receive buffer	IEMUAI[7]	12-18
Enable reading of UART status	IEMUAI[6]	12-18
Enable reading of UGP2I through emulation address	IEMUAI[5]	12-18
Enable reading of UGP1I through emulation address	IEMUAI[4]	12-18
Enable writing to UART transmit buffer	IEMUAI[3]	12-18
Enable writing to UART command buffer	IEMUAI[2]	12-18
Enable writing to UGP2I through emulation address	IEMUAI[1]	12-18
Enable writing to UGP1I through emulation address	IEMUAI[0]	12-18
Enable MIDI receive data	IEMUBI[7]	12-19
Enable MIDI transmit data	IEMUBI[6]	12-19
Select status emulation register bit 7	IEMUBI[5]	12-19
Select status emulation register bit 6	IEMUBI[4]	12-19
Enable emulation register 1 and 2 write IRQs	IEMUBI[3:2]	12-19
Enable emulation registers 1 and 2 read IRQs	IEMUBI[1:0]	12-19

Note:

** These timers are also enabled by UADR*

Table 5-6 Game Port and MIDI Port Functions

Function	Register and Bit Field	Reference
Enable MIDI loopback	UMCR[5]	page 12-1
Enable joystick	UJMPI[2]	page 12-11
Enable MIDI port	UJMPI[1]	page 12-11

Table 5-7 Audio I/O Functions

Function	Register and Bit Field	Reference
Enable mono and stereo microphone input	UMCR[2]	12-1
Enable line output	UMCR[1]	12-1
Enable line input	UMCR[0]	12-1

Initialization

Generally, the system control registers are set to their default values at reset. See “Initialization” on page 4-7 for a global discussion of initialization and reset.

When the InterWave IC is configured by its hardware implementation to operate in Plug and Play mode, the system software performs all PNP operations through registers in the system control module. The PNP configuration procedure detects each PNP ISA card installed in the system, identifies it, assigns it the necessary system resources (IRQ lines, DMA channels, etc.), and activates its logical devices. For a complete discussion of the PNP initialization sequence, see “Plug and Play Functions” on page 5-13.

Interrupt Level Selection

The system control module receives interrupt requests from other modules. It then uses mapping data in its own control registers to encode the requests onto the ISA bus.

Software clears interrupts by reading from or writing to the appropriate registers within the requesting module.

The discussion in this section assumes an understanding of the interrupt structure and the types of interrupts available (see “Interrupt Structure” on page 4-8). This section outlines the enabling, requesting, reporting, and mapping of all InterWave interrupts.

Registers for Enabling, Reporting, and Clearing Interrupts

Table 5-8 provides comprehensive data for all InterWave interrupt events. I/O reads and writes of the clearing registers are generally indexed. See Part , “InterWave Registers Reference” for details on indexing the registers.

Table 5-8 Registers for Interrupt Events

Group	Event Description	IRQ Enables	Reporting Mechanism	Clear Mechanism
SIRQ	Synth voice reaches end of volume ramp	SVCI[5] & URSTI[2]	UISR[6], SVCI[7], SVII[6]	IOR of all volume IRQs from SVII or a single IOW to SVII
SIRQ	Synth voice finishes loop	SACI[5] & URSTI[2]	UISR[5], SACI[7], SVII[7]	IOR of all loop IRQs from SVIR or a single IOW to SVIR
CIRQ	Codec record sample counter rolls past zero	CFIG1I[1], mode 2 or 3	CSR3I[5], CSR1R[0]	IOW to CSR1R or CSR3I[5]=0
CIRQ	Codec playback sample counter rolls past zero	CFIG1I[0]	CSR3I[4], CSR1R[0]	IOW to CSR1R or CSR3I[4]=0
CIRQ	Codec record FIFO reaches threshold	CFIG3I[7], mode 3	CSR3I[5], CSR1R[0]	IOW to CSR1R or CSR3I[5]=0
CIRQ	Codec playback FIFO reaches threshold	CFIG3I[6], mode 3	CSR3I[4], CSR1R[0]	IOW to CSR1R or CSR3I[4]=0
CIRQ	Codec timer reaches zero	CFIG2I[6]	CSR3I[6], CSR1R[0]	IOW to CSR1R or CSR3I[6]=0
	Extra IRQ: set enables	UDCI[7], UICI[6]	not reported	IOW to UDCI[7] = 0
SBIRQ	IOR of general port 1	URCR[6], URCR[3], IEMUBI[0]	USRR[4]	IOW to UCLRII
SBIRQ	IOW to general port 1	URCR[6], URCR[3], IEMUBI[2]	USRR[3]	IOW to UCLRII
SBIRQ	IOR of general port 2	URCR[6], URCR[4], IEMUBI[1]	USRR[6]	IOW to UCLRII
SBIRQ	IOW to general port 2	URCR[6], URCR[4], IEMUBI[3]	USRR[5]	IOW to UCLRII
SBIRQ	IOR of 2xE	URCR[7]	USRR[7]	IOW to UCLRII
SIRQ	TC (ISA bus) is reached	LDMACI[5]	UISR[7] & LDMACI[6]	IOR of LDMACI
SBIRQ	IOW to AdLib data register (UADR) **	UASBCI[1]	UISR[4] & UASRR[0]	IOW of UASBCI[1]=0
SBIRQ	IOW to SB U2X6R	UASBCI[5]	UASRR[3]	IOW of UASBCI[5]=0
SBIRQ	IOW to SB UI2XCR	UASBCI[5]	UASRR[4]	IOW of UASBCI[5]=0
SBIRQ	AdLib timer 1 rolls past FF	UASBCI[2]	UISR[2], UASRR[2]	IOW to UASBCI[2]=0
SBIRQ	AdLib timer 2 rolls past FF	UASBCI[3]	UISR[3], UASRR[1]	IOW to UASBCI[3]=0
MIRQ	MIDI transmit ready	GMCR[6:5]	UISR[0]	IOW to GMTDR
MIRQ	MIDI data received	GMCR[7]	UISR[1]	IOR of GMRDR
EXDIRQ	External-device interrupt	PRACTI[0]	none	none
SB_EMU_IRQ	Sound Blaster emulation (IEIRQI[0] = 1)	PSACTI[0]	IEIRQI[0]	IEIRQI[0] = 0
MPU401_IRQ	MPU-401 emulation (IEIRQI[1] = 1)	PMACTI[0]	IEIRQI[1]	IEIRQI[1] = 0

Note:

*** When in auto-timer mode and the UACWR has been written to a 04h, then the write to the UADR does not generate an interrupt.*

Interrupt-Mapping Equations

Equation 5-1 through Equation 5-5 show how the interrupt events in Table 5-8 map into the InterWave IRQ channels. For definitions of the bit fields and variables used in the interrupt equations, see Table 5-9.

Equation 5-1 Audio IRQ Channel 1

$$\text{Channel\_1\_IRQ} = \text{PUACTI}[0] \cdot \text{UMCR}[3] \cdot \text{IDECI}[6] \cdot$$
$$(((\overline{\text{IDECI}[7]} \cdot \text{CIRQ}) + \text{SIRQ}) \cdot \overline{\text{UMCR}[4]}$$
$$+ \text{SBIRQ} \cdot \overline{\text{UICI}[7]} + \text{MIRQ} \cdot \text{UICI}[6])$$

Equation 5-2 Audio IRQ Channel 2

$$\text{Channel\_2\_IRQ} = \text{PUACTI}[0] \cdot \text{UMCR}[3] \cdot \text{IDECI}[5] \cdot$$
$$(((\overline{\text{IDECI}[7]} \cdot \text{CIRQ}) + \text{SIRQ}) \cdot \text{UMCR}[4] + \text{IDECI}[7] \cdot \text{CIRQ}$$
$$+ \text{UDCI}[7] \cdot \text{UICI}[6] + \text{MIRQ} \cdot \overline{\text{UICI}[6]})$$

Equation 5-3 CD-ROM IRQ Channel

$$\text{CD\_ROM\_IRQ} = \text{EXDIRQ} \cdot \text{PRACTI}[0]$$

Equation 5-4 Sound Blaster Emulation IRQ Channel

$$\text{SB\_EMU\_IRQ} = \text{IEIRQI}[0] \cdot \text{PSACTI}[0];$$

Equation 5-5 MPU-401 IRQ Channel

$$\text{MPU401\_IRQ} = \text{IEIRQI}[1] \cdot \text{PMACTI}[0];$$

Equation 5-6 through Equation 5-9 show how the IRQ channel equations map to the IRQ pins, where x in IRQx specifies the IRQ number (2/9, 3, 4, 5, 7, 10, 11, 12, or 15). The notation "(UICI[2:0] == IRQx)" should read "UICI[2:0] specifies IRQx." IRQ10 and IRQ4 have slightly different equations since they are not supported by the Interleave Control register (UICI). In these equations, the term "(x ≠ 15)" specifies that the rest of the block, "(PUI1SI[3:0] ≠ 04h, or 0Ah)," is added only for IRQ15. That is, the equation for IRQ15 specifies that it is not enabled if UICI[2:0] points to IRQ15 and PUI1SI[3:0] selects either IRQ4 or IRQ10. This logic is implemented because IRQ4 and IRQ10 are legitimate selections for PUI1SI but they cause UICI[2:0] to point to IRQ15.

Equation 5-6 IRQ Selection

$$\text{IRQx} = (\text{Channel\_1\_IRQ}) \cdot (\text{UICI}[2:0] == \text{IRQx}) \cdot ((x \neq 15) + (\text{PUI1SI}[3:0] \neq 04\text{h, or } 0\text{Ah}))$$
$$+ (\text{Channel\_2\_IRQ}) \cdot (\text{UICI}[5:3] == \text{IRQx})$$
$$+ (\text{CD\_ROM\_IRQ} \cdot (\text{PRISI}[3:0] == \text{IRQx}))$$
$$+ (\text{SB\_EMU\_IRQ}) \cdot (\text{PSBISI}[3:0] == \text{IRQx})$$
$$+ (\text{MPU401\_IRQ}) \cdot (\text{PMISI}[3:0] == \text{IRQx}) ;$$

Equation 5-7 IRQ Enabling

$$\text{IRQx Enable} = \overline{\text{SUSPEND}} \bullet \text{PUACTI[0]} \bullet \text{UMCR[3]} \bullet$$
$$(\text{IDECI[6]} \bullet (\text{UICI[2:0]} == \text{IRQx}) \bullet ((x \neq 15) + (\text{PUI1SI[3:0]} \neq 04h, \text{ or } 0Ah))$$
$$+ \text{IDECI[5]} \bullet (\text{UICI[5:3]} == \text{IRQx}))$$
$$+ \overline{\text{SUSPEND}} \bullet \text{PRACTI[0]} \bullet (\text{PRISI[3:0]} == \text{IRQx})$$
$$+ \overline{\text{SUSPEND}} \bullet \text{PSACTI[0]} \bullet (\text{PSBISI[3:0]} == \text{IRQx})$$
$$+ \overline{\text{SUSPEND}} \bullet \text{PMACTI[0]} \bullet (\text{PMISI[3:0]} == \text{IRQx}) ;$$

Equation 5-8 IRQ10 and IRQ4 Selection

$$\text{IRQ[10,4]} = \text{Channel\_1\_IRQ} \bullet (\text{PUI1SI} == [0Ah, 04h])$$
$$+ \text{Channel\_2\_IRQ} \bullet (\text{PUI2SI} == [0Ah, 04h])$$
$$+ \text{CD\_ROM\_IRQ} \bullet (\text{PRISI[3:0]} == \text{IRQ}[0Ah, 04h])$$
$$+ \text{SB\_EMU\_IRQ} \bullet (\text{PSBISI[3:0]} == \text{IRQ}[0Ah, 04h])$$
$$+ \text{MPU401\_IRQ} \bullet (\text{PMISI[3:0]} == \text{IRQ}[0Ah, 04h]) ;$$

Equation 5-9 IRQ10 and IRQ4 Enabling

$$\text{IRQ[0Ah,04h] Enable} = \overline{\text{SUSPEND}} \bullet \text{PUACTI[0]} \bullet \text{UMCR[3]} \bullet$$
$$(\text{IDECI[6]} \bullet (\text{PUI1SI} == \text{IRQ}[0Ah, 04h])$$
$$+ \text{IDECI[5]} \bullet (\text{PU21SI} == \text{IRQ}[0Ah, 04h]))$$
$$+ \overline{\text{SUSPEND}} \bullet \text{PRACTI[0]} \bullet (\text{PRISI[3:0]} == \text{IRQ}[0Ah, 04h])$$
$$+ \overline{\text{SUSPEND}} \bullet \text{PSACTI[0]} \bullet (\text{PSBISI[3:0]} == \text{IRQ}[0Ah, 04h])$$
$$+ \overline{\text{SUSPEND}} \bullet \text{PMACTI[0]} \bullet (\text{PMISI[3:0]} == \text{IRQ}[0Ah, 04h]) ;$$

The NMI function is controlled as shown in Equation 5-10.

Equation 5-10 NMI Function

$$\overline{\text{IOCHK}} = / (\overline{\text{SUSPEND}} \bullet \text{PUACTI[0]} \bullet \text{UMCR[3]} \bullet \text{IDECI[4]} \bullet$$
$$((\text{UICI[2:0]} = 0) \bullet (((\overline{\text{IDECI[7]}} \bullet \text{CIRQ}) + \text{SIRQ}) \bullet \overline{\text{UMCR[4]}}$$
$$+ \text{SBIRQ} \bullet \overline{\text{UICI[7]}} + \text{MIRQ} \bullet \text{UICI[6]})$$
$$+ \text{SBIRQ} \bullet \text{UICI[7]}))$$

Table 5-9 describes the programmable bit fields and signals associated with Equation 5-1 through Equation 5-10.

Table 5-9 Bit Fields and Variables in IRQ Equations

Bit Field	Description
IDECI[7]	Send codec interrupts to interrupt channel 2 (and remove them from channel 1).
IDECI[6:4]	IRQ channel enables for channel 1 (bit 6), channel 2 (bit 5), and NMI (bit 4).
UMCR[4]	Send synth volume and loop interrupts to interrupt channel 2 (and remove them from channel 1).
UMCR[3]	Enables all IRQ and DRQ lines from the high-impedance state.
UICI[2:0]	Selects the IRQ number for interrupt channel 1.
UICI[5:3]	Selects the IRQ number for interrupt channel 2.
UICI[6]	Combines MIDI interrupts to interrupt channel 1 (and removes them from channel 2).
UICI[7]	Disables AdLib–Sound Blaster interrupts from channel 1 and generates NMIs instead.
UDCI[7]	Extra interrupt; used to force the channel 2 IRQ line active.
PUACTI[0]	AUDIO functions activate bit.
PRACTI[0]	External functions (e.g., CD-ROM) activate bit.
SUSPEND	Suspend in progress.
SIRQ	OR of all synthesizer interrupt events
CIRQ	OR of all codec interrupt events
SBIRQ	OR of all AdLib–Sound Blaster interrupt events
MIRQ	OR of all MIDI interrupt events
SB_EMU_IRQ	Sound Blaster emulation interrupt
MPU401_IRQ	MPU-401 emulation interrupt

DMA Channel Selection

The system control module processes and encodes DMA requests generated by other InterWave modules and passes them to the system bus. The topics in this section describe the types of DMA requests and their mapping to the DRQ lines. For more details on DMA transfers, see “DMA Data Transfers” on page 8-11.

Categories of DMA Requests

Table 5-10 lists the DMA requests by category.

Table 5-10 DMA Requests by Category

Category	Description
DRQMEM	DMA request for system memory to or from local memory transfers.
DRQPLY	DMA request for system memory to codec playback FIFO transfers.
DRQREC	DMA request for codec record FIFO to system memory transfers.
DRQEX	DMA request from the external function (e.g., CD-ROM) interface.

DRQ-Mapping Equations

Equation 5-11 shows how the DMA requests are combined into the three InterWave DRQ channel possibilities.

Equation 5-11 DRQ Channel Selection

$$Channel\_1\_DRQ = PUACTI[0] \bullet (DRQMEM + DRQREC + (UDCI[6] \bullet DRQPLY));$$

$$Channel\_2\_DRQ = PUACTI[0] \bullet \overline{UDCI[6]} \bullet DRQPLY;$$

$$External\_Device\_DRQ = PRACTI[0] \bullet DRQEX;$$

Equation 5-12 shows how the channels described in Equation 5-11 map to the DRQ pins, where x in DRQx specifies the DRQ number. The DRQ used for each DMA channel is selected with the DMA Channel Control register (UDCI). The notation "(UDCI[2:0]==DRQx)" should read "UDCI[2:0] specifies DRQx".

Equation 5-12 Mapping to the DRQ Pins

$$DRQx = (Channel\_1\_DRQ \bullet (UDCI[2:0] == DRQx))$$
$$+ (Channel\_2\_DRQ \bullet (UDCI[5:3] == DRQx))$$
$$+ (External\_Device\_DRQ \bullet (PRDSI[2:0] == DRQx));$$

Equation 5-13 shows the equations for the signals that enable the DRQ lines.

Equation 5-13 Enabling DMA Requests

$$DRQx\ Enable = \overline{SUSPEND} \bullet PUACTI[0] \bullet UMCR[3] \bullet ((UDCI[2:0] == DRQx)$$
$$+ (UDCI[5:3] == DRQx) \bullet (\overline{UDCI[6]}))$$
$$+ \overline{SUSPEND} \bullet PRACTI[0] \bullet (PRDSI[2:0] == DRQx) ;$$

DMA Data Width

For DMA accesses, the DMA request-acknowledge number determines the data width. DRQs 0, 1, and 3 use 8-bit transfer and DRQs 5, 6, and 7 use 16-bit transfer. Software can read the data width in the *DMA Width* bit of the LMC DMA Control register (LDMACI[2]).

DMA Transfer Rates

For DMA transfers between local and system memory, the *DMA Rate Divider* field of the LMC DMA Control register (LDMACI[4:3]) controls the rate of transfer.

System Bus Interface

The InterWave IC can connect directly to the ISA bus. Its bus attributes are

- Compliance with the Plug and Play ISA specification—includes an interface to a serial EEPROM, where the configuration information is to be stored
- Compliance with the ISA portion of the EISA bus specification
- 16-bit data bus
- 10-bit or 12-bit address bus. The InterWave IC-based sound card may decode the upper bits for 16-bit addresses.
- Support for two audio interrupts, one external device (e.g., CD-ROM) interrupt, one Sound Blaster emulation interrupt, and one MPU-401 emulation interrupt chosen from nine IRQ lines
- Support for using the IOCHK signal of the ISA bus to generate NMIs to the CPU
- Support for three DMA channels: codec play, codec record, and external (CD-ROM). The synthesizer may share the codec DMA channels for GUS compatibility.

Plug and Play Functions

This section describes the Plug and Play Industry Standard Architecture (PNP ISA) functions of the InterWave audio IC. It covers the following topics:

- Purpose of PNP
- Card mode versus system mode
- PNP auto-configuration—setting an InterWave IC-based PNP card for the system
- Programming in PNP card mode—PNP states, registers, and functions
- PNP resource requirements—system resources required by the InterWave IC

The Purpose of PNP

Plug and Play is a software and hardware mechanism that allows the optimum allocation of system resources without user interaction. In a system with only PNP-compliant cards attached to the ISA bus, fully automatic configuration of the PNP cards is possible. However, if the system also contains non-PNP ISA cards, user action may be necessary to assure coexistence of the two types of cards within the system.

To configure a PNP-compliant card, the PNP ISA software performs the following tasks for each PNP card installed:

- Isolates the card by assigning a unique handle, the Card Select Number (CSN)
- Reads from the card the system resources that it requires
- Assigns conflict-free system resources to be used by the card

Card Mode versus System Mode

The InterWave IC can operate in two PNP modes.

In *card mode*, an on-card serial EEPROM contains the resource requirements for the card. The system software accesses these requirements and configures the card according to the PNP protocol. See “InterWave Programming in PNP Card Mode” on page 5-14.

In *system mode*, the InterWave PNP registers are accessible without going through the PNP isolation process. There is no serial EEPROM, or the first byte of the serial EEPROM is A5h. This mode allows integration of the IC onto a system board. See “InterWave Programming in PNP System Mode” on page 5-20.

PNP Auto-Configuration Ports

The *PNP ISA Specification Version 1.0A* requires the implementation of three 8-bit ports used by the PNP system software to issue commands, check status, access resource data, and configure a PNP card. These ports are defined in Table 5-11.

Table 5-11 PNP Auto-Configuration Ports

PNP Port	InterWave Register	I/O Address	Type
ADDRESS	PIDXR—PNP Index Address	279h	write only
WRITE_DATA	PNPWRP—PNP Write Data Port	A79h	write only
READ_DATA	PNPRDP—PNP Read Data Port	Range: 0203h–03FFh	read only

The ADDRESS and WRITE_DATA ports have fixed addresses as shown in Table 5-11. Software can assign the READ_DATA register an address in the range 0203h–03FFh. The ADDRESS port acts as an index register, used to gain access to all PNP registers.

All PNP-compliant cards present in the system share these ports. The ports are used only during configuration procedures. One card is differentiated from another by a unique number called the Card Select Number (CSN).

All PNP registers are written to through the WRITE_DATA port. All PNP registers are read through the READ_DATA port. The contents of the ADDRESS register determines the destination or source of the data.

Set the I/O address of the READ_DATA port by writing the appropriate value to the PNP Set Read Data Port Address register (PSRPAI). To access PSRPAI, set PIDXR to 00h and write the address to PNPWRP (0A79h). The address of this register is set during the PNP isolation phase (discussed later in this section). See the **IwavePnpIsol** DDK function.

The PNP software selects the address of the PNP Read Data Port (PNPRDP) during the isolation phase. If the selected address is in conflict with another device, the software selects a new address. This selection process may be repeated several times. If there is no PNP card installed in the system, the isolation process fails.

IwavePnpIsol tries various I/O addresses before concluding that there are no PNP cards in the system.

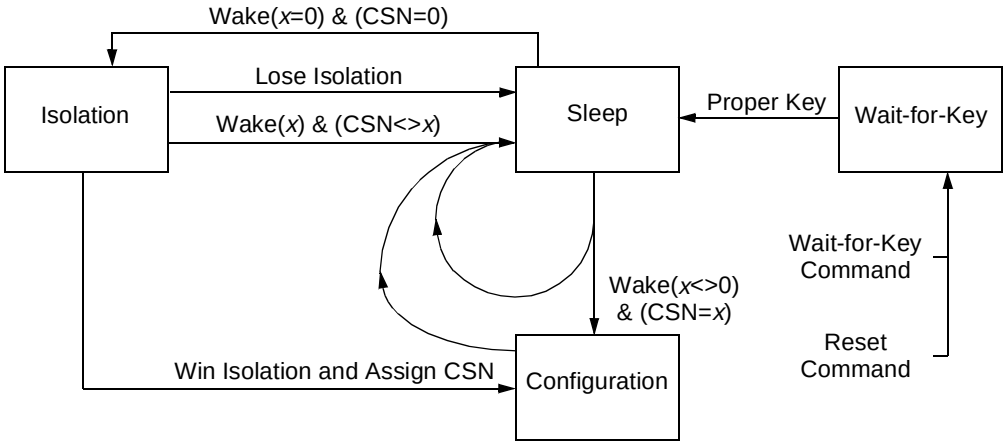
InterWave Programming in PNP Card Mode

The InterWave IC enters card mode whenever the output pin PNPCS is latched Low by the RESET signal at power-up. In this mode, the IC should be on a PNP-compliant card containing a serial EEPROM. This EEPROM contains resource allocation requirements for the logical devices contained on the card.

The Auto-Configuration Process

The PNP logic is dormant at power-up and must be enabled by the system software. Each PNP-compliant card attached to the PNP ISA bus must go through the series of auto-configuration phases (shown in Figure 5-2) to configure and activate the card.

Figure 5-2 PNP Auto-Configuration States



At power-up, each PNP card enters the *wait-for-key* state, in which its registers are not accessible. The system software must send to the card a sequence of 32 bytes known as the *initiation key*.

After all cards have been initiated, the system software places each card in the *isolation* state. Isolation is the process that identifies each card and assigns it a unique number, called the *card select number* (CSN), that is used later during the configuration phase.

Since all PNP cards respond to the same I/O ports described in Table 5-11, the system requires the CSN to distinguish between cards.

After the isolation phase is completed, the system software places each card in the *configuration* state. In this state, the card can respond to all configuration commands—the PNP system software can read the card's resource information and program the card's resource selections.

After configuration, the card's PNP functions enter the *sleep* state. From there, it can "wake up" in response to a need to change its configuration.

Auto-Configuration States

The following four subsections detail the functions of the PNP auto-configuration states introduced above.

Wait-for-Key State

At power-up or reset, the InterWave IC waits to receive a sequence of 32 bytes—the initiation key. The system software should write this sequence to register PIDXR (0279h). The exact sequence for the initiation key in hexadecimal notation is:

```
6A, B5, DA, ED, F6, FB, 7D, BE,
DF, 6F, 37, 1B, 0D, 86, C3, 61,
B0, 58, 2C, 16, 8B, 45, A2, D1,
E8, 74, 3A, 9D, CE, E7, 73, 39
```

Before writing the above sequence, the software should initialize the PNP initiation-key logic by writing 00h to register PIDXR two times. For details on generating the sequence, see the **IwavePnpKey** DDK function.

After the initiation key has been written, the PNP card enters the sleep state.

Isolation State

The isolation phase uses the InterWave registers listed in Table 5-12.

Table 5-12 Isolation-Phase Registers

Register	Description	I/O Address	Type
PISOCI	Serial isolation register.	PNPRDP,PIDXR=0x01	rd-wr
PCSNI	Card select number Reg.	PNPRDP,0x0A79	rd-wr
PWAKEI	PNP Wake[CSN] command	0x0A79, PIDXR=0x03	write

To start the isolation process for each card, the system software must:

Issue the initiation key to the card.

1. Send the Wake command to the card.
2. Set the PNP read data port address in PSRPAI.

The last step implemented in assembly language may look like this:

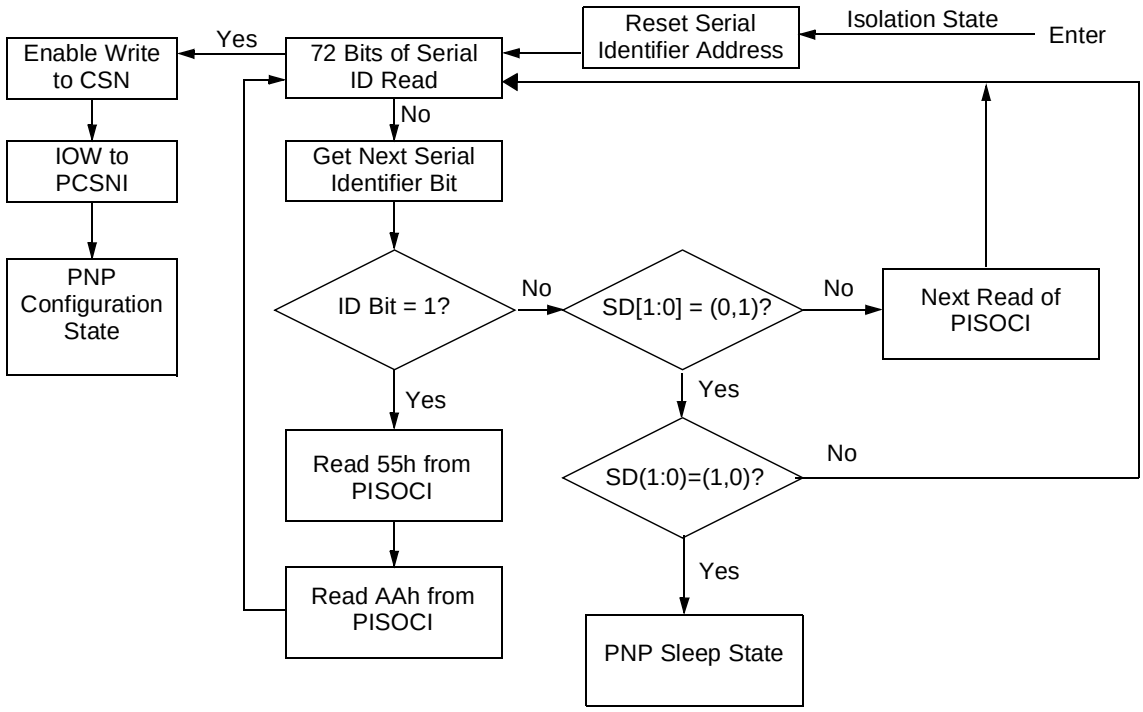
```
...
OUT    279h, 00h     ; set PIDXR to 00h to select PSRPAI
OUT   0A79h, 80h     ; set address in PSRPAI
...
```

These instructions set the address of the PNP Read Data Port register (PNPRDP) to 203h. The address of PNPRDP can be set only within the isolation state.

At power-up, each card has its card select number set to 0x00 by default. To issue the wake command, the system software writes 0x00 to the PNP Wake Command register (PWAKEI). This command wakes up all cards, but only one card gains the isolation state in the current iteration.

The isolation process depends on each PNP card having a unique 72-bit *serial identifier.* These bits are broken into two 32-bit fields and an 8-bit checksum. The PNP configuration software reads this identifier as part of the isolation process.

Figure 5-3 Reading the PNP Serial Identifier



To read the serial identifier, the system software generates 72 pairs of I/O read cycles to the PNP Isolate Command register (PISOCI). The software then checks the data received from each pair of reads. If the software reads 55h followed by AAh, it assumes that the current serial bit is a 1; otherwise, it assumes a 0. During the first 64 reads, the software generates a checksum using the received data. This checksum is compared with the last 8 bits of the sequence. Note that reading PISOCI is allowed only in the isolation state.

Note: If the first byte out of the serial EEPROM is the value A5h, then the IC is automatically forced into PNP system mode. At that point, the card select number can be assigned through the PNP Card Select Number Back Door register (PCSNBR) to place the IC in the PNP configuration state.

Two possible situations may arise during each iteration in the isolation process. It is possible to not ever receive the 0x55 and 0xAA data pair. It is also possible that the checksum may not match. If either case occurs, the software makes the assumption that the I/O address assigned to PNPRDP is in conflict and relocates it somewhere else. The software then repeats the process. Even though the entire range between 0x203 and 0x3FF is available to try, the software tries only a few locations before it assumes there are no PNP cards installed and exits.

If the current attempt to read a serial identifier is successful, then the software assigns the isolated card a unique card select number (CSN). The CSN must be a number in the range of 1 to 255 (FFh). In the next iteration, only PNP cards whose CSN are still 00h participate in the isolation process. The software continues this process until the last card is isolated.

See the **IwavePnpIsol** DDK function for an example of the isolation process.

Sleep State

From this state, a card can enter either the isolation state or the configuration state. If the PNP wake command is issued by writing 00h to the PWAKEI register, then all cards with a CSN = 00h (not yet isolated) enter the isolation state. If the wake command is issued by writing a non-zero value to PWAKEI, the card whose CSN matches this non-zero value enters the configuration state.

Configuration State

In this state, the system software can read the card's resource requirements and allocate them.

Card Control and Logical Device Configuration Registers

Programming the InterWave PNP functions requires a good working knowledge of the PNP registers. The following subsections introduce these registers in logical order and describe their functions in detail.

Table 5-13 summarizes the first registers encountered in the PNP configuration phase.

Table 5-13 PNP Card Control Registers

Register	Description	I/O Address	Type
PCCCI	PNP Configuration Control Command	A79h, PIDXR = 02h	write
PRESDI	PNP Resource Data	PNPRDP, PIDXR = 04h	read
PRESSI	PNP Resource Data Status	PNPRDP, PIDXR = 05h	read
PLDNI	PNP Logical Device Number	[PNPRDP, 0A79h]/ PIDXR = 07h	read-write

Reading the PNP Resource Data register (PRESDI) returns one byte of resource data from the InterWave IC while in the configuration state. These data come from the serial EEPROM. Software must read the PNP Resource Data Status register (PRESSI) to confirm that resource data are available before reading PRESDI. If PRESSI[0] = 1, then new resource data are available. Reading PRESDI resets this bit.

The PNP Logical Device Number register (PLDNI) selects an InterWave logical device. There are five such logical devices:

- Audio
- External device (typically a CD-ROM)
- Game port
- AdLib–Sound Blaster emulation
- MPU-401 emulation

The PNP Configuration Control Command register (PCCCI) has three independent commands associated with it, as described in Table 5-14.

Table 5-14 PCCCI Configuration Commands

Configuration Command	PCCCI Setting	Description
Reset command	01h	Resets all logical devices. Resets all configuration registers to their default values. Preserves PSRPAI, PCSNI, and the PNP state.
Wait-for-key command	02h	Forces all PNP cards to enter the wait-for-key state. Preserves PCNSI and does not change any logical device status.
Reset CSN	04h	All PNP cards not in wait-for-key states reset their CSN to 00h (PCSNI)

The system software should always wait 10 ms after issuing a reset command before accessing auto-configuration ports.

For information about the registers used to activate the logical devices, see “PUACTI, PRACTI, PGACTI, PSACTI, PMACTI—PNP Activate Registers” on page 12-24. For information about the registers used to check for possible I/O conflicts with the logical devices, see “PURCI, PRRCI, PGRCI, PSRCI, PMRCI—PNP I/O Range Check Registers” on page 12-24.

To determine whether an I/O range conflict exists, the system software should use the following procedure:

Generate I/O reads from all ports to be used by a logical device.

1. Verify that the data values 55h and AAh are received.
2. If the data values are incorrect, assume that a conflict exists and relocate the I/O range.

The software initiates this checking mechanism by enabling the I/O range check logic. It does this by writing 02h to the corresponding I/O range check register. With bit 0 of these registers set to 1, the logic should return the value 55h from the I/O ports. With bit 0 set to 0, the logic should return the value AAh from the ports. Otherwise, software should assume a conflict and relocate the I/O address. This operation must be conducted with the corresponding device in the inactive state.

Bit 0 of the activate registers indicates the state of the corresponding logical device. If this bit is set, the device is active; otherwise, the device is inactive.

See the **IwavePnpIOchk** DDK function for an example of I/O-conflict checking.

Resource Requirements

Internally, the InterWave local memory control module reads resource requirements information from the serial EEPROM. The system control module then makes the resource data available to the system software one byte at a time through the PNP Read Data Port register (PNPRDP).

The first byte of each data item contains information indicating the ID of the item and the number of bytes of data contained in the item. Furthermore, these data items are classified as small or large item structures. A 0 in the most significant bit of the first byte indicates a small data item structure (2 to 8 bytes), whereas a 1 indicates a large data item structure.

For a complete sample resource map, see Appendix B, "Sample Plug and Play Resource Map." For more information about PNP resource specification, see the *PNP ISA Specification, Version 1.0A* published by the Microsoft Corporation.

Reading Resource Data

Software can read resource data only from cards in the configuration state. During the isolation process all cards are placed in the sleep state. To place a card in the configuration state, the software must issue the wake command to that card. All other cards remain in the sleep state. Because the configuration state is attained from the sleep state, software must read the 9-byte serial identifier again before reaching the resource data, but it can do so one byte at a time rather than one bit at a time.

Software reads card resource data by polling the PNP Resource Status register (PRESSI) and waiting for PRESSI[0] to be set. When this bit is set, one byte of resource data is ready. Software must repeat this process for each new byte of data. Reading the data from the PNP Resource Data register (PRESDI) clears PRESSI[0]. When new data becomes available, the hardware sets PRESSI[0] again.

The resource data describes how many logical devices are on the card as well as the resource requirements for each logical device. From this information, software can program the logical device's configuration registers.

The **IwavePnpPeek** DDK function illustrates the reading of the card's resource data requirements.

InterWave Programming in PNP System Mode

If the InterWave IC powers up in PNP-compliant mode, the IC can still be forced into PNP-system mode. If, after the IC is placed in the PNP isolation state, the first byte that is read from the serial EEPROM is A5h, then the IC automatically goes into PNP-system mode and the PNP Card Select Number Back Door register (PCSNBR) becomes available to assign the CSN. Thus, configuration software can change whether the card is treated as PNP compliant or legacy (depending on the end user's system requirements) by writing to the first byte of the serial EEPROM. According to the PNP-ISA specification, A5h is an illegal value (not PNP compliant) because the most-significant bit (MSB) of the first byte should be zero. Therefore, there is no danger that this value would be accidentally used in a PNP-compliant situation.

Additional information on this topic is being developed.

Programming Tips and Examples

This section provides information about the following system control programming tasks:

- Configuring the PNP Card
- Isolating the PNP card
- Programming the serial EEPROM

Configuring the PNP Card

The InterWave DDK includes functions that allow access to the Plug and Play logic built into the IC. A programmer can use these low-level functions to write a program that isolates and configures any PNP device present in the system. The program in Sample 5-1 illustrates the use of several DDK functions in configuring the PNP card. For general information about using the DDK, see Chapter 18, "Programming With The Driver

Developer's Kit." For details about the PNP functions and other system control functions, see Chapter 20, "System Control DDK Functions."

Warning: This code should not be shipped to end-users. The iwinit.exe program is the only approved method for configuring the InterWave IC in DOS.

Sample 5-1 Configuring the PNP Card

```
//#############################################################################
//
// FILE: pnpinit.c
//
// PROFILE: Sample code to configure the part once it is in PNP mode. The user
// still has control over what resources get configured in the PNP interface via
// DOS variable IWCFG. IWCFG should be set defined as:
//   IWCFG=P2XR, PCODAR, PCDRAR, PATAAR, P401AR, PUI1SI, PUI2SR, PRISI, PMISI,
//         PSBISI, PUD1SI, PUD2SR, PRDSI
// If the variable is not defined in the environment then the part is initialized
// as if it were defined as follows:
//   IWCFG=220,32C,1F0,3F6,300,11,5,0,0,0,1,1,0
// This program is a good example of how to detect the InterWave hardware.
// Detection depends on the PNP ISA Specification-defined vendor ID. This vendor
// ID is assumed by the program to be 0x0496550A. IwavePnpPing takes this number
// with its bytes written backward as an argument.
//
//#############################################################################

#include <stdio.h>
#include "iwprotos.h"
#include "iwdefs.h"
#include "iwcore.h"

void main()
{
  WORD reg;
  BYTE csn_max=0, csn=0;

  IwaveInit();                          // get IWCFG environment variable
  IwavePnpKey();
  IwaveRegPoke(PCCCI, 0x05);            // software reset (CSN and all)
  IwaveDelay(10);                       // wait for reset to complete
  csn_max = IwavePnpIsol(&iw.pnprdp);   // Isolate card(s) if needed
  csn = IwavePnpPing(0x0A559604);       // get InterWave's CSN
  if (csn == FALSE) {
    printf("InterWave Card not found\n");
    exit(-1);
  }

  printf("Found %u PnP Card(s) in System\n", csn_max);
  printf("InterWave CSN       : %u\n", csn);
  printf("InterWave Vendor ID : %lx\n", iw.vendor);
  printf("PnP READ DATA PORT  : %x\n", iw.pnprdp);

  IwavePnpKey();
  IwavePnpWake(csn);                    // Select the InterWave Board
  IwavePnpSetCfg();                     // Configure PnP Interface Registers
  IwavePnpActivate(AUDIO, ON);
  IwaveGusReset();                      // place IC in GUS compatible mode
```

```
//#########################################################################
//  Configure Memory and enable IRQ/DMA interrupts
//#########################################################################

  IwaveMemCfg();
  _poke(iw.p2xr, 0x0b);
  if (_peek(iw.p2xr)!=0x0b)
  printf("Failure to write UMCR\n");

//#########################################################################
// The following instructions apply to the codec and they are
// designed to leave it in a functional state. Remember legacy
// software should be able to run.
//#########################################################################

  IwaveCodecIrq(CODEC_IRQ_ENABLE);  // ensure that codec can IRQ
  IwaveRegPoke(URSTI, 0x07);
  IwaveCodecMode(CODEC_MODE3);      // select mode 3
  IwaveRegPoke(CLOAI, 0);
  IwaveRegPoke(CROAI, 0);
  IwaveRegPoke(CLMICI, 0);
  IwaveRegPoke(CRMICI, 0);
  IwaveRegPoke(CLAX1I, 0x08);
  IwaveRegPoke(CRAX1I, 0x08);
  IwaveRegPoke(CFIG3I, 0x02);
  IwaveRegPoke(CLDACI, 0x00);
  IwaveRegPoke(CRDACI, 0x00);
  IwaveRegPoke(ICMPTI, 0x00);

//#########################################################################
// If both audio DMA channels are the same, combine them
//#########################################################################

  if(iw.dma1_chan == iw.dma2_chan) {
    reg = IwaveRegPeek(UDCI);
    IwaveRegPoke(UDCI, (reg & 0x07)|0x40);
  }

  if(iw.synth_irq == iw.midi_irq) {
    reg = IwaveRegPeek(UICI);
    IwaveRegPoke(UICI, (reg & 0x07)|0x40);
  }

  printf("InterWave Initialization. Version B0\n");
}
```

Isolating the PNP Card

The isolation process relies on the fact that each PNP-compliant device has a serial identifier consisting of a 72-bit unique, non-zero number. This number has two 32-bit fields and one 8-bit checksum.

On power up, each device automatically sets its card select number (CSN) to 00h and enters the wait-for-key state. Immediately after the device receives the initiation key sent by the **IwavePnpKey** function, it enters the sleep state. In the sleep state, the device listens

for a PNP wake command to enter isolation. After the device receives this command (through the PWAKEI register), the I/O address of the PNP Read Data Port (PNPRDP) must be set. Bits 1 and 0 of this address are always High, so only bits 9–2 need to be specified. Set the address by writing 00h to the PNP Index register (PIDXR) and then writing bits 9–2 of the address to PNPRDP. The **IwavePnpIsol** function performs these steps.

The **IwavePnpIsol** function checks to make sure the address for PNPRDP is conflict free. The function starts with location 203h. If it encounters a problem, it tries a different location. It tries various locations before assuming that no PNP cards are present in the system.

After securing a conflict free address for PNPRDP, **IwavePnpIsol** starts the isolation process. For every PNP device in the system, **IwavePnpIsol** reads the PNP Isolate Command (PISOCI) register 144 times. If no problems are encountered during the isolation process, the function writes a card select number to the PNP Card Select Number (PCSNI) register. The **IwavePnpIsol** function returns the last CSN assigned (greatest CSN).

Programming the Serial EEPROM

This section describes how to program the serial EEPROM on the InterWave IC-based board directly through the InterWave IC. The DDK provides the **IwavePokeEEPROM** and **IwavePeekEEPROM** drivers that allow an application to easily program the serial EEPROM. These drivers are written for the KM93C66 256x16 serial EEPROM and compatible units. For a sample program that writes the contents of a file containing a PNP resource map to the serial EEPROM, see Sample 5-2. For a sample resource map, see Appendix B, “Sample Plug and Play Resource Map.”

The InterWave IC provides the means to program an on-board PNP serial EEPROM through the PNP Serial EEPROM Control register (PSECI). Bits 3–0 of the PNP Serial EEPROM Control register (PSECI) control the EEPROM:

Serial EEPROM Data Out (PSECI[0])
: This bit corresponds to signal DO on the KM93C66; software reads data from the serial EEPROM through this bit.

Serial EEPROM Data In (PSECI[1])
: This bit corresponds to signal DI on the KM93C66; software writes data to the serial EEPROM through this bit.

Serial EEPROM Serial Clock (PSECI[2])
: This bit corresponds to signal SK on the KM93C66. This is the serial clock that drives the data into or out of the serial EEPROM. This signal should not be driven at a rate higher than 1 MHz.

Serial EEPROM Chip Select (PSECI[3])
: This bit corresponds to signal CS on the KM93C66. This is the chip select flag.

Note: To drive the serial EEPROM directly through PSECI, software must first set the* Serial EEPROM Mode *bit of the PNP Serial EEPROM Enable register (PSEENI[0]) and the IC must be in the PNP configuration state.

Sample 5-2 Code to Program the Serial EEPROM

```
/*////////////////////////////////////////////////////////////////////////////
/ FILE: eewrite.c
/
/ REMARKS: This program reads the contents of a file containing the data to be
/ stored in the serial EEPROM attached to the InterWave IC and programs the
/ EEPROM with it. The reads back the data from the EEPROM and verifies that it
/ was written correctly.
/
////////////////////////////////////////////////////////////////////////////*/

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "iwdefs.h"
#include "iwprotos.h"
#include "iwcore.h"

void main()
{
  FILE *fp;
  BYTE eeprom1[512];
  BYTE eeprom2[512];
  WORD i, j, k;
  char yorn;

  fp = fopen("plugplay.txt", "r");   /* file where resource map is stored */
  i = 0;
  while(fscanf(fp, "%x\n", &eeprom1[i++]) != EOF);
  for(j = i; j < 512; j++) eeprom1[i] = 0;

  IwaveInit();                       /* Initialize global var "iw" */
  printf("Warning! This program will write over the contents of the Serial\n");
  printf("         EEPROM located on the InterWave-based sound card.\n");
  printf("         Proceed? (y,n)\n");
  scanf("%1c", &yorn);

  if(yorn == 'y') {
    IwaveRegPoke(PLDNI,0x00);
    IwaveRegPoke(PUACTI,0x00);
    IwaveRegPoke(PSEENI,0x01);       /* Activate EEPROM control mode */
    PokePSECI(0x00);
    IwavePokeEEPROM(eeprom1);        /* Program serial EEPROM */
    IwavePeekEEPROM(eeprom2);        /* Read back data from serial EEPROM */

    for(i = 0; i < 512; i++) {       /* Verify write operation */
      if (eeprom1[i]!=eeprom2[i]) {
        printf("Failed. Expected %x, but  found %x\n", eeprom1[i], eeprom2[i]);
        break;
      }
    }
  }
}
```

CHAPTER

6 CODEC/MIXER

The InterWave codec module provides a complete audio subsystem for PC multimedia integration. The codec module includes the codec 16-sample record and playback FIFOs, the record analog-to-digital converter (ADC), the playback digital-to-analog converter (DAC), the synthesizer DAC, a stereo mixer, attenuation and gain control for a variety of mixer inputs and outputs, record and playback sample counters, and a timer.

This chapter covers the following topics:

- Codec basics—briefly describes the components of the codec module.
- Data paths—illustrates the flow of data through the codec components.
- Register overview—lists of codec functions available through the programmable registers
- Initialization—discusses what the programmer needs to know about the codec module when it is powered up or reset.
- Interrupt structure—discusses the available codec interrupts and shows the equations that set and clear them.
- Operating modes—describes the three operating modes of the codec module.
- Data conversion—the possible data formats, the available sampling rates, and the variable frequency playback mode.
- FIFOs—describes the FIFOs and tells how to get sample information into and out of them.
- Mixer—describes the analog processing functions of the codec module.
- Serial interface—describes the serial interface used to send data to and receive data from an external digital signal processor (DSP).
- Miscellaneous functions—describes the codec module timer and the general purpose output pins.
- Programming tips and examples

Codec Basics

The InterWave codec module contains the following components:

Standard 16-sample FIFOs:

The codec module connects to the system bus interface (SBI) through two FIFOs, one each for record and playback. These FIFOs have programmable thresholds and can generate interrupts when the selected threshold is reached.

Record and playback sample counters:

Two 16-bit sample counters, one for playback and one for record, can be used either for DMA transfers to the SBI or for transfers to the local memory record and play FIFOs (LMRF and LMPF: see Chapter 8, “Local Memory

Control"). The record sample counter can also be used to count the number of samples being passed into the LMRF from the synthesizer DSP.

Record ADC and playback DAC:
The ADC and DAC provide the processing necessary to get information from the playback FIFO to the mixer and from the mixer to the record FIFO. These converters are the coder and decoder for which the codec module is named.

Synthesizer DAC:
The stereo digital output of the InterWave synthesizer module is converted to analog by the codec module.

Stereo mixer: The mixer provides analog input and output, signal routing, and mixing for audio signals. It also provides attenuation and gain control and muting for a variety of mixer inputs and outputs.

Serial DSP interface:
The serial interface allows an external general purpose digital signal processor (DSP) to utilize the codec, mixer, and analog I/O resources of the InterWave IC.

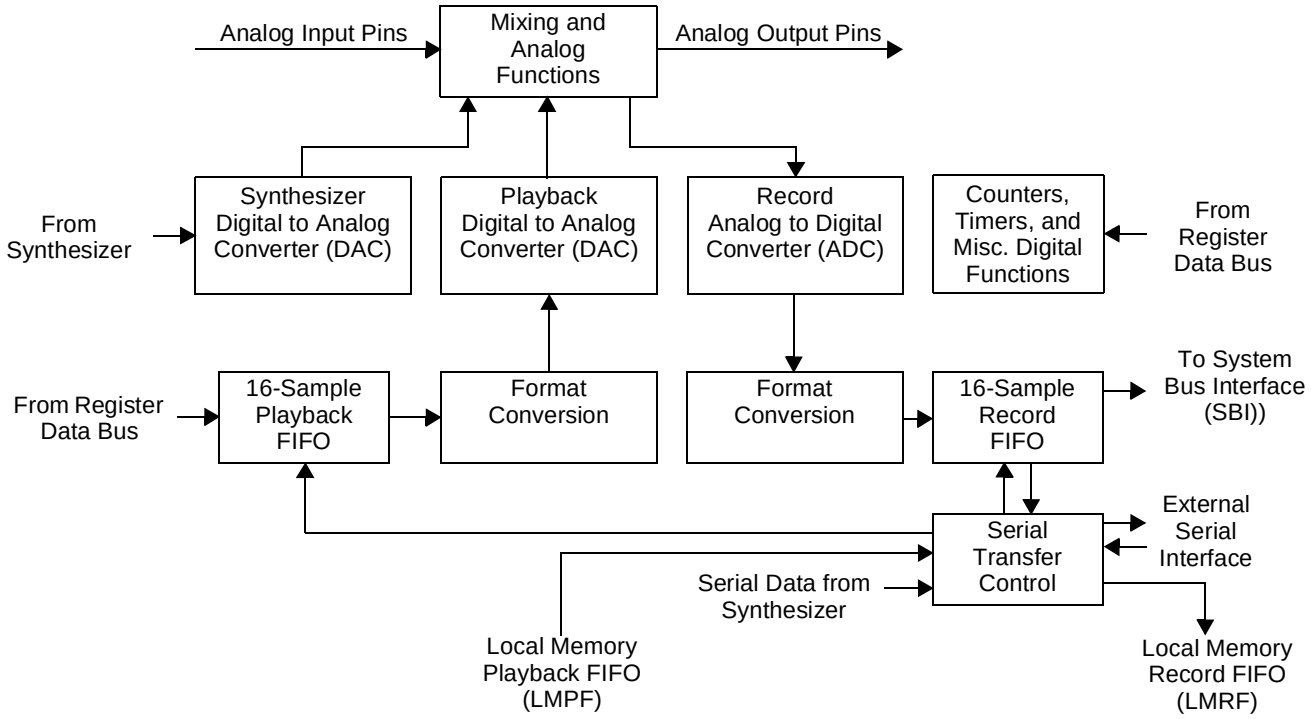
Timer: The programmable 16-bit timer with 10-µs resolution provides a timed interrupt signal.

External control outputs:
Two of the IRQ pins (IRQ4, IRQ10) can be programmed to perform as general purpose, digital control output pins.

Codec Data Paths

Figure 6-1 illustrates the flow of data and signals through the components of the codec module.

Figure 6-1 Codec Data Paths



Register Overview

The following tables outline the codec and mixer functions that can be accomplished through the InterWave programmable registers. For detailed information about the registers, see the reference page listed in the last column of each table. These references point to the applicable section in Chapter 13, "Codec/Mixer Registers."

Table 6-1 through Table 6-3 group the codec and mixer functions into the following areas:

- General control and configuration functions
- Input and output control functions
- DMA and IRQ functions

Table 6-1 Codec General Control and Configuration Functions

Function	Register and Bit Field	Reference
Read codec initialization status	CIDXR[7]	13-1
Protect the CPDFI, CRDFI, and CFIG1I registers	CIDXR[6]	13-1
Set the indirect I/O address pointer (index)	CIDXR[4:0]	13-1
Read and write to codec registers	CDATAP	13-2
Determine whether upper or lower byte of 16-bit sample is ready in record FIFO	CSR1R[7]	13-2
Determine whether left or right sample is ready in record FIFO	CSR1R[6]	13-2
Determine if data is ready to be read from the record FIFO	CSR1R[5]	13-2
Determine if sample has been lost due to overrun or underrun	CSR1R[4]	13-2
Determine whether to write upper or lower byte of 16-bit sample to playback FIFO	CSR1R[3]	13-2

Table 6-1 Codec General Control and Configuration Functions (Continued)

Function	Register and Bit Field	Reference
Determine whether to write left or right sample to playback FIFO	CSR1R[2]	13-2
Determine if data can be written to the playback FIFO	CSR1R[1]	13-2
Write data to the playback FIFO	CPDR	13-4
Read data from the record FIFO	CRDR	13-4
Select playback data format	CPDFI[7:5]	13-6
Select stereo or mono playback	CPDFI[4]	13-6
Select playback clock divider	CPDFI[3:1]	13-6
Select playback crystal	CPDFI[0]	13-6
Enable the record codec path	CFIG1I[1]	13-7
Enable the playback codec path	CFIG1I[0]	13-7
Set or read the state of the GPOUT flags	CEXTI[7:6]	13-8
Determine if record FIFO is full	CSR2I[7]	13-9
Determine if playback FIFO is empty	CSR2I[6]	13-9
Read the revision ID number of the IC	CMODEI[7,3:0]	13-10
Select codec operation mode	CMODEI[6:5]	13-10
Set the playback sample counter load value	CUPCTI, CLPCTI	13-11
Enable codec timer and timer interrupt	CFIG2I[6]	13-11
Disable record sample counter	CFIG2I[5]	13-11
Disable playback sample counter	CFIG2I[4]	13-11
Select variable frequency playback mode	CFIG3I[2]	13-12
Set the codec timer load value	CUTIMI, CLTIMI	13-14
Determine if attempted read from empty record FIFO (underrun)	CSR3I[3]	13-15
Determine if record FIFO is full (overrun)	CSR3I[2]	13-15
Determine if attempted write to full playback FIFO (overrun)	CSR3I[1]	13-15
Determine if playback FIFO is empty (underrun)	CSR3I[0]	13-15
Select record data format	CRDFI[7:5]	13-17
Select stereo or mono record	CRDFI[4]	13-17
Select record clock divider	CRDFI[3:1]	13-17
Select record crystal	CRDFI[0]	13-17
Set variable playback frequency	CPVFI	13-18
Set the record sample counter load value	CURCTI, CLRCTI	13-19

Table 6-2 Codec Input and Output Control Functions

Function	Register and Bit Field	Reference
Determine which input source is fed to the left/right ADC	CLICI[7:6], CRICI[7:6]	13-4
Set gain for input to left/right ADC	CLICI[3:0], CRICI[3:0]	13-4
Mute the left/right auxiliary 1/synthesizer input	CLAX1I[7], CRAX1I[7]	13-5
Set gain for left/right auxiliary 1/synthesizer input	CLAX1I[4:0], CRAX1I[4:0]	13-5
Mute the left/right auxiliary 2 input	CLAX2I[7], CRAX2I[7]	13-5
Set gain for left/right auxiliary 2 input	CLAX2I[4:0], CRAX2I[4:0]	13-5
Mute the left/right DAC output and ADC loopback	CLDACI[7], CRDACI[7]	13-6
Set attenuation for left/right DAC output and ADC loopback	CLDACI[5:0], CRDACI[5:0]	13-6
Determine if input to left/right ADC is causing clipping	CSR2I[1:0], CSR2I[3:2]	13-9
Specify loopback attenuation	CLCI[7:2]	13-10
Enable loopback	CLCI[0]	13-10
Select the output full-scale voltage	CFIG2I[7]	13-11
Force the output of DACs to center of scale during playback underrun	CFIG2I[0]	13-11
Select auxiliary 1 or synthesizer input	CFIG3I[1]	13-12
Mute the left/right line input	CLLICI[7], CRLICI[7]	13-13
Set gain for left/right line input	CLLICI[4:0], CRLICI[4:0]	13-13
Mute the left/right microphone input	CLMICI[7], CRMICI[7]	13-14
Set gain for left/right microphone input	CLMICI[4:0], CRMICI[4:0]	13-14
Mute the left/right line output	CLOAI[7[, CROAI[7]	13-16
Set gain for left/right line output	CLOAI[4:0], CROAI[4:0]	13-16
Mute the left/right mono input	CMONOI[7]	13-17
Mute the left/right mono output	CMONOI[6]	13-17
Place AREF pin in high-impedance mode	CMONOI[5]	13-17
Set attenuation for left/right mono input	CMONOI[3:0]	13-17

Table 6-3 Codec DMA and IRQ Functions

Function	Register and Bit Field	Reference
Disable DMA on sample counter interrupt	CIDXR[5]	13-1
Read global interrupt status	CSR1R[0]	13-2
Select I/O or DMA for record FIFO	CFIG1I[7]	13-7
Select I/O or DMA for playback FIFO	CFIG1I[6]	13-7
Select 1 or 2 channel DMA operation	CFIG1I[2]	13-7
Enable codec interrupts	CEXTI[1]	13-8
Determine if a record or playback DMA request is active	CSR2I[4]	13-9
Enable record FIFO service request IRQ	CFIG3I[7]	13-12
Enable playback FIFO service request IRQ	CFIG3I[6]	13-12
Select the FIFO threshold	CFIG3I[5:4]	13-12
Read status of codec timer IRQ	CSR3I[6]	13-15
Read status of record FIFO IRQ	CSR3I[5]	13-15
Read status of playback FIFO IRQ	CSR3I[4]	13-15

Initialization

Three bits from the PNP Power Mode register (PPWRI[2:0]) must be set High before certain paths in the codec module can be used. These bits, described below, enable or disable the record path, the playback path, or the analog circuitry.

PPWRI[1], Codec Record Path Enable
Enables (High) or disables (Low) the record ADC

PPWRI[2], Codec Playback Path Enable
Enables (High) or disables (Low) the playback DAC

PPWRI[0], Codec Analog Circuitry Enable
Setting this bit Low puts the codec analog circuitry into a low-power state and deactivates all the analog pins. The codec outputs, LINEOUT[L,R] and MONOOUT, stay at their nominal voltage.

To set these bits, write 80h plus the hex value for the bit to the PPWRI register. For example, to set PPWRI[1], write 82h to the register. To set all three bits, write 87h. These bits are High on power-up and reset. When the InterWave IC is in suspend mode, the codec behaves as if all three of these bits are Low.

Codec Interrupt Structure

The codec module can generate the following interrupts:

- Playback and record FIFO I/O threshold reached
- Playback and record sample counters receive an additional sample after the counter has decremented all the way to zero
- Codec timer decrements to zero

The codec module combines the result of the module's interrupt logic into one interrupt signal and passes it to the interrupt selection logic of the system control module. Software enables the codec interrupts onto the ISA bus by setting the *Global Interrupt Enable* bit of the External Control register (CEXTI[1]) to High. Setting CEXTI[1] to Low disables the codec interrupts. When any one or more of the codec interrupt events occur, the *Global Interrupt Status* bit of the Codec Status Register 1 (CSR1R[0]) goes High.

For information about using the DDK to handle codec interrupts, see “Handling Codec Interrupts” on page 6-20.

Equation 6-1 through Equation 6-8 show the actions required to set (CSET) and clear (CCLR) all the latches associated with codec interrupts. There is one latch to drive each of the three interrupt status bits in the Codec Status Register 2 (CSR2I). For a list and definitions of the variables used in these equations, see Table 6-4.

Equation 6-1 Playback FIFO Interrupt Set

$$
\begin{aligned}
&\text{CSET\_CSR3I[4]} = \\
&(\,(\,(\,\text{MODE} == 1\,) + (\,\text{MODE} == 2\,)\,) \bullet (\,\text{Playback sample counter rollover}\,) \\
&+ (\,\text{MODE} == 3\,) \bullet \text{CFIG3I[6]} \bullet \overline{\text{CFIG1I}}\text{[6]} \bullet \text{CFIG1I[0]} \bullet (\,\text{Playback sample counter rollover}\,) \\
&+ (\,\text{MODE} == 3\,) \bullet \text{CFIG3I[6]} \bullet \text{CFIG1I[6]} \bullet \text{CFIG1I[0]} \bullet (\,\text{Playback FIFO threshold reached}\,)\,);
\end{aligned}
$$

Equation 6-2 Playback FIFO Interrupt Clear

CCLR_CSR3I[4] =

((IOW to CDATAP) • ($\overline{\text{RDB}}$[4] • (CIDXR[4:0] == 18h)) + (IOW to CSR1R) ;

Equation 6-3 Record FIFO Interrupt Set

CSET_CSR3I[5] =

((MODE == 2) • (Record sample counter rollover)

+ (MODE == 3) • CFIG3I[7] • $\overline{\text{CFIG1I}}$[7] • CFIG1I[1] • (Record sample counter rollover)

+ (MODE == 3) • CFIG3I[7] • CFIG1I[7] • CFIG1I[1] • (Record FIFO threshold reached)) ;

Equation 6-4 Record FIFO Interrupt Clear

CCLR_CSR3I[5] =

((IOW to CDATAP) • ($\overline{\text{RDB}}$[5] • (CIDXR[4:0] == 18h)) + (IOW to CSR1R) ;

Equation 6-5 Codec Timer Interrupt Set

CSET_CSR3I[6] =

(((MODE == 2) + (MODE == 3)) • (Timer reaches zero)) ;

Equation 6-6 Codec Timer Interrupt Clear

CCLR_CSR3I[6] =

((IOW to CDATAP) • ($\overline{\text{RDB}}$[6] • (CIDXR[4:0] == 18h)) + (IOW to CSR1R) ;

Equation 6-7 Codec Global Interrupt Status

CSR1R[0] =

(CSR3I[4] + CSR3I[5] + CSR3I[6]) • (MODE == 2 + MODE == 3)

+ (CSR3I[4] • (MODE == 1)) ;

Equation 6-8 Codec Interrupt Signal

CIRQ = CSR1R[0] • CEXTI[1] ;

Table 6-4 Codec Interrupt Equation Variables

Variable	Definition
CSR3I[6, 5, 4]	The timer, record path, and playback path interrupt status bits of the Codec Status Register 3
CFIG3I[7:6]	The record and playback path mode 3 interrupt enables
CFIG1I[7:6]	The record and playback path DMA-I/O cycle selection bits
CDATAP	The codec indexed register data port
CIDXR[4:0] == 18h	The codec indexed register index field is set to the Codec Status Register 3
RDB[X]	The register data bus, where X is 6, 5, or 4. These bits on the data bus are set Low by setting Low the corresponding bit in the Codec Status Register 3.
CSR1R	The Codec Status Register 1
CEXTI[1]	The global codec interrupt enable

Operating Modes

The codec module has three operating modes. Modes 1 and 2 provide the functionality of the popular CS4231 codec device. Mode 3 is an AMD-defined mode that provides the following new features:

- Independent record and playback rates
- Variable frequency playback mode, with 256 programmable sampling rates
- I/O cycle interrupts
- Record and playback FIFO thresholds
- Volume controls for the line outputs and mute control for the mono output

Select the codec operating mode with the *Mode Select* field of the Mode Select, ID register (CMODEI[6:5]). The values for each mode are:

mode 1 bits 6–5 = 00 (write 00h to CMODEI)

mode 2 bits 6–5 = 10 (write 40h to CMODEI)

mode 3 bits 6–5 = 11 (write 6Ch to CMODEI)

The details of operation for each mode are covered in the relevant sections of this chapter and in the register reference pages of Chapter 13, “Codec/Mixer Registers.”

Data Conversion

The codec module provides analog-to-digital converters (ADCs) to feed the record FIFO and digital-to-analog converters (DACs) from the playback FIFO. These functions operate independently; any combination of ADC data format and sampling rate works with any independent combination of DAC data format and sampling rate. The format-conversion function lies between the FIFOs and the data conversion function. The data is 16-bit signed as it enters the DACs and exits the ADCs.

Data Format

The Playback Data Format register (CPDFI) and the Record Data Format register (CRDFI) select the digital data format into the ADCs and out of the DACs. The register to be used depends on the codec operation mode.

Available Data Formats

The InterWave IC supports both mono and stereo in the following formats:

- 8-bit unsigned linear
- 8-bit µ-law
- 8-bit A-law
- 16-bit signed little endian
- 16-bit signed big endian
- 4-bit 4:1 IMA ADPCM

For information about the conversion formulas used for µ-law, A-law, and ADPCM conversion, see the *Proposal for Standardized Audio Interchange Formats, Version 2.12,* published by the Interactive Multimedia Association.

Software must load the sample counters with the correct value, depending on the type of data selected. For information about calculating the sample counter load value, see "Selecting Data Format and Sampling Rate" on page 6-25.

Mode 1 Format Control

In mode 1 operation, the Playback Data Format register (CPDFI) controls the data format for both playback and record. Therefore, both playback and record must be in the same format.

Mode 2 and 3 Format Control

In modes 2 and 3, the Playback Data Format register (CPDFI) controls the data format for playback and the Record Data Format register (CRDFI) for record.

Mono Mode

When the playback path is in mono mode (CPDFI[4] set Low), the codec drives both the left and right DACs with the same data. When the record path is in mono mode (CRDFI[4] set Low), the codec passes only the state of the left ADC onto the codec FIFO.

Sampling Rates

In mode 3 operation, software can specify the sampling rate independently for playback and record. For playback, two ranges of sampling rates are available: standard or variable frequency. For information about setting the data format and sampling rate using the DDK, see "Selecting Data Format and Sampling Rate" on page 6-25.

Standard Mode—Playback and Record

In standard mode, the DACs and ADCs can each operate at one of 14 different sampling rates ranging from 5.5 kHz to 48 kHz. In modes 1 and 2, set the playback and record sampling rate in the Playback Data Format (CPDFI) register. In mode 3, set the playback sampling rate in CPDFI and the record sampling rate in the Record Data Format (CRDFI) register. For more information about selecting the sampling rate, see "CPDFI—Playback Data Format" on page 13-6 and "CRDFI—Record Data Format" on page 13-17.

Variable Frequency Playback Mode

In variable frequency playback mode—enabled by setting the *Variable Frequency Playback Mode* bit of the Configuration Register 3 (CFIG3I[2]) High—the playback frequency can be varied continuously. Two ranges are possible, depending on which crystal has been selected in the *Crystal Select* bits of the Playback Data Format (CPDFI[0]) or Record Data Format (CRDFI[0]) registers. Each range has 256 steps, determined by the value stored in the Playback Variable Frequency register (CPVFI). Table 6-5 lists the available ranges.

Table 6-5 Variable Frequency Formula and Ranges

Oscillator	Formula for Frequency	Range	CPDFI[0]
16.9344 MHz	16,934,400/(16 • (48+CPVFI))	3.5 kHz to 22.05 kHz	1
24.576 MHz	24,576,000/(16 • (48+CPVFI))	5.0 kHz to 32.00 kHz	0

Synthesizer DAC

The codec module converts the stereo digital output of the synthesizer module to analog. It multiplexes the outputs of the left and right synthesizer DACs with the AUX1[L,R] inputs before passing them, through attenuators, into the main mixer. The InterWave IC presents the digital data to the synthesizer DACs as stereo, 16-bit signed, 44.1-kHz samples.

Codec FIFOs

The codec module connects to the system bus interface (SBI) through two 16-sample record and playback FIFOs. As an enhancement to the CS4231, the FIFOs have programmable thresholds (full, half full, and empty) and can generate interrupts in response to threshold-reached events.

These FIFOs can be accessed through either DMA cycles or programmed I/O cycles. To remove data from the record FIFO, read the Record Data register (CRDR). To put data in the playback FIFO, write to the Playback Data register (CPDR). These direct registers share the same I/O address (PCODAR + 3).

In addition to the codec FIFOs, the InterWave IC allows local DRAM to be configured as very large playback and record FIFO buffers. These can be as large as 256 Kbytes each. See Chapter 8, “Local Memory Control” for information about using local memory as FIFO buffers.

Data Order

Table 6-6 shows how data is ordered into and out of the FIFOs from the CPU's perspective. In this table, S stands for sample, which is followed by a number that implies the order; L stands for left channel; and R stands for right channel.

Table 6-6 FIFO Data Ordering

Sample Data Format	Order (first byte, second byte,...)
4-bit ADPCM mono	(S2 in bits 7–4; S1 in bits 3–0), (S4 in bits 7–4; S3 in bits 3–0), . . .
4-bit ADPCM stereo	(S1R in bits 7–4; S1L in bits 3–0), (S2R in bits 7–4; S2L in bits 3–0), . . .
8-bit mono (linear, µ-law, A-law)	S1, S2, S3, . . .
8-bit stereo (linear, µ-law, A-law)	S1L, S1R, S2L, ...
16-bit mono little endian	S1 [7:0], S1[15:8], S2 [7:0], ...
16-bit mono big endian	S1[15:8], S1[7:0], S2[15:8], ...
16-bit stereo little endian	S1L[7:0], S1L[15:8], S1R[7:0], S1R[15:8], S2L[7:0], ...
16-bit stereo big endian	S1L[15:8], S1L[7:0], S1R[15:8], S1R[7:0], S2L[15:8], ...

FIFO Thresholds

In mode 3 operation, software can set the FIFO thresholds at which DMA or interrupt requests become active to one of three configurations. Specify the threshold configuration with the *FIFO Threshold Select* field of Configuration Register 3 (CFIG3I[5:4]). Table 6-7 shows the possible configurations.

Table 6-7 FIFO Threshold Configurations

CFIG3I[5:4]	Name	Record FIFO State	Playback FIFO State
0 0	Minimum	Not empty	Not full
0 1	Middle	Half full	Half empty
1 0	Maximum	Full	Empty
1 1	Reserved		

DMA Transfers

The InterWave IC generates separate DMA request signals for the record and playback FIFOs. In systems that can spare only a single DMA channel, the IC provides a mode that allows the playback DMA request pin to function as either the record or playback DMA request pin. In this mode, simultaneous record and playback operation is prohibited; only playback or record, but not both, will function.

Use Configuration Register 1 (CFIG1I) to select DMA operation for the record and playback FIFOs and 1-channel or 2-channel DMA operation. Setting the *Record FIFO I/O Select* bit (CFIG1I[7]) and the *Playback FIFO I/O Select* bit (CFIG1I[6]) both Low selects DMA operation for record and playback respectively. Setting the *1 or 2 channel DMA Operation Select* bit (CFIG1I[2]) High selects single-channel operation and setting it Low selects 2-channel. These bits are protected; to write to these bits, the *Mode Change Enable* bit of the Codec Index Address register (CIDXR[6]) must be set High.

If the record or playback paths are disabled—by setting the *Record Enable* (CFIG1I[1]) or *Playback Enable* (CFIG1I[0]) bits Low—after the associated DMA request signal has

become active, the circuitry continues the sample transfer (waiting for the acknowledge) as if the path were still enabled. After this final transfer, no other DMA requests occur.

Normally, DMA requests for the playback FIFO occur one after another until that FIFO is full, and DMA requests for the record FIFO occur one after another until that FIFO is empty. However, in mode 3 operation, the FIFO thresholds can be set to minimum, middle, and maximum using CFIG3I as listed in Table 6-7; the threshold configuration specifies the point at which the DMA request becomes active. Table 6-8 shows the number of samples transferred per DMA request-acknowledge cycle. The “Samples per DRQ” column is valid in modes 1 and 2 or in mode 3 when the threshold is set to minimum. In mode 3, when the threshold is middle or maximum, the IC transfers more data; after DMA acknowledge becomes active, samples are transferred until the playback FIFO is full or the record FIFO is empty (depending on which FIFO is being serviced).

Note: Middle and maximum mode can cause the chip to burst DMA data for more time than the ISA bus was designed to allow. In some ISA systems, use of these modes cause ISA-bus refresh cycles to be skipped. Some ISA-bus functions, such as video controllers and additional-system-memory cards, rely on these refresh cycles to retain DRAM integrity.

Table 6-8 Samples and Cycles per DMA Request

	8-bit DMA		16-bit DMA	
Sample Data Format	**Samples per DRQ**	**Cycles per DRQ**	**Samples per DRQ**	**Cycles per DRQ**
4-bit ADPCM mono	8	4	8	2
4-bit ADPCM stereo	4	4	4	2
8-bit mono (linear, µ-law, A-law)	1	1	2	1
8-bit stereo (linear, µ-law, A-law)	1	2	1	1
16-bit mono	1	2	1	1
16-bit stereo	1	4	1	2

When the Record FIFO Is Disabled

If software disables the record FIFO by setting CFIG1I[1] Low (or under the special circumstance of enabling both the record and playback paths for DMA but selecting single-channel DMA operation—CFIG1I[2:0] = 1, 1, 1), the codec clears all data still in the FIFO so that when it is reactivated, no old data is available. The codec subsequently allows four sample periods to clear the record data path prior to the FIFO (format translation and filtering); then it disables the record path to minimize power consumption.

When the Playback FIFO Is Disabled

If software disables the playback FIFO by setting CFIG1I[0] Low, the following events occur:

The codec mutes the playback audio immediately.

1. The playback FIFO is immediately cleared (emptied).
2. Four sample periods later the codec disables the playback path to minimize power consumption.

I/O Transfers

Software can move samples through the FIFOs using I/O cycles instead of DMA. In mode 1 or mode 2 operation, the IC generates interrupts (if not masked) when the sample counter decrements to zero. In mode 3 operation, the IC generates interrupts (if not masked) when the FIFOs reach their selected threshold levels. The codec generates the interrupts at the same point that DMA requests would have gone active had DMA operation been selected.

Status bits are provided for:

- Record FIFO data available
- Playback FIFO buffer available
- Record FIFO overrun and underrun
- Playback FIFO overrun and underrun

Select I/O operation for the record and playback FIFOs by setting the *Record FIFO I/O Select* (CFIG1I[7]) and *Playback FIFO I/O Select* (CFIG1I[6]) bits High. These bits are protected; to write to these bits, set the *Mode Change Enable* bit of the Codec Index Address register (CIDXR[6]) High.

ADPCM Issues

When the record or playback paths are in ADPCM mode, there are special requirements. Each path has an index variable that is used during ADPCM translation. This index is initialized at the beginning of the translation and changes to a different state after every sample passes through the compressor or expander. For proper operation, system software must synchronize initialization of the index with the hardware. For example, if an ADPCM waveform is being played through the expander and the user pauses the playback in the middle of the wave file, then the current state of the index must be preserved in case the user wishes to resume play. However, if playback of the file completes, then the ADPCM index can be initialized for playback of the next wave file.

For the record path, when the Record Enable bit of Configuration Register 1 (CFIG1I[1]) is activated (set High), the index is calculated with every sample (assuming the record path is in ADPCM mode) after starting from its initialized state. Then, when CFIG1I[1] is deactivated (set Low), the index is placed back into its initialized state. It is also possible to stop the record FIFO with the *ADPCM Record Suspend* bit of Configuration Register 3 (CFIG3I[3]). This bit stops the format-conversion compressor from taking samples from the ADC, but does initialize the ADPCM index. After CFIG3I[3] is set, system software must empty the remaining samples from the FIFO. When CFIG3I[3] is cleared, record continues as if no interruption ever occurred. If there is a FIFO overrun during ADPCM record, then the index used to compress the data loses synchronization with the file that is created; thus, when the file is played back, distortions will occur.

For the playback path, as with record, the *Playback Enable* bit of Configuration Register 1 (CFIG1I[0]) controls both the FIFO and the ADPCM index. When CFIG1I[0] is low, the FIFO is cleared and the ADPCM index in the format conversion's expander is initialized. Playback can be paused without initializing the index by discontinuing the data flow to the IC (for example, disabling the system's DMA controller from passing any data from system memory to the IC). Discontinuing the data flow causes a playback-FIFO underrun error which in turn causes the expander to stop (freezing the index). The underrun is reported as a sample error; during the underrun condition, the D/A Output Force Enable bit of Configuration Register 2 (CFIG2I[0]) determines what sample is sent to the DAC. After data flow is resumed, the expander continues. If any samples are skipped (with a FIFO overrun error),

then the ADPCM index is not synchronized with the data to be played and distortions will occur.

Sample Counters

The codec contains two 16-bit sample counters, one for playback and one for record. Each counter decrements one count for each sample loaded into the playback FIFO or unloaded from the record FIFO through DMA transfer. When the counter decrements to 0, the InterWave IC generates an interrupt (if not masked) and reloads the counters on the next sample transfer after the counter has reached zero. Program the load value of the counters with the Upper/Lower Record Count registers (CURCTI and CLRCTI) and the Upper/Lower Playback Count registers (CUPCTI and CLPCTI). Reading these registers returns the load value, not the current state of the counter. Check sample counter status in the *Record FIFO Interrupt Request* and *Playback FIFO Interrupt Request* fields of the Codec Status Register 3 (CSR3I[5] and CSR3I[4]).

Table 6-9 shows the relationship between the data format and the rate at which the sample counters decrement. To correctly program the counters, software must consider the data format to determine how many bytes represent a single sample. For example, 16-bit stereo data uses four bytes per sample and the counters decrement only after four bytes have been transferred. For information about setting the sample counters and tracking sample transfers with the DDK, see “Selecting Data Format and Sampling Rate” on page 6-25.

Table 6-9 Sample Counter Decrement Events

Sample Data Format	Event that causes the counter to decrement (sample event)
4-bit ADPCM mono	Every 4 bytes (8 mono samples) transferred through DMA or I/O cycles
4-bit ADPCM stereo	Every 4 bytes (4 stereo samples) transferred through DMA or I/O cycles
8-bit mono	Every byte (1 mono sample) transferred through DMA or I/O cycles
8-bit stereo	Every 2 bytes (1 stereo sample) transferred through DMA or I/O cycles
16-bit mono	Every 2 bytes (1 mono sample) transferred through DMA or I/O cycles
16-bit stereo	Every 4 bytes (1 stereo sample) transferred through DMA or I/O cycles

The information in Table 6-9 is adjusted for 16-bit DMA modes such that sample events depend only on the sample format and the number of bytes transferred and not on the DMA width. For example, if the codec is in 8-bit mono mode and two samples are transferred through a 16-bit DMA cycle, then the counter decrements twice. Table 6-9 also applies to serial transfers to the local memory record and playback FIFOs.

Only playback DMA transfers occur if the codec is in single-channel DMA mode and both the playback and record paths are enabled for DMA. All codec DMA transfers discontinue if the *DMA Transfer Disable* bit of the Codec Index Address register (CIDXR[5]) is High and an interrupt is set as indicated by the *Playback FIFO Interrupt Request* bit or the *Record FIFO Interrupt Request* bit of the Codec Status Register 3 (CSR3I[4] or CSR3I[5]) being High. I/O writes to a full playback FIFO and I/O reads from an empty record FIFO do not cause the sample counter to decrement.

Mode 1

In mode 1, the playback sample counter decrements when the playback path is enabled (the *Playback Enable* bit of Configuration Register 1 (CFIG1I[0]) set High) and a sample is loaded into the playback FIFO from system memory, or when the record path is enabled (the *Record Enable* bit of Configuration Register 1 (CFIG1I[1]) set High) and a sample is

unloaded from the record FIFO to system memory. When both playback and record are enabled, the sample events from both the record and playback paths cause the counter to decrement. The record sample counter is not available. The setting of the *Record FIFO I/O Select* bit or the *Playback FIFO I/O Select* bit in Configuration Register 1 (CFIG1I[7] or CFIG1I[6]) do not affect the sample counter's behavior.

Mode 2

In mode 2, the playback sample counter decrements when the playback path is enabled (CFIG1I[0] set High) and a sample is loaded into the playback FIFO from system memory. The record sample counter decrements when the record path is enabled (CFIG1I[1] set High) and a sample is unloaded from the record FIFO to system memory. The setting of the *Record FIFO I/O Select* bit or the *Playback FIFO I/O Select* bit in Configuration Register 1 (CFIG1I[7] or CFIG1I[6]) do not affect the sample counter's behavior.

Mode 3

In mode 3, the sample counters behave the same as in mode 2, with two exceptions: If the *Record FIFO I/O Select* bit or the *Playback FIFO I/O Select* bit in Configuration Register 1 (CFIG1I[7] or CFIG1I[6]) are set High, the relevant counter does not count. And the *Record Sample Counter Disable* bit or the *Playback Sample Counter Disable* bit of Configuration Register 3 (CFIG3I[5] or CFIG3I[4]) must be Low for the counter to operate.

In mode 3, the playback sample counter can be decremented by reading samples from the local memory playback FIFO or by DMA cycles. The record sample counter can be decremented by writing samples to the local memory record FIFO, by writes of samples from the synthesizer DSP to the local memory record FIFO, or by DMA cycles.

FIFO Error Conditions

Error conditions during FIFO operations generate flags. Table 6-10 lists these errors and their flags.

Table 6-10 FIFO Error Conditions

Error Condition	FIFO State	Action	Result
Playback FIFO Underrun	Playback FIFO empty	DAC needs another sample	In mode 1, the last sample in the FIFO will be reused; in modes 2 and 3, either the last sample will be reused or zeros will be used based on the state of CFIG2I[0]. The condition is reported in CSR1R[4], CSR2I[6], and CSR3I[0].
Playback FIFO Overrun	Playback FIFO full	System bus interface (SBI) writes another sample	The sample is thrown out and CSR1R[3:2] are not updated. The condition is reported in CSR3I[1].
Record FIFO Underrun	Record FIFO empty	System bus interface (SBI) reads another sample	The data is not valid and CSR1R[7:6] are not updated. The condition is reported in CSR3I[3].
Record FIFO Overrun	Record FIFO full	ADC gets another sample	The new sample is thrown out; condition is reported in CSR1R[4], CSR2I[7], and CSR3R[2].

Mixer

The mixer provides the analog I/O, signal routing, and mixing for the audio signals. Figure 6-2 on page 6-17 shows the left channel of the mixer.

All inputs to the summing stage as well as the line and mono outputs can be muted under program control.

Outputs

The IC provides the following outputs:

line Stereo single-ended line drivers.

mono Single-ended line driver provides the sum of the left and right line output signals. The source for the summation is after the output attenuator volume control.

Inputs

The InterWave IC provides the following inputs:

Stereo Microphone Inputs (MIC)
Software can feed this input pair to the ADC through the analog-to-digital (A/D) input multiplexer. This signal also sums into the output mixer. Software can program the attenuation (gain) associated with the output mixer path. The input can be muted.

Stereo Line Inputs (LINE)
Software can feed this input pair to the ADC through the A/D input multiplexer. This signal also sums into the output mixer. Software can program the attenuation (gain) associated with the output mixer path. The input can be muted.

Stereo Auxiliary 1 Inputs (AUX1)
Software can feed this input pair to the ADC through the A/D input multiplexer. This signal also sums into the output mixer. Software can program the attenuation (gain) associated with the output mixer path. The input can be muted.

Stereo Auxiliary 2 Inputs (AUX2)
This input pair sums into the output mixer path and does not connect to the ADCs. Software can feed the AUX2 signal into the ADC as part of the summed output of the output mixer. The attenuation (gain) is programmable. The input can be muted.

Synthesizer DAC Output (Internal)
The output of the synthesizer DAC pair feeds into the output mixer. The codec multiplexes this output with the AUX1 input through the *Aux 1/Synth Signal Select* bit of the Configuration Register 3 (CFIG3I[1]).

Mono Input The mono input feeds into both the left and right output mixer paths. The attenuation is programmable.

Loopback

The codec contains a loopback path from a point at the input to the ADCs to the output mixer. The attenuation is programmable.

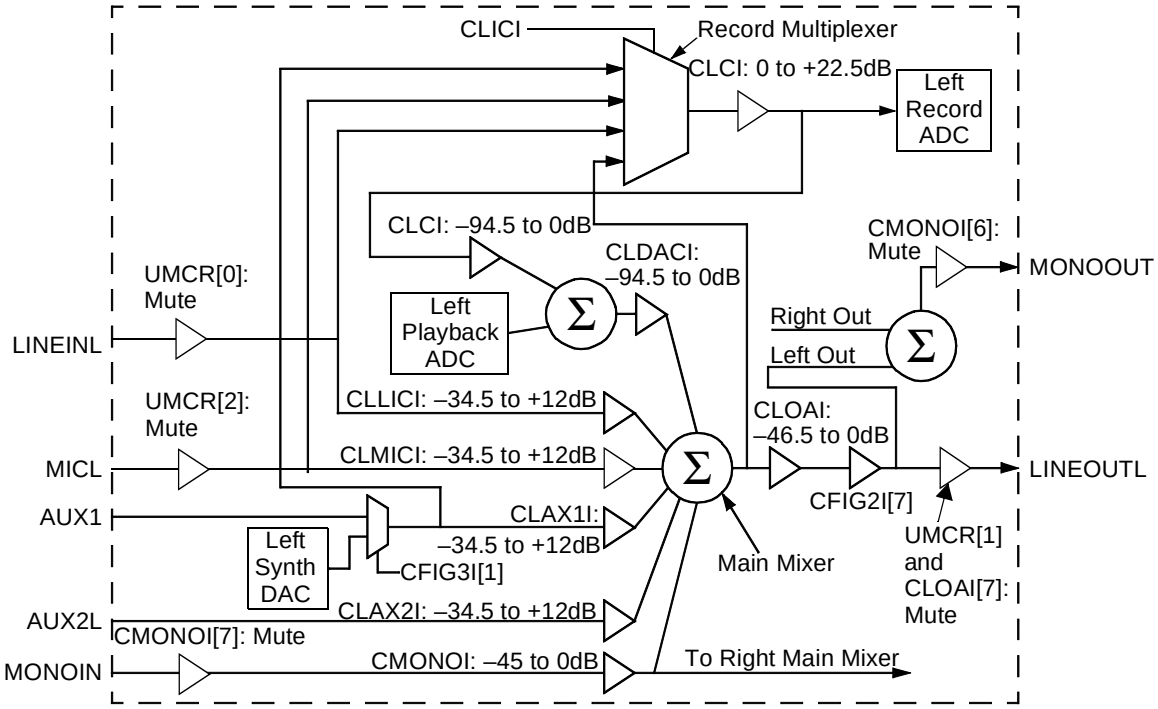
Output Mixer to ADC Path

The output of the output mixer can be fed into the ADCs through the ADC input multiplexer.

Signal Flow

Figure 6-2 shows the left channel of the stereo mixer; the right channel is identical. All the I/O pins are stereo except MONOIN and MONOOUT. Each of the labeled triangles represents attenuation (or gain) controls; the controlling register and value ranges are given. The three encircled sigma characters are signal summations or mixers. The two trapezoids are signal-selection multiplexers.

Figure 6-2 Left Half of the InterWave Mixer



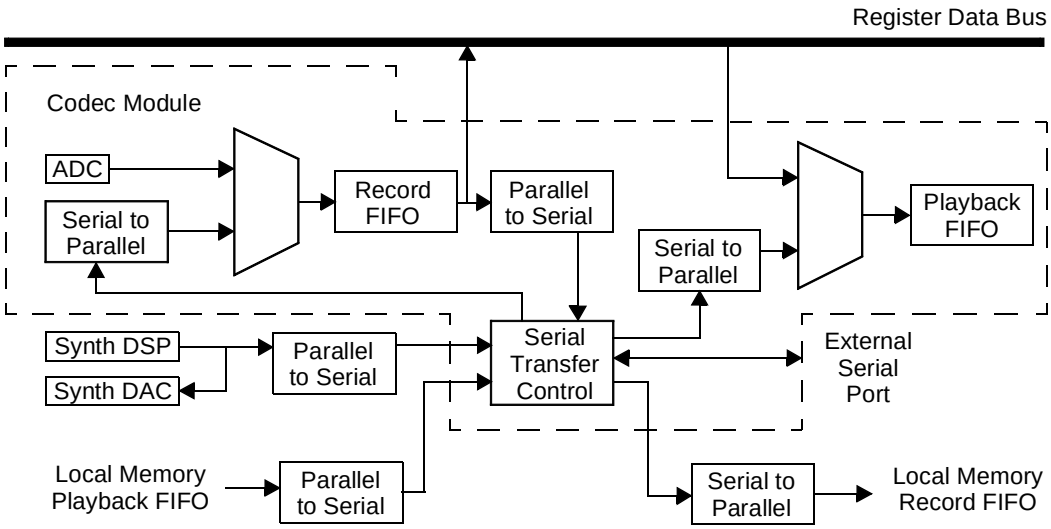
Serial Interface

The codec module includes a serial interface that allows the following functions:

- The synthesizer's digital signal processor (DSP) results can be transferred to an external serial DAC.
- An external general purpose DSP can access the codec FIFOs.
- The output of the record FIFO can be looped back to the playback FIFO.
- Synthesizer data can be transferred to the codec record and playback FIFOs.

The serial transfer control logic consists of serial-to-parallel converters, parallel-to-serial converters, bit-stream multiplexers, and state machines for the control signals and clock. Most of the transfers operate from a 2.1-MHz, 50-percent duty-cycle clock that is derived by dividing the 16.9344-MHz oscillator by 8. One exception is the transfers from the synthesizer DSP to an external DAC; these transfers utilize exactly 32 clocks per frame, based on the synthesizer frame rate. The other exception is the transfer of data between the codec and the local memory controller; these transfers are clocked at about 8 MHz (the 16.9344-MHz oscillator divided by 2). Figure 6-3 illustrates the flow of data through the codec.

Figure 6-3 Codec Data Flow



With the exception of local memory, the serial transfer control block multiplexes three sources and three destinations. The *Serial Transfer Mode* field of the Compatibility register (ICMPTI[7:5]) controls the possible modes, as shown in Table 6-11.

Table 6-11 Serial Transfer Data Flow and Format

ICMPTI[7:5]	Source	Destination	Format	Sampling Rate
0	Serial transfer mode not enabled			
1	Synth DSP	Record FIFO input	16-bit stereo	44.1 kHz
2	Synth DSP	Playback FIFO input	16-bit stereo	44.1 kHz
3	Record FIFO output	Playback FIFO input	CRDFI[3:0]	CRDFI[7:4]
4	Synth DSP	External serial port out	16-bit stereo	44.1 kHz + ?
5	Record FIFO output	External serial port out	CRDFI[3:0]	CRDFI[7:4]
	External serial port in	Playback FIFO input		

In general, if a codec FIFO is the destination, then software must adjust the format and sampling rate of that path to match the values in Table 6-11; otherwise, indeterminate data transfers result. For example, if ICMPTI[7:5] = 2, the playback path must be the same as the synthesizer (16-bit stereo, 44.1 kHz). Or if ICMPTI[7:5] = 3, the playback path must be set to match the record path. The modes whereby the synthesizer specifies the sampling rate can be slower than 44.1 kHz if the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is Low and the number of active voices set in the Synthesizer Active Voices (SAVI) register is greater than 14. If the synthesizer operates at other than 44.1 kHz, then ICMPTI[7:5] = 1 and ICMPTI[7:5] = 2 do not function properly because the codec does not support those rates. The modes in the above table cannot operate at the same time as the local memory record or playback FIFOs.

Multiplexed Pins

Setting ICMPTI[7] High changes the definitions of four pins to be used for the external serial port. See Appendix A, “Packaging and Pin Designations” for details.

Parallel-to-Serial Converters

Each parallel-to-serial converter brings in 16 bits of data to be shifted out serially, starting with bit 15 and ending with bit 0. The number of transfers, the data configuration, and the order of the data vary with the mode. The parallel-to-serial converters behave as shown in Table 6-12. For example, in 8-bit mono mode, there is one serial transfer for every two samples, with the first sample on the least significant bits (LSBs) and the second on the most significant bits (MSBs). In 16-bit stereo mode, there are two transfers for every sample received. The parallel-to-serial converters indicate that there is data ready to be transferred by setting a flag. The serial transfer control block responds by generating a pulse that is intended to initiate the flow of data, MSB first. After the 16 bits are transferred, a clear pulse is sent from the serial transfer control block so that new data can be loaded into the parallel-to-serial converters.

Table 6-12 Parallel-to-Serial Converter Data Ordering

Sample Data Format	Order (bits 15–0)	Samples Per Serial Transfer
4-bit ADPCM mono	S4, S3, S2, S1	4
4-bit ADPCM stereo	S2R, S2L, S1R, S1L	2
8-bit mono (linear, µ-law, A-law)	S2, S1	2
8-bit stereo (linear, µ-law, A-law)	S1R, S1L	1
16-bit mono little endian	S1[15:0]	1
16-bit mono big endian	S1[7:0], S1[15:8]	1
16-bit stereo little endian	S1L[15:0]*	1/2
16-bit stereo big endian	S1L[7:0], S1L[15:8]*	1/2

Notes:

1. S stands for sample and is followed by a number that implies the order.

2. L stands for left channel and R stands for right channel.

*3. * 16-bit stereo samples are sent left sample first, right sample second.*

Serial-to-Parallel Converters

The serial-to-parallel converters are 16-bit slaves to the bit streams that drive them. A pulse comes followed by the 16 bits of data, MSB first. As with the parallel-to-serial converters, the data configuration and order is the same as for 16-bit DMA.

Miscellaneous Functions

The codec module contains two miscellaneous functions as part of its compatibility with the CS4231 device.

Codec Timer

A programmable 16-bit timer is available in modes 2 and 3. To enable timer, set the *Timer Enable* bit of Configuration Register 2 (CFIG2I[6]) High. This timer has a resolution of approximately 10 µs.

Specify the 16-bit counter preset with the Upper Timer (CUTIMI) and Lower Timer (CLTIMI) registers. Writing to CLTIMI causes the combined 16-bit timer load value to be loaded into the coded timer, so CUTIMI should be written to first. The counter decrements every 10 µs until it reaches zero. At this point, the codec sets the *Timer Interrupt* bit in Codec Status Register 3 (CSR3I[6]) and the *Interrupt* bit in Codec Status Register 1 (CSR1R[0]) both

High and generates an interrupt. The codec reloads the counter with the value CUTIMI and CLTIMI on the next timer clock. To clear the timer interrupt, write a 0 to CSR3I[6] or write any value to CSR1R. For more information about clearing the interrupt, see Equation 6-6 on page 6-7.

For a sample codec timer program, see “Programming the Codec Timer” on page 6-23.

External Control Outputs

The codec multiplexes two InterWave pins between general purpose digital control outputs and digital signals driving the IRQ4 and IRQ10 lines. To select these pins as general outputs, set the *Select GPOUT1–GPOUT0 Codec Flags* bit of the Emulation IRQ register (IEIRQI[7]) High. When acting as output pins, the state of the pins reflects the state of the corresponding control bits in the External Control register (CEXTI[7:6]). For more information about these pins, see “IEIRQI—Emulation IRQ” on page 12-20.

Programming Tips and Examples

This section provides codec programming tips and examples using functions from the InterWave Driver Developer’s Kit (DDK). For detailed information about the DDK functions, see Chapter 21, “Codec/Mixer DDK Functions.”

Handling Codec Interrupts

When any one or more of the codec interrupt events occur, the *Global Interrupt Status* bit of Codec Status Register 1 (CSR1R[0]) goes High. Setting the *Global Interrupt Enable* bit of the External Control register (CEXTI[1]) High enables codec interrupts onto the ISA bus. Setting CEXTI[1] Low disables the interrupts. To enable codec interrupt requests onto the ISA bus, issue the following call to the **IwaveCodecIrq** DDK function:

```
IwaveCodecIrq(CODEC_IRQ_ENABLE)
```

To mask the codec interrupts from the ISA bus, issue the following call:

```
IwaveCodecIrq(~CODEC_IRQ_ENABLE)
```

The **IwaveHandler** DDK function routes interrupt requests from all sections of the InterWave IC to the appropriate callbacks. The code in Sample 6-1 corresponds to the code segment within **IwaveHandler** that routes codec interrupt requests. This segment of code goes through the following steps:

1. **Determine if a codec interrupt has occurred by looking at CSR1R[0].**
2. **If an interrupt has occurred, determine the source of the interrupt from CSR3I.**
3. **Issue a call to the appropriate callback.**

The callback must be registered by the application and its execution address stored inside a DDK-defined `iw` variable. In particular, there are three callbacks whose addresses should be stored within the following members of the `iw` variable:

codec_play_func
: The address of the callback for interrupts from the playback path. This function takes one argument, the contents of CSR3I, which contains important status information other than the interrupt source, such as FIFO error conditions.

codec_rec_func
: The address of the callback for interrupts from the record path. This function takes the same argument as **codec_play_func**.

codec_timer_func

The address of the callback for the codec timer. This callback accepts no arguments and returns nothing to the calling program.

In each of the three cases above, the interrupt reporting bit is always cleared to allow the reporting of other interrupts.

4. Clear the interrupt reporting bit in CSR3I.

The handler could have written any value to Codec Status Register 1 (CSR1R), which would clear all interrupt reporting bits.

Sample 6-1 Codec Interrupt Handler

```
void IwaveHandler (void)
{
  ... /* See DDK source code for complete handler */

  /* step 1 */
  if (_peek(iw.pcodar + 0x02) & CODEC_INT) {   /* Codec Interrupts? */
    BYTE source;
    ENTER_CRITICAL;
    _poke(iw.pcodar, _CSR3I);
    source = _peek(iw.cdatap);
    source &= (CODEC_PLAY_IRQ|CODEC_REC_IRQ|CODEC_TIMER_IRQ);

    /* step 2 for codec playback interrupt */
    if (source & CODEC_PLAY_IRQ) {
      /* step 3 */
      iw.codec_play_func(source);
      if ((iw.dma2 != NULL) && (iw.dma2->flags & CODEC_DMA) &&
      (iw.dma2->type == DMA_READ)) {
        iw.dma2->flags &= ~(DMA_BUSY|CODEC_DMA);
        iw.flags &= ~DMA_BUSY;
        iw.dma2->amnt_sent += iw.dma2->cur_size;
      }
      /* step 4 */
      _poke(iw.pcodar, _CSR3I);
      _poke(iw.cdatap, CODEC_REC_IRQ|CODEC_TIMER_IRQ);
    }
    /* step 2 for codec record interrupt */
    if (source & CODEC_REC_IRQ) {
      /* step 3 */
      iw.codec_rec_func(source);
      if ((iw.dma1 != NULL) && (iw.dma1->flags & CODEC_DMA) &&
      (iw.dma1->type == DMA_WRITE)) {
        iw.dma1->flags &= ~(DMA_BUSY|CODEC_DMA);
        iw.flags &= ~DMA_BUSY;
        iw.dma1->amnt_sent += iw.dma1->cur_size;
      }
      /* step 4 */
      _poke(iw.pcodar, _CSR3I);
      _poke(iw.cdatap, CODEC_PLAY_IRQ|CODEC_TIMER_IRQ);
    }

    if (source & CODEC_TIMER_IRQ) {
      iw.codec_timer_func();
```

```
      _poke(iw.pcodar, _CSR3I);
      _poke(iw.cdatap, (CODEC_PLAY_IRQ|CODEC_REC_IRQ));
    }
    LEAVE_CRITICAL;
  }
}
```

Transferring Data to the Codec Playback FIFO Using DMA

The **cplay.c** program in Sample 6-2 shows how to transfer data into or out of the codec using DMA transfer. This application is a **.wav** file player and recorder for any InterWave IC-based sound board and illustrates the steps to follow when setting up the InterWave IC and DMA controller for a transfer. To conduct DMA transfers to or from the codec, the DDK provides the **IwavePlayData** function for playback and the **IwaveRecordData** function for recording.

Sample 6-2 Servicing the Codec Playback FIFO Through DMA Transfer

```
/*////////////////////////////////////////////////////////////////////////////
/ FILE: cplay.c
/
/ REMARKS: This program illustrates the steps needed to conduct a DMA
/ transfer to the codec's playback FIFO.  Note how the transfer is described to
/ the DDK, how to register a callback and the DMA and IRQ structure variables.
/
////////////////////////////////////////////////////////////////////////////*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "iwdefs.h"
#include "iwprotos.h"
#include "iwcore.h"

BYTE playflag = 0;                    /* callback flag */

void CodecDma(BYTE source)
{
  if (source & CODEC_PLAY_IRQ)
    playflag++;                       /* flag completion of DMA xfer */
}

void main()
{
  IRQ irq1;
  DMA dma1;
  BYTE far *ptr, i;

  IwaveOpen(14, GUS_MODE);
  IwaveSetInterface(&dma1, NULL, &irq1, NULL); /* register DMA and IRQ structures */
  IwaveSetCallback(CodecDma, CODEC_PLAY_HANDLER); /* register callback */
  IwaveCodecMode(CODEC_MODE2);                    /* select codec mode */
  IwaveCodecIrq(CODEC_IRQ_ENABLE);                /* make sure codec can IRQ */

  if ((dma1.pc_ram = IwaveDmaMalloc(16)) == NULL) {  /* allocate memory for data */
```

```
    printf("Failed to allocate DMA buffer\n");
    goto bye;
  }
  ptr = (BYTE far *)dma1.pc_ram;

  for (i = 0; i <= 15; i++)           /* load buffer with some data */
    *ptr++ = i;
  IwaveDataFormat(BIT8_ULAW, _CPDFI);  /* set up data format */
  IwaveCodecCnt(PLAY, 16);            /* load codec's sample counter */
  dma1.flags |= CODEC_DMA;            /* this is a codec DMA xfer */
  dma1.type = DMA_READ;               /* tell DMA controller this is a download */

  if (IwaveDmaCtrl(&dma1, 16) != DMA_OK) {  /* program DMA controller */
    printf("DMA Failure\n");
    goto bye;
  }
  IwaveCodecTrigger(PLAYBACK);        /* trigger DMA */
  IwaveDmaWait();                     /* block until DMA is done */
  IwaveStopDma(PLAYBACK);             /* disable play path and DMA channel*/

bye:
  printf("playflag: %x\n", playflag);  /* flag is set by callback */
  farfree(dma1.pc_ram);               /* free buffer */
  IwaveClose();                       /* close down DDK and board */
}
```

Programming the Codec Timer

The DDK provides the following routines to program and start the codec's 10-µs timer.

IwaveSetTimer

Loads the Upper Timer and Lower Timer registers (CUTIMI and CLTIMI) with the specified value. For example, to have the timer decrement to zero in about 0.5 seconds issue the following call:

```
IwaveSetTimer(50000)
```

IwaveStartTimer

Starts the timer by setting the *Timer Enable* bit of Configuration Register 2 (CFIG2I[6]) High. Setting this bit also enables the timer to generate an interrupt as soon as the counter decrements to zero. If the software enables the codec interrupts onto the ISA bus by setting the Global Interrupt Enable bit of the External Control register (CEXTI[2]) High, a timer interrupt handler (provided by the application) can be entered when the timer decrements to zero.

IwaveStopTimer

Stops the timer. The timer counter continues to be reloaded with the contents of CUTIMI and CLTIMI and continues to decrement unless explicitly stopped. To disable the timer, issue a call to **IwaveStopTimer** with no arguments.

Programming the timer is a simple process. The **ctimer.exe** program in Sample 6-3 sets the timer to run for about 5 seconds. The counter is set up to generate an interrupt every half second and the interrupt handler updates a count flag that allows the main program to

determine the number of seconds already elapsed. This program can be used to test codec timer operation.

The important steps of the program are as follows:

1. **Initialize the InterWave IC and sound board.**
2. **Register a callback routine for the timer interrupt event.**

   A call to the **IwaveSetCallback** function associates the **CodecTimer** function, defined in the program, with the CODEC_TIMER_HANDLER message.
3. **Register an IRQ variable.**
4. **Ensure that the codec interrupt request is passed on to the ISA bus with a call to the IwaveCodecIrq function.**
5. **Load the timer counter through a call to IwaveSetTimer.**

   This function loads the Upper Timer (CUTIMI) and Lower Timer (CLTIMI) registers with a value of 25,000. This setting causes the timer to run for 0.25 seconds before generating an interrupt.
6. **After the timer is loaded and the timer interrupt enabled, start the timer with a call to IwaveStartTimer.**
7. **After 5 seconds, stop the timer with a call to the IwaveTimerStop function.**

   This step is necessary because the timer continues to run if it is not explicitly stopped.

During execution, the program monitors the contents of Codec Status Register 1 (CSR1R) and Codec Status Register 3 (CSR3I). A codec interrupt request sets CSR1R[0] High and the timer interrupt request sets CSR3I[6] High. Both of these bits should be zero at the end of the program.

Sample 6-3 Codec Timer Program

```
/*////////////////////////////////////////////////////////////////////////////
/ FILE: ctimer.c
/
/ REMARKS: This program runs the codec's timer for 5 seconds.  It programs the
/ timer to generate 2 interrupts per second.  A callback is defined and registered.
/
////////////////////////////////////////////////////////////////////////////*/

#include <stdio.h>
#include <conio.h>
#include "iwdefs.h"
#include "iwprotos.h"
#include "iwcore.h"

BYTE ticks = 0x00;
BYTE ctimer = 0x00;

void CodecTimer(void)                    /* Codec Timer callback */
{
  if (++ctimer == 4) {
    ticks++;
    ctimer = 0x00;
```

```
  }
}

void main()
{
  IRQ irq1;

  /* step 1 */
  IwaveOpen(14, GUS_MODE);              /* Initialize card and DDK */

  /* step 2 */
  IwaveSetCallback(CodecTimer, CODEC_TIMER_HANDLER); /* register timer callback */

  /* step 3 */
  IwaveRegisterIRQ(&irq1, NULL);        /* register IRQ variable */
  IwaveCodecMode(CODEC_MODE3);

  /* step 4 */
  IwaveCodecIrq(CODEC_IRQ_ENABLE);      /* make sure codec can IRQ */

  /* step 5 */
  IwaveSetTimer(25000);                 /* .25 seconds to interrupt */

  /* step 6 */
  IwaveTimerStart();                    /* set timer off */

  while(!kbhit() && ticks<5)            /* wait for 5 seconds to pass */
  { };

  /* step 7 */
  IwaveTimerStop();

  /* step 8 */
  IwaveClose();                         /* close down DDK and sound board */
}
```

Selecting Data Format and Sampling Rate

The **IwaveDataFormat** DDK function allows the calling program to select the data format. For example, to set the record data format to stereo, 8-bit μ-law, issue the following call:

```
IwaveDataFormat(BIT8_ULAW|STEREO, REC_DFORMAT)
```

To select mono instead of stereo, do not include the STEREO symbolic constant.

The **IwaveSetFrequency** DDK function allows the calling program to specify the sampling rate. For example, to set the playback sampling rate to 5.5 kHz, issue the following call:

```
IwaveSetFrequency(_CPDFI, 5500)
```

Setting the Sample Counters

Typically, programmers do not think in terms of *samples* but rather in terms of *bytes* of data. The **IwaveCodecCnt** DDK function takes this into account and allows a calling program to load the codec sample counters with the correct value by simply specifying the codec path, the data format, and the total number of bytes of data. For example, to load the playback counter with the appropriate value (512) for 2048 bytes of 16-bit big endian stereo data, use the following call:

```
IwaveCodecCnt(PLAY, 2048)
```

IwaveCodecCnt takes the relationships in Table 6-9 on page 6-14 into account when loading the sample counters. Therefore, the data format should be set first, perhaps using a call to **IwaveDataFormat**.

CHAPTER

7 SYNTHESIZER

The InterWave synthesizer uses data stored in off-chip DRAM or ROM, called *local memory*, to generate a digital output. The synthesizer sends its output to a dedicated synthesizer digital-to-analog converter (DAC) for conversion to analog audio signals.

This chapter covers the following topics:

- Synthesizer features—a brief description of the programmable functions
- Synthesizer basics—a brief description of the InterWave synthesizer's operation
- Register overview—an outline of the synthesizer programmable function
- Initialization—the states of the programmable registers after reset or power-up
- Interrupts—marking the boundaries between address ranges and volume segments
- Addressing wavetable data—looping, pitch control
- Envelope generation—volume segments
- Envelope variations—tremolo, panning, effects volume
- Low-frequency oscillators (LFOs) for tremolo and vibrato
- Delay-based effects—echo, reverb, chorus, flange
- Voice accumulation—combining all voices into the frame's output
- Loading patches
- Playing digital audio files—ENPCM mode
- Advanced Gravis UltraSound (GUS) compatibility mode

Synthesizer Features

The synthesizer offers the following programmable functions:

32 16-bit voices

The wavetable synthesizer offers 32 16-bit voices, all running at a 44.1-kHz frame rate. Each voice supports sample interpolation (for pitch control), envelope generation and volume control, tremolo, vibrato, and panning (stereo positioning). Up to eight of the voices can be designated as *effects processors* to handle special effects processing. A voice not designated as an effects processor is called a *signal voice*.

Wavetable data in local memory

The InterWave synthesizer accomplishes voice generation and processing by manipulating wavetable data, which is stored in local memory. The InterWave IC supports up to 16 MB of DRAM and 16 MB of ROM for local memory.

pitch control The address increment during playback of wavetable data determines the playback rate. A process of linear interpolation between wavetable data points smooths the output, filling in missing data values, especially when the data is played back at a rate different than its recorded rate. Playing back digital sound data at a rate different than the recording rate has the effect of changing the overall pitch of the sound, much like speeding up or slowing down a tape recorder during playback.

Volume The amplitude of a voice has three components: the volume envelope, the left and right amplitude (panning, or stereo positioning), and LFO effects (tremolo).

Low-frequency oscillators (LFOs) for tremolo, vibrato, and special effects

There are 64 LFOs—two associated with each of the 32 voices—that can cause pitch and volume variation. When used with signal voices, the LFOs produce tremolo (volume variation) and vibrato (pitch variation). Special effects such as chorus and flange result when the LFOs are used with effects-processor voices. Each LFO has programmable frequency, depth, and ramp-rate parameters. The LFOs require local memory for storing the programmable parameters and, therefore, do not operate in a ROM-only hardware application.

Delay-based effects

Voices can be programmed to provide delay-based effects. Effects can be assigned to individual voices or to any combination of voices.

Digital mixer with file playback

The synthesizer can serve as a digital mixer for playing back multiple digital audio files, such as **.wav** files. Feeding each file through a separate signal voice not only provides mixing, but also makes all the power of the synthesizer available to control each signal, such as LFOs and special effects.

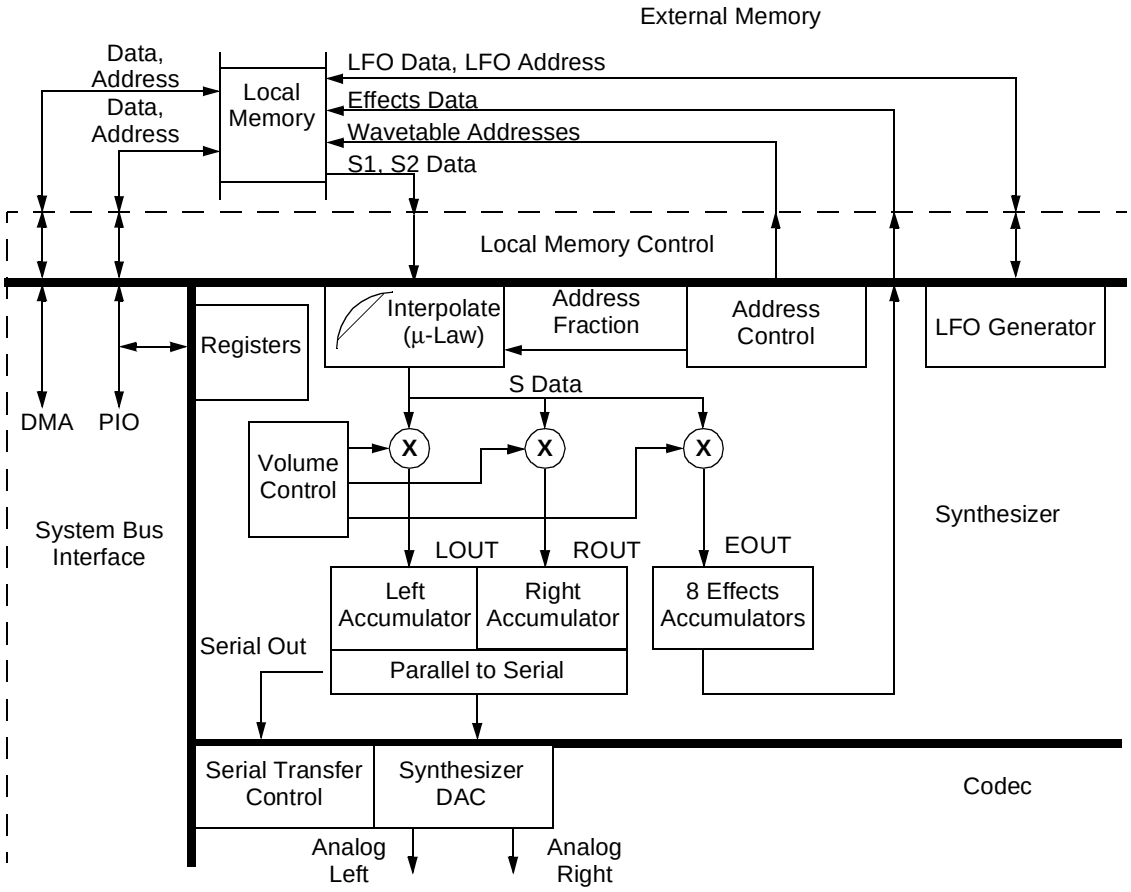
UltraSound compatibility

The InterWave synthesizer is backward compatible with the synthesizer used in the Advanced Gravis UltraSound (GUS) products.

Synthesizer Basics

The diagram in Figure 7-1 shows the synthesizer module's interfaces to the local memory control module, the system bus interface module and the codec module. It also shows the internal signal flow of logic contained within the synthesizer module.

Figure 7-1 Basic Synthesizer Data Paths



The synthesizer processes voices in frames. A sample frame produces one left and right digital output to the Synthesizer DAC. In each sample frame there are 32 slots, one for each voice. During each slot, one voice is individually processed through the signal paths shown in Figure 7-1.

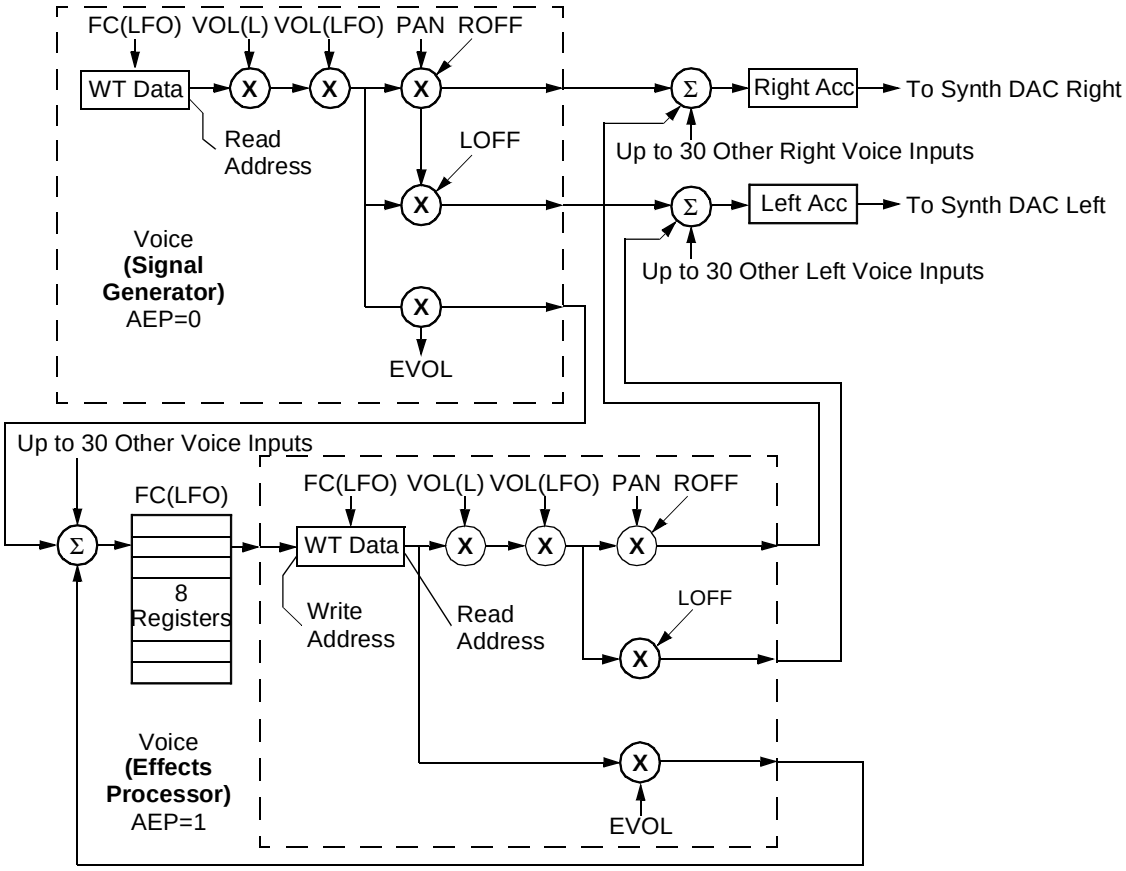
At the beginning of processing for each voice, two samples are read from wavetable data, S1 and S2. The wavetable address contains an integer and a fractional portion. The integer portion addresses S1 data and is incremented by 1 to address S2 data. The interpolation process uses the fractional portion when interpolating the sample, S data, from S1 and S2. Wavetable data can be μ-law compressed, in which case S1 and S2 are expanded before interpolation. The *μ-Law* bit of the Synthesizer Mode Select register (SMSI[6]) enables μ-law decompression of wavetable data.

S data is then multiplied by volume values that add envelope, low-frequency oscillator (LFO) variation, left and right stereo offset, and effects volume to produce three outputs: left (LOUT), right (ROUT), and effects (EOUT).

LOUT and ROUT connect to the left and right accumulators. EOUT sums into any, all, or none of the eight effects accumulators if effects are enabled. The left and right accumulators' data are output serially to the synthesizer DAC after processing all voices. Each effects accumulator can accumulate any, all, or none of the other voices during a sample frame. The outputs of the effects accumulators are written to local memory as wavetable data for an effects processor voice to read. By having the effects processor voice read the data at a later time, the InterWave IC can produce delay-based effects. A serial output to the codec module is also available.

The diagram in Figure 7-2 shows signal flow from one voice to another voice. In this figure, two of the 32 voices available are linked as a signal generator and an effects processor.

Figure 7-2 Envelope Generation and Effects Paths



Signal Voices

When the *Effects Processor Enable* bit of the Synthesizer Mode Select register (SMSI[0]) is Low for a particular voice, that voice acts as a signal generator and is called a *signal voice*. A signal voice plays back recorded data. A signal voice's input is wavetable data contained in off-chip ROM or DRAM. The addressing rate of the wavetable data, set with the Synthesizer Frequency Control (SFCI) register, controls the apparent pitch of the output. The address rate can be modified by a frequency LFO (FC(LFO)) to add vibrato to a tone. After interpolation, the sample data passes through three volume multiplying paths. The volume multiplying paths in Figure 7-2 are broken down to show the individual components of each volume multiply. The looping volume component (VOL(L)) can be looped and

ramped under register control and provides a mechanism for volume envelope generation. The volume LFO component VOL(LFO) adds LFO variations in volume resulting in tremolo. Then the signal path splits three ways. The top two paths generate right and left data outputs for the voice. The stereo positioning of a voice can be controlled in one of two ways:

- In GUS-compatibility mode, a single PAN value can be programmed, placing the signal in one of 16 pan positions from left to right.
- In Enhanced mode, separate left and right offset values (LOFF and ROFF) can be programmed to place the voice anywhere in the stereo field.

LOFF and ROFF can also be used to affect the total volume output. Left and right volume outputs for the voice are then summed with all other voices' left and right outputs and the summed output is converted to analog signals by the synthesizer DAC.

The effects volume (VOL(EVOL)) component controls the third signal path's volume. This output can go to any, all, or none of the effects accumulators. An effects accumulator sums all voice outputs assigned to it.

Effects Processor Voices

When the *Effects Processor Enable* bit of the Synthesizer Mode Select register (SMSI[0]) is High for a particular voice, that voice acts as an *effects processor*. An effects processor adds delay-based effects to voices. When a voice is performing effects processing, it writes one of the effects accumulators' outputs to local memory using the value in the Synthesizer Effects Address High/Low registers (SEAHI and SEALI) as the current write address. The current read address, as with all voices, is the value in the Synthesizer Address High/Low registers (SAHI and SALI). The difference between write and read addresses provides a delay for delay-based effects. The write address always increments by 1. The read address increments an average of 1, but can have variations in time added by an LFO. These variations in time generate chorus and flange effects. The volume components in the left and right path determine how much of the effect is heard and the stereo position of the effects processor's output.

Alternate Effects Signal Paths

The effects signal path for a voice can take one of two routes, as determined by the *Alternate Effects Path* bit of the Synthesizer Mode Select register (SMSI[4]). When SMSI[4] is High, the interpolated data is only affected by the effects volume component (EVOL) before it is fed back to the effects accumulators. If the effects path is fed back to the same effects accumulator, EVOL controls the decay of echoes heard. When SMSI[4] is Low, the interpolated data is affected by the looping volume component (VOL(L)), the LFO volume component (VOL(LFO)), and EVOL.

Register Overview

The following tables outline the synthesizer control functions that can be accomplished through the InterWave programmable registers. For detailed information about the registers, see the reference page listed in the last column of each table. These references point to the applicable section in Chapter 14, “Synthesizer Registers.”

Table 7-1 through Table 7-4 group the synthesizer functions into the following areas:

- General control and configuration functions
- Voice wavetable control functions
- Voice volume control functions

- IRQ functions

Table 7-1 Synthesizer General Control and Configuration Functions

Function	Register and Bit Field	Reference
Specify the number of active voices	SAVI[4:0]	14-1
Turn on auto-incrementing of IGIDXR (enhanced mode only)	SVSR[7]	14-1
Enable all LFOs	SGMI[1]	14-3
Enable enhanced mode operation	SGMI[0]	14-3
Specify bits 23–10 of the local memory base address for voice LFO parameters	SLFOBI[13:0]	14-3
Select the voice to be programmed	SVSR[4:0]	14-1

Table 7-2 Synthesizer Voice Wavetable Control Functions

Function	Register and Bit Field	Reference
Specify bits 23–22 of a voice's local memory address	SUAI[1:0]	14-4
Specify the starting address of a voice's wavetable data	SASHI, SASLI	14-4
Specify the ending address of a voice's wavetable data	SAEHI, SAELI	14-5
Specify the current address of a voice's wavetable data	SAHI, SALI	14-6
Specify the address where a voice's effects data is being written	SEAHI, SEALI	14-7
Specify the voice's frequency control value	SFCI	14-7
Read or specify the voice's frequency LFO value (vibrato)	SFLFOI	14-8
Specify the direction in which wavetable data is read	SACI[6]	14-8
Enable wavetable bidirectional looping (direction changes at start and end boundary addresses)	SACI[4]	14-8
Enable wavetable data looping	SACI[3]	14-8
Specify the data width for wavetable data	SACI[2]	14-8
Start and stop voice activity	SACI[1:0]	14-8
Read state of wavetable address looping	SACI[0]	14-8
Enable PCM operation	SVCI[2]	14-10
Select voice input data from DRAM or ROM	SMSI[7]	14-14
Enable μ-law decompression of voice input data	SMSI[6]	14-14
Deactivate a voice	SMSI[1]	14-14
Enable the alternate effects signal path	SMSI[4]	14-14
Enable the voice as an effects processor	SMSI[0]	14-14

Table 7-3 Synthesizer Voice Volume Control Functions

Function	Register and Bit Field	Reference
Specify the starting volume value (the low point of the volume ramp)	SVSI	page 14-9
Specify the ending volume value (the high point of the volume ramp)	SVEI	14-9
Specify the current volume level	SVLI	14-10
Specify the volume rate (how often the volume is incremented)	SVRI[7:6]	14-10
Specify the volume increment	SVRI[5:0]	14-10
Specify the direction of volume ramping	SVCI[6]	14-10
Enable volume ramp bidirectional looping (direction changes at start and end volumes)	SVCI[4]	14-10
Enable volume ramp looping	SVCI[3]	14-10
Start and stop volume looping	SVCI[1:0]	14-10
Read state of volume looping	SVCI[0]	14-10
Read or specify the voice's volume LFO value (tremolo)	SVLFOI	14-11
Specify the voice's pan value (if not in offset mode)	SROI[11:8]	14-12
Read or specify the voice's current right offset value (in offset mode)	SROI[15:4]	14-12
Specify the voice's target right offset value (in offset mode)	SROFI[15:4]	14-12
Read or specify the voice's current left offset value (in offset mode)	SLOI[15:4]	14-13
Specify the voice's target left offset value (in offset mode)	SLOFI[15:4]	14-13
Read or specify the voice's current effects volume value	SEVI[15:4]	14-13
Specify the voice's target effects volume value	SEVFI[15:4]	14-14
Select which effects accumulator receives the voice's effects output	SEASI	14-14
Enable offset mode	SMSI[5]	14-14

Table 7-4 Synthesizer IRQ Functions

Function	Register and Bit Field	Reference
Determine which voice has generated an interrupt	SVII[4:0]	14-2
Read the status of the wavetable address boundary IRQ for the specified voice	SVII[7]	14-2
Read the status of the volume boundary IRQ for the specified voice	SVII[6]	14-2
Determine which voice has generated an interrupt without clearing IRQs	SVIRI[4:0]	14-2
Read the status of the wavetable address boundary IRQ w/o clearing	SVIRI[7]	14-2
Read the status of the volume boundary IRQ w/o clearing	SVIRI[6]	14-2
Read, clear, or set the wavetable address boundary IRQ	SACI[7]	14-8
Enable or clear the wavetable address boundary IRQ	SACI[5]	14-8
Read, clear, or set the status of the volume boundary IRQ	SVCI[7]	14-10
Enable or clear the volume boundary IRQ	SVCI[5]	14-10

Initialization

All registers are initialized to their default settings at power-up time or by a hardware reset. The default settings are defined in Chapter 14, "Synthesizer Registers." The global registers are initialized by a hardware reset and the register array containing the voice-specific registers is initialized with a 128-clock sequence following the hardware reset. For more information about the voice-specific register array, see "Programming Voice-Specific Registers" on page 7-30.

Setting the *Reset GUS* bit of the GUS Reset register (URSTI[0]) Low resets several of the synthesizer registers. For a complete list of registers and events reset through URSTI[0], see "URSTI—GUS Reset" on page 12-14.

After a return from suspend mode, the synthesizer registers are in the states they were in before SUSPEND became active, provided that suspend-mode DRAM refreshing took place—see Chapter 8, "Local Memory Control."

Interrupts

Each active synthesizer voice can generate address and volume boundary interrupts. Both address and volume interrupts are handled identically by internal hardware in terms of reporting and clearing. There are three levels of reporting for these two types of interrupts. When a boundary is crossed during voice processing, depending on the boundary, either the voice-specific *Wavetable IRQ* bit of the Synthesizer Address Control register (SACI[7]) or the voice-specific *Volume IRQ* bit of the Synthesizer Volume Control register (SVCI[7]) indicates the type of interrupt and the global *Wavetable IRQ* bit or *Volume IRQ* bit of the Synthesizer Voices IRQ register (SVII[7] or SVII[6]) is set Low. SVII also contains the number of the voice that caused the interrupt (SVII[4:0]). SVII[7] and SVII[6] are mirrored in the *Volume Loop IRQ* and *Address Loop IRQ* bits of the IRQ Status register (UISR[6] and UISR[5]). A program should read UISR to determine the source of the interrupt. Then, when the program writes a value of 8Fh to the General Index register (IGIDXR) to index SVII, the contents of SVII are latched and the process of clearing all three levels of reporting begins. UISR[6] and UISR[5] are cleared shortly after the write of 8Fh to IGIDXR. When the voice that caused the interrupt is next processed, SACI[7] and SVCI[7] are cleared.

Multiple voice interrupts can be "stacked" by the voice interrupt reporting logic. If another voice reaches a boundary during processing and SVII already contains an active interrupt, the voice-specific SACI[7] or SVCI[7] holds the new interrupt until the active interrupt has been cleared from SVII. SVII is then updated with the new interrupt during processing of the new interrupting voice.

SVII[7:6] and the number of the voice that caused an interrupt can be observed by reading the Synthesizer Voices IRQ Read register (SVIRI). Reading SVIRI does not clear any stored interrupt reporting bits. Reading this register allows a program to check the interrupt reporting bits and change the boundary condition that caused the interrupt before clearing the interrupt reporting bits. If only SVII is read, it is possible to have multiple interrupts reported for the same boundary condition.

The Frame/Voice Structure

The InterWave synthesizer can process up to 32 voices continuously. A *frame* is the basic time unit in the synthesizer. During one frame, all 32 voices can be processed and their outputs summed. Thus, the output of one frame is the sum of all voices that were active during that frame. At the end of each frame, one right and one left output is passed to the synthesizer DAC (located in the codec module).

The *frame rate* is 44.1 kHz; that is, all 32 voices are processed 44,100 times each second.

Note: For compatibility with the GUS, the frame rate can be adjusted downward, with a trade-off in performance. See "GUS Frame Expansion" on page 7-29.

Certain components of a sound—for example, a gradually increasing volume—can carry over to the next frame. These components are retained in the appropriate registers as the starting point for the next frame.

Addressing Wavetable Data

This section discusses the addressing of wavetable data in local memory DRAM or ROM. Chapter 8, "Local Memory Control," contains additional information about local memory addressing.

Address Control

Voice generation starts with the wavetable data in local memory addressed by the value in the Synthesizer Address High and Synthesizer Address Low registers (SALI and SAHI). Computation of the next value to be stored in the synthesizer address registers is controlled by the following bits:

- *Enable PCM Operation* bit of the Synthesizer Volume Control register (SVCI[2])
- *Loop Enable* bit of the Synthesizer Address Control register (SACI[3])
- *Bidirectional Loop Enable* bit of the Synthesizer Address Control register (SACI[4])
- *Direction* bit of the Synthesizer Address Control register (SACI[6])

Address Looping

As the synthesis process takes place, the synthesizer steps through the wavetable data stored in local memory. Depending on the nature of the stored signal, the synthesizer may step directly through the stored data, or it may be programmed to loop through some range of data. For example, to imitate a key of an organ being held down for some period of time, the synthesizer may repeat one stored cycle over and over. Looping can reduce the amount of local memory needed for repetitive wavetable data. The synthesizer addresses local memory in the following patterns, as shown in Figure 7-3 through Figure 7-5.

- Single pass
- Forward loop—start at any point (lower address), step up through memory to an end boundary, then return to a start boundary to start stepping up again
- Reverse loop—start at any point (higher address), step down through memory to a start boundary, then return to an end boundary to start stepping down again
- Bidirectional loop or Zigzag—start at any point, step up through memory to an end boundary, and then step back down to the start boundary
- Play back digital files

As the synthesizer steps through the wavetable data, it uses a process of interpolation to smooth out the voice, especially when processing data at a frequency different than the original recording rate. See "Sample Interpolation" on page 7-13.

Figure 7-3 through Figure 7-5 shows six graphs of address looping control possibilities. An interrupt, if enabled, is generated each time an address boundary is crossed. The Synthesizer Address Start High and Synthesizer Address Start Low registers (SASHI and SASLI) hold the starting address boundary, and the Synthesizer Address End High and

Synthesizer Address End Low registers (SAEHI and SAELI) hold the ending address boundary.

Figure 7-3 Forward and Reverse Single-Pass Addressing

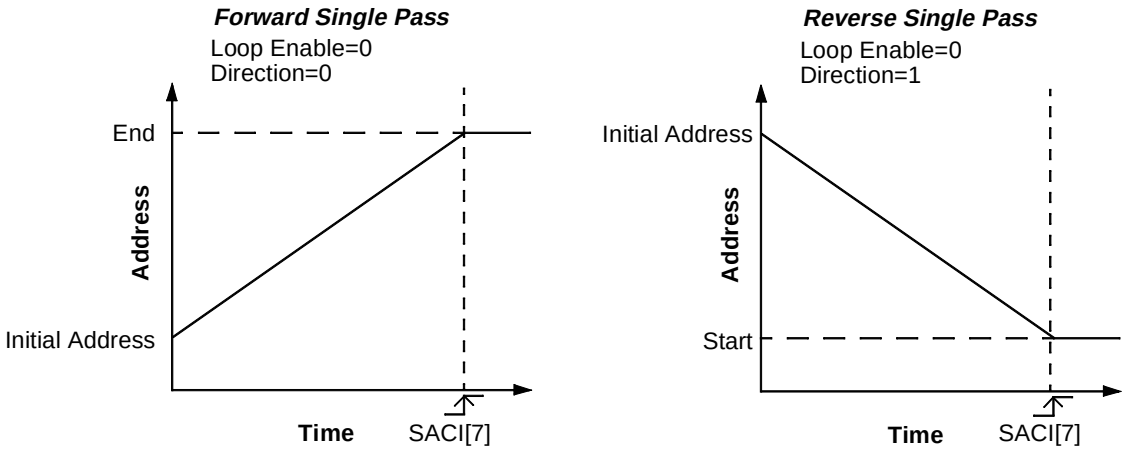


Figure 7-4 Forward and Reverse Looping

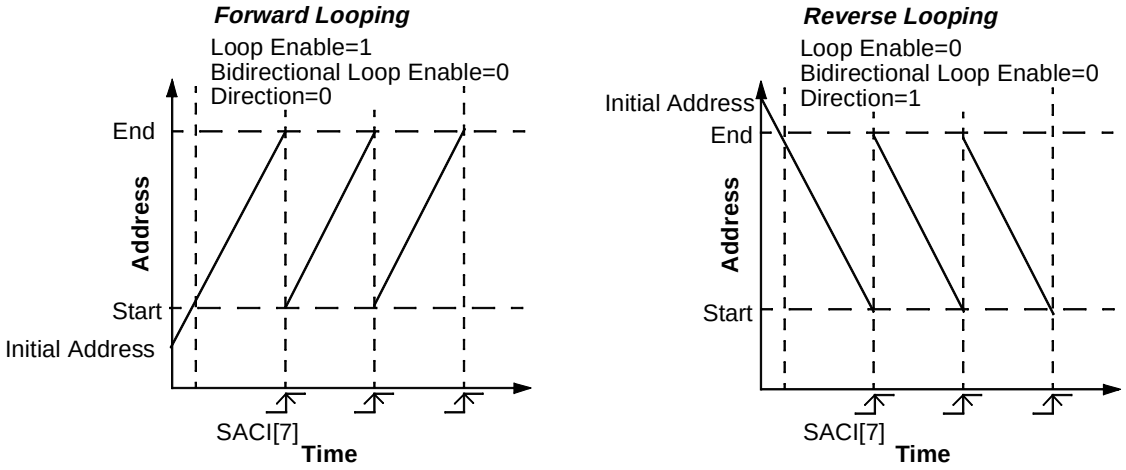
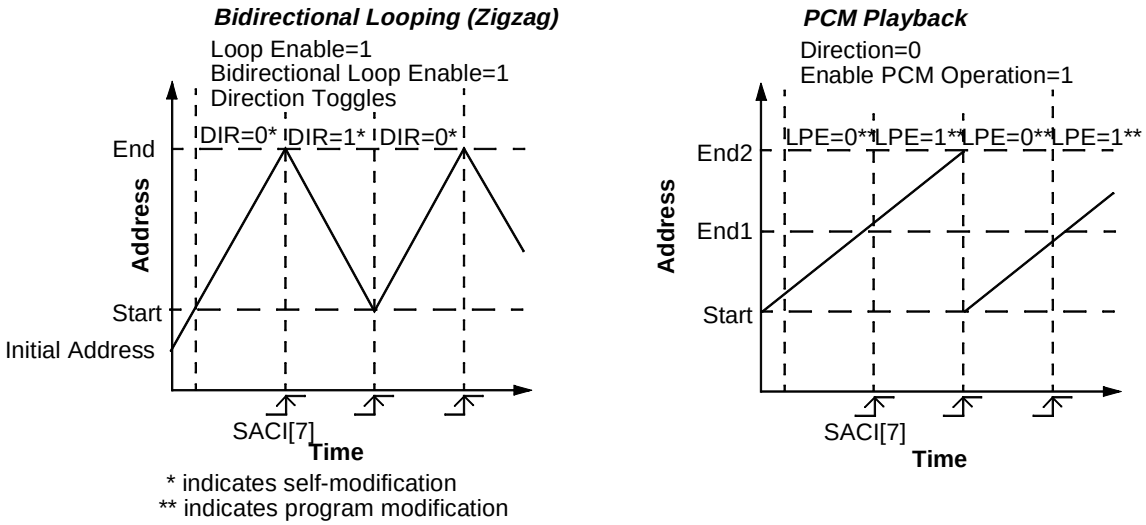


Figure 7-5 Bidirectional Looping (Zigzag) and PCM Playback



The address control logic also controls the write address for delay-based effects processing—see “Delay-Based Effects” on page 7-26.

Frequency Control

The FC(LFO) component (see Figure 7-2 on page 7-4) controls the rate at which the Synthesizer Address register pair (SAHI and SALI) is incremented or decremented. FC(LFO) is made up of the subcomponents FC and FLFO. FC is the value programmed into the Synthesizer Frequency Control register (SFCI). FLFO, the value in the Synthesizer Frequency LFO register (SFLFOI), is added to FC before the address calculations are done. FLFO is a signed value. If FLFO is negative, the pitch of the voice is lowered. If FLFO is positive, the pitch of the voice is raised.

Computing Next Address

Table 7-5 shows how all combinations of wavetable addressing control, along with the boundary crossed (BC) internal flag, affect the next wavetable address. BC becomes a 1 when (END – (ADD + FC(LFO))) is negative and the *Direction* bit of the Synthesizer Address Control register (SACI[6]) is low or when (ADD – FC(LFO)) – START) is negative and SACI[6] is high. If BC is high, an interrupt is generated if enabled by the *Wavetable IRQ Enable* bit (SACI[7]). The Next Address column indicates the expressions used to compute the next address value using ADD (the current address value), FC(LFO), START, and END. The current address value is contained in the Synthesizer Address High and Synthesizer Address Low registers (SAHI and SALI). START and END are the address boundaries for address looping contained in the Synthesizer Address Start High and Synthesizer Address Start Low registers (SASHI and SASLI) and the Synthesizer Address End High and Synthesizer Address End Low registers (SAEHI and SAELI).

Table 7-5 Wavetable Addressing Control

Enable PCM Operation (SVCI[2])	Loop Enable (SACI[3])	Bidirectional Loop Enable (SACI[4])	Direction (SACI[6])	Boundary Crossed (BC)	Next Address
X	X	X	0	0	ADD + FC(LFO)
X	X	X	1	0	ADD – FC(LFO)
0	0	X	X	1	ADD
X	1	0	0	1	START – (END – (ADD + FC(LFO)))
X	1	0	1	1	END + ((ADD – FC(LFO)) – START)
X	1	1	0	1	END + (END – (ADD + FC(LFO)))
X	1	1	1	1	START – ((ADD – FC(LFO)) – START)
1	0	X	0	X	ADD + FC(LFO)
1	0	X	1	X	ADD – FC(LFO)

END to START Interpolation

Discontinuities in a voice's signal can be caused when the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is Low, the *Loop Enable* bit of the Synthesizer Address Control register (SACI[3]) is High, and the *Bidirectional Loop Enable* bit (SACI[4]) is Low if the data at the END and START addresses are not the same. The discontinuity occurs because with SGMI[0] = 0, there is no way to interpolate between data at the END address and data at the START address.

Setting SGMI[0], the *Enable PCM Operation* bit (SVCI[2]), and the *Loop Enable* bit (SACI[3]) all High and the *Bidirectional Loop Enable* bit (SACI[4]) and the *Direction* bit (SACI[6]) both Low enables the synthesizer module to interpolate between data at the END address and data at the START address. This mode of interpolation should be used during digital file playback and when a voice is being used as an effects processor. In this mode, the interrupt normally generated when the END address is crossed is not generated until the data at the END address is no longer needed for interpolation. For more details about interpolation, see “Sample Interpolation” on page 7-13.

DRAM and ROM Access

When the *ROM* bit of the Synthesizer Mode Select register (SMSI[7]) is set Low, a voice uses 8-bit-wide DRAM to obtain both 8-bit and 16-bit data samples. For voices that use 8-bit data, all the addresses in the address registers represent actual local memory addresses. For voices that use 16-bit data, a translation is done from the addresses in the address registers to the real address space. The translation allows the synthesizer module to generate addresses for 8-bit and 16-bit data in the same way and for the local memory control module to use DRAM fast page mode to access two 8-bit values to provide a 16-bit sample. Address translation is explained in “Accessing Local Memory” on page 8-8.

When the *ROM* bit of the Synthesizer Mode Select register (SMSI[7]) is set High, a voice uses 16-bit wide ROM to obtain both 8-bit and 16-bit data samples. For voices that use 8-bit data, the least significant bit (LSB) of the address is kept for internal use only to determine which byte of the 16-bit wide ROM word is used. If the LSB is 0, the lower byte of the word is used as sample data; if the LSB is 1, the upper byte of the word is used. For voices that use 16-bit data, the address directly addresses the ROM.

µ-Law Data Decompression

To conserve memory space, µ-law compressed wavetable data may be stored in local memory. When µ-law decompression is selected, the data is automatically decompressed by the synthesizer when it is read from local memory. The µ-law compression algorithm converts 14-bit linear data to an 8-bit format. Decompression restores the data to 14-bit linear form and aligns it in the 16-bit linear format used by the synthesizer.

To select µ-law decompression, set the *µ-Law* bit of the Synthesizer Mode Select register (SMSI[6]) High.

Sample Interpolation

During voice generation, the synthesizer fetches sample 1 (S1) from the wavetable data in DRAM using the integer portion of the current Synthesizer Address High/Low registers (SAHI and SALI), then increments the integer portion by 1 and uses it to fetch sample 2 (S2). The synthesizer then obtains the interpolated sample (S) by plugging S1, S2, and the fractional portion (ADDfr) of the synthesizer address into Equation 7-1.

Equation 7-1 S Data Interpolation

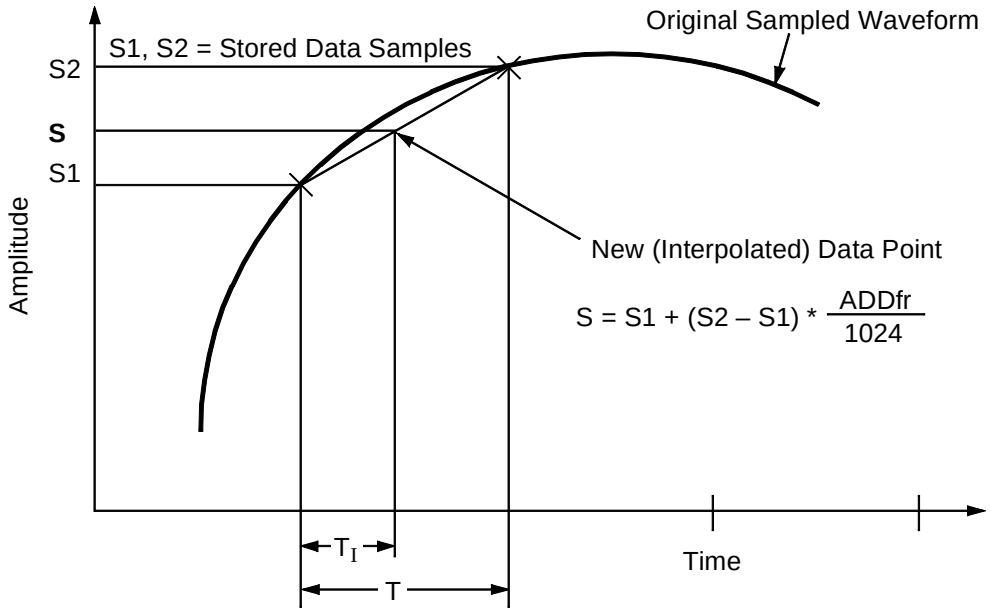
$$S = S1 + (S2 - S1) \cdot \frac{ADDfr}{1024}$$

Software specifies the10-bit fractional portion of the synthesizer address in the Synthesizer Frequency Control register (SFCI[9:0]). However, the Synthesizer Address Low register (SALI) contains only the 9 most significant bits of that fractional portion. The 1024 divisor in Equation 7-1 is needed to correctly multiply by a 10-bit fractional number.

The synthesizer interpolates 8-bit and 16-bit samples in the same fashion, padding the 8-bit samples with zeros in the least significant bits.

Figure 7-6 illustrates the interpolation process.

Figure 7-6 Graph of Sample Interpolation Process



Vibrato—Varying the Pitch

One of the two LFOs assigned to each voice can continuously vary the wavetable address increment, thus adding vibrato to the voice.

For details on programming a voice's LFO for vibrato, see "LFOs for Tremolo and Vibrato" on page 7-21.

Volume Control

The InterWave IC uses three volume-multiplying signal paths—left, right, and effects—to process each voice. Each path's volume multiplier is made up of several components. After each component is calculated, the components are summed and used to control the volume of the three signal paths. Equation 7-2 through Equation 7-5 shows the equations for each of the three signal paths and the alternate effects path. Programming the terms of these equations is covered in the sections that follow.

Equation 7-2 Volume Multiplying Components For Left Path

$$\text{Volume Left} = VOL(L) + VOL(LFO) - LOFF$$

Equation 7-3 Volume Multiplying Components For Right Path

$$\text{Volume Right} = VOL(L) + VOL(LFO) - ROFF$$

Equation 7-4 Volume Multiplying Components For Effects Path (SMSI[4] = 0)

$$\text{Volume Effects} = VOL(L) + VOL(LFO) - EVOL$$

Equation 7-5 Volume Multiplying Components For Alternate Effects Path (SMSI[4] = 1)

$$\text{Volume Effects} = EVOL$$

After all the volume components are generated, they are summed for each signal path's volume multiply. The theoretical equation for volume multiplication is shown in Equation 7-6.

Equation 7-6 Volume Multiplication

$$O = S \cdot 2^{(V/256) - 16}$$

In Equation 7-6, *O* is the output data, *V* is the value of volume and *S* is the interpolated sample value. An increment of 1 in the value of *V* causes about 0.0235 dB of change in *O*. This equation is difficult to implement directly in digital logic because of the exponential term, but a piecewise linear approximation is relatively easy to implement. The sum of each volume is a 12-bit value. The synthesizer splits the 12-bit value into two bit fields, V[11:8] and V[7:0], and uses those fields to implement an approximation, as shown in Equation 7-7.

Equation 7-7 Implemented Volume Multiplication

$$O = S \cdot \left(\frac{256 + V[7:0]}{2^{24 - V[11:8]}} \right)$$

The synthesizer uses Equation 7-7 to generate each of the outputs: a right voice output, a left voice output, and an effects output. The error introduced by the approximation for $0 \le V \le 4095$ ranges from 0 dB–0.52 dB with an average of 0.34 dB. Differences in power of less than 1 dB are not perceptible to the human ear, so there is no perceived error in the output power introduced by the implementation.

The Basic Envelope Segments—VOL(L)

The looping volume (VOL(L)) component adds the basic volume envelope segments to a voice.

Every sound has its own volume envelope. For example, a plucked guitar string has the following envelop segments:

- *attack*, which is the rapid initial increase in volume when the string is first plucked
- *decay*, a fall-off in volume from the high initial level
- *sustain*, a fairly constant volume level while the string is vibrating, and
- *release*, the quick down-ramp of volume when the player damps the string with a finger.

The synthesizer generates envelope segments individually. A note may have more or fewer segments than in the above example; the synthesizer can generate as many as necessary.

Each segment can be a ramp up, a ramp down, a forward loop, a reverse loop, or a bi-directional loop between two volume levels.The ramp rate is programmable in terms of the starting point, the amount of change per frame time, and the end boundary. When the volume reaches a segment boundary, a maskable interrupt is generated. The envelope generation mechanism operates similarly to the address generation mechanism, allowing the generation of bidirectional looping segments; that is, the synthesizer can be programmed to ramp up to a volume boundary and then ramp down to a second boundary (in actuality, the ramp up starts at zero, or wherever the envelope happens to be, and ramps to a point specified as the end boundary, then it ramps down to a point specified as the start boundary).

For step-by-step instructions on how to process volume envelope segments, see “Processing Volume Envelope Segments” on page 7-34.

Computing VOL(L)

Computation of the next value stored in the Synthesizer Volume Level register (SVLI) is controlled by the following 3 bits in the Synthesizer Volume Control register (SVCI):

- *Loop Enable* (SVCI[3])
- *Bidirectional Loop Enable* (SVCI[4])
- *Direction* (SVCI[6])

The following graphs show five volume looping possibilities. If enabled, an interrupt is generated each time the volume crosses a boundary. Volume boundaries are held in the Synthesizer Volume Start register (SVSI) and the Synthesizer Volume End register (SVEI).

Figure 7-7 Volume Ramp-up and Ramp-down

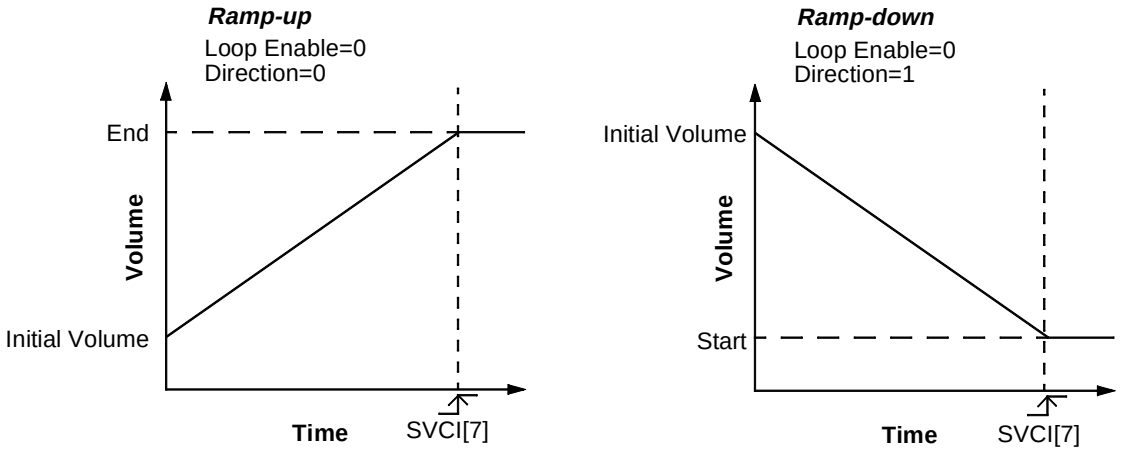


Figure 7-8 Forward and Reverse Volume Looping

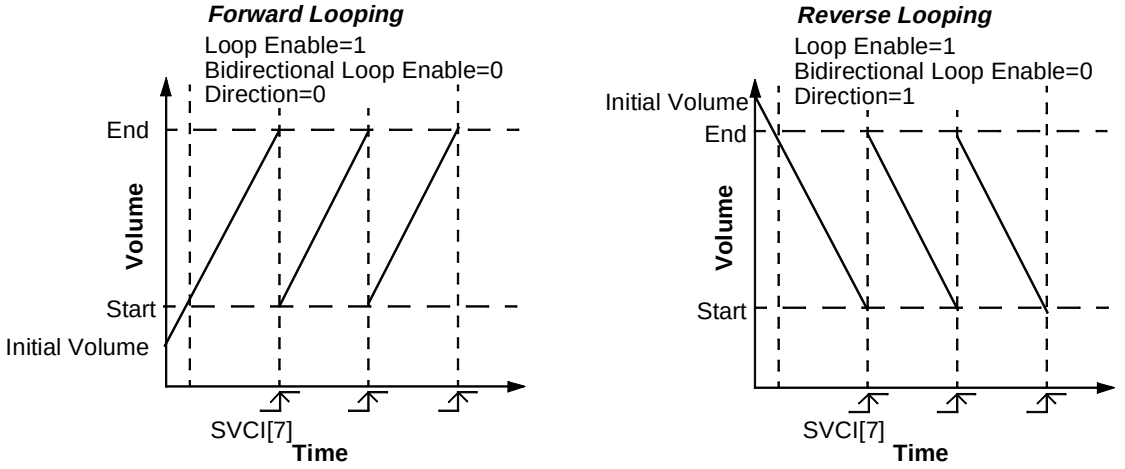
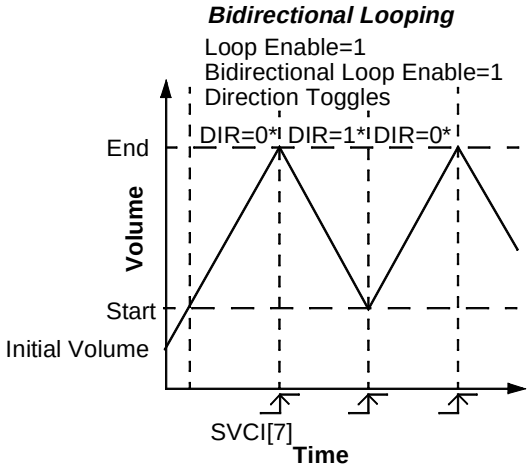


Figure 7-9 Bidirectional Volume Looping



* indicates self-modification

Table 7-6 shows how all combinations of volume control along with the input update volume (UVOL) and boundary crossed (BC) internal flag affect the equation for the next VOL(L). UVOL is an internal flag that controls the rate at which VOL(L) is modified. Volume rate bits in the Synthesizer Volume Rate register (SVRI) set the rate of VOL(L) modification. UVOL remains at 0 until the voice has been processed the number of times set by the volume rate bits. When UVOL becomes a 1, VOL(L) increments under the control of the *Loop Enable* (SVCI[3]), *Bidirectional Loop Enable* (SVCI[4]), and *Direction* (SVCI[6]) bits. BC becomes a 1 when volume boundaries are crossed. BC generates an interrupt if enabled by the *Volume IRQ Enable* bit in the Synthesizer Volume Control register (SVCI). The Next VOL(L) column indicates the expressions used to compute the next value of VOL(L) using VOL(L), the volume increment (VINC), the start volume (START), and the end volume (END). VINC is held in the Synthesizer Volume rate register. START and END are the volume boundaries for volume looping contained in the Synthesizer Volume Start register (SVSI) and the Synthesizer Volume End register (SVEI), respectively.

Table 7-6 Volume Control Combinations

Update Volume (UVOL)	Loop Enable (SVCI[3])	Bidirectional Loop Enable (SVCI[4])	Direction (SVCI[6])	Boundary Cross (BC)	Next VOL(L)
0	X	X	X	X	VOL(L)
1	X	X	0	0	VOL(L) + VINC
1	X	X	1	0	VOL(L) – VINC
1	0	X	X	1	VOL(L)
1	1	0	0	1	START – (END - (VOL(L) + VINC))
1	1	0	1	1	END + ((VOL(L) – VINC) – START)
1	1	1	0	1	END + (END – (VOL(L) + VINC))
1	1	1	1	1	START – ((VOL(L) – VINC) – START)

Ramp Rates—Rate of Volume Change

VOL(L) can be updated once per frame (44,100 updates per second). An increase of one in VOL(L) changes the volume by 0.0235 dB.

Ramp rates have four programmable ranges. Two ranges cause the envelope to be modified every frame time, one modifies the envelope every eighth frame time, and one modifies the envelope every sixty-fourth frame time.

The ramp rate is programmed by writing two values to the Synthesizer Volume Rate register (SVRI):

- *Volume Increment* (SVRI[5:0])—the amount of volume increment
- *Volume Rate* (SVRI[7:6])—one of four ramp rates:
  - — 0 = add *increment value* every frame
  - — 1 = add (*increment value*)/8 every frame
  - — 2 = add (*increment value*)/8 every 8th frame
  - — 3 = add (*increment value*)/8 every 64th frame

Envelope Variations

The InterWave IC supports three types of variations in the volume envelope of a voice:

- Tremolo
- Stereo positioning, or panning
- Effects

Tremolo—VOL(LFO)

The synthesizer can use a low-frequency oscillator (LF0) to continuously modify a voice's volume, which creates a tremolo effect. The Synthesizer Volume LFO register (SVLFOI) contains the value of VOL(LFO), which is the final result of LFO calculations.

For details on using the LFOs, see “LFOs for Tremolo and Vibrato” on page 7-21.

Stereo Positioning—Offset and Pan

The synthesizer controls stereo positioning of a voice in one of two ways. In offset mode, left and right offsets can place the voice anywhere in the stereo field. In pan mode, a single pan value places the voice in one of 16 pan positions. The *Offset Enable* bit of the Synthesizer Mode Select register (SMSI[5]) controls the two different modes of stereo positioning.

Offset Mode

When SMSI[5] is set High, the offset values allow independent attenuation of the left and right channels. To calculate left and right offset values with constant total power, use Equation 7-8 and Equation 7-9. Equation 7-10 calculates the attenuation resulting from an given offset value.

Equation 7-8 Left Offset Value

$$LeftOffset = 128 \cdot \log_2\left(\frac{PanMax - Pan}{PanMax}\right)$$

Equation 7-9 Right Offset Value

$$RightOffset = 128 \cdot \log_2\left(\frac{Pan}{PanMax}\right)$$

Equation 7-10 Attenuation by Offset Values

$$Attenuation = 20 \cdot \log_{10}\left(2^{-\frac{offset}{256}}\right) dB$$

PanMax +1 is the total number of pan positions desired. *Pan* is the stereo position desired between 0 and *PanMax*.

The left and right offsets allow very accurate stereo positioning. They also provide a separate volume control from all the other volume components. Writing all bits high to the

left or right offset turns off the respective output because, if the volume sum of the left or right component becomes negative, the volume multiplier is set to maximum attenuation for that path. Software can control the total volume output of a voice by considering the left and right offsets to be made up of two components. One component controls stereo position and is unique to the left or the right offset and the other component is common to both offsets and controls the overall volume. By combining the two components in software, writing the appropriate values to the Synthesizer Offset registers can control both overall volume and the stereo position.

In offset mode, two registers control the value of each offset. The Synthesizer Right Offset (SROI) and Synthesizer Left Offset (SLOI) registers contain the current values of right offset (ROFF) and left offset (LOFF). The Synthesizer Right Offset Final (SROFI) and Synthesizer Left Offset Final (SLOFI) registers contain the final values of ROFF and LOFF. The synthesizer increments or decrements the values in SROI and SLOI by 1 each sample frame until they reach the values contained in SROFI and SLOFI. These sets of registers allow a smooth offset change with only one write operation. Software can cause an instantaneous offset change by writing the same value to both registers (SROI and SROFI, or SLOI and SLOFI).

PAN Mode

When SMSI[5] is zero, bits 11–8 of the Synthesizer Right Offset register (SROI[11:8]) provide a pan value that determines the stereo position of the voice (left offset is not used in this mode). Table 7-7 shows the 16 pan positions and the corresponding left and right offsets. The table values were calculated using Equation 7-8 and Equation 7-9 to keep total power constant in all pan positions.

Table 7-7 Left and Right Amplitudes for PAN Values

Pan Value (SROI[11:8])	Left Offset Value	Left Attenuation (dB)	Right Offset Value	Right Attenuation (dB)
0	0	0	4095	$-\infty$
1	13	–0.31	500	–11.76
2	26	–0.61	372	–8.75
3	41	–0.96	297	–6.98
4	57	–1.34	244	–5.74
5	75	–1.76	203	–4.77
6	94	–2.21	169	–3.97
7	116	–2.73	141	–3.32
8	141	–3.32	116	–2.73
9	169	–3.97	94	–2.21
10	203	–4.77	75	–1.76
11	244	–5.74	57	–1.34
12	297	–6.98	41	–0.96
13	372	–8.75	26	–0.61
14	500	–11.76	13	–0.31
15	4095	$-\infty$	0	0

Effects Volume—EVOL

Whether a voice acts as an effects processor or as a signal generator, a mono effects output signal splits off from the signal path of the voice at one of two places, depending on the state of the *Alternate Effects Path* bit of the Synthesizer Mode Select register (SMSI[4]). If SMSI[4] is Low, the effects output splits off after the VOL(L) and VOL(LFO) attenuators. If SMSI[4] is High, the output splits off before those attenuators. For an illustration of these two paths, see Figure 7-2 on page 7-4.

The effects volume (EVOL) component attenuates the output volume of the effects signal path before the effects signal is fed to the effects accumulators. Two registers control the value of EVOL. The Synthesizer Effects Volume register (SEVI) contains the current value of EVOL and the Synthesizer Effects Volume Final (SEVFI) register contains the final value of EVOL. The value in SEVI is incremented or decremented by 1 each sample frame until it reaches the value contained in SEVFI. This register combination allows a smooth change in effects volume with only one write operation. Software can produce an instantaneous effects volume change by writing the same value to both registers.

A voice's effects output can be sent to any, all, or none of the eight possible effects accumulators as selected in the Synthesizer Effects Output Accumulator Select register (SEASI). For more details, see "Effects Accumulation" on page 7-27.

LFOs for Tremolo and Vibrato

The InterWave IC contains two triangle-wave low-frequency oscillators (LFOs) assigned to each voice. One controls vibrato (frequency modulation) and the other controls tremolo (amplitude modulation). Setting the *Global LFO Enable* bit of the Synthesizer Global Mode register (SGMI[1]) High enables all LFOs.

Note: To use the LFOs, an InterWave IC-based sound board must utilize RAM for local memory. The LFOs do not function in a ROM-only hardware application.

Before an LFO can be used, its operational parameters must be written to local memory by the application software. Then, during operation, the synthesizer reads and writes those parameter values. Software can control the following aspects of LFO operation:

- *Frequency*—the speed at which the volume or pitch is varied
- *Depth*—the amount of the effect
- *Ramp in/out*—the gradual application or removal of an effect

Table 7-8 summarizes the characteristics of the LFOs.

Table 7-8 LFO Characteristics

Characteristic	Requirements or Capabilities
Number of LFOs per voice	2 (one for tremolo and one for vibrato)
Total number of LFOs	64
Local DRAM needed	1K bytes total for 64 LFOs
Register array space needed	64 bytes (2 LFOs x 32 voices x 1 byte per LFO)
LFO update rate	689 Hz
LFO frequency range	21.5 Hz to 95 seconds
Vibrato Maximum Depth (FC=1)	12.4 percent or 215 cents (more than two half-steps)
Vibrato Resolution (FC=1)	0.098 percent or 1.69 cents

Table 7-8 LFO Characteristics (Continued)

Characteristic	Requirements or Capabilities
Tremolo Maximum Depth	12 dB
Tremolo Resolution	.094 dB
LFO ramp update rate	86.13 Hz
Ramp range (for maximum depth)	0.37 to 95 seconds

Addressing the LFO Parameters

The LFO parameters are stored in local memory in a 1024-byte block that contains the data for all 64 LFOs.

Each LFO's data is addressed using the Synthesizer LFO Base Address register (SLFOBI) as the starting point. The most-significant 14 bits of the base address should be written to SFLOBI when memory is allocated for the LFOs (see Chapter 8, "Local Memory Control").

To completely address the individual parameters for a voice, the SFLOBI address is combined with three other bit fields, producing a 24-bit address, as shown in Table 7-9.

Table 7-9 The 24-bit LFO Address

LFO address bits 23:10	bits 9:5	bit 4	bits 3:0
Base Address Register (SLFOBI[13:0])	Voice	V/T	Data Select

The three lower bit fields have the following functions:

Voice The number of the voice associated with the LFO

V/T Selects between the LFO for vibrato (High) and tremolo (Low). This bit completes the selection of a specific LFO.

Data Select Selects the data for the addressed LFO

Using the LFO Parameters

Table 7-10 and Table 7-11 depict the locations of the programmable LFO parameters accessible through SLFOBI, followed by a discussion of the parameters.

Table 7-10 Decoding the Data Select Field

LFO Address Bits 3–0	Name	Width (bits)	Synth. Access	Description
0 0 0 x	CONTROL	16	read	LFO frequency (11 bits) and control bits
0 0 1 0	DEPTHFINAL	8	read	final depth value
0 0 1 1	DEPTHINC	8	read	depth addition (ramp rate)
0 1 x x		n/a		not used
1 0 0 x	TWAVE[0]	16	read/write	LFO current waveform value
1 0 1 x	DEPTH[0]	13	read/write	LFO depth (must write bits 15:13 = 0)
1 1 0 x	TWAVE[1]	16	read/write	LFO current waveform value
1 1 1 x	DEPTH[1]	13	read/write	LFO depth (must write bits 15:13 = 0)

There are two DEPTH values and two TWAVE values per LFO. The *Wave Select* bit (bit 14) of the CONTROL word determines which of the values a particular LFO uses. The Wave Select feature allows an LFO to be modified during operation. For example, while an LFO is using TWAVE[0] and DEPTH[0], software can write new values to TWAVE[1] and DEPTH[1] without concern for the synthesizer overwriting the values. After writing the new values, software can change the *Wave Select* bit to select the new values.

The CONTROL word contains the data shown in Table 7-11.

Table 7-11 Contents of the LFO CONTROL Word

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
LFO Enable	Wave Select	Shift	Invert	Res.	Waveform Increment										

Bit 15: **LFO Enable**. If High, then the LFO is enabled. If Low, then no further accesses take place to process the LFO.

Bit 14: **Wave Select**. Selects between the two sets of waveform and depth values: Low for TWAVE[0] and DEPTH[0], High for TWAVE[1] and DEPTH[1].

Bit 13: **Shift**. If High, shifts the waveform up and to the right so that it starts at 0 and rises to 7FFFh.

Bit 12: **Invert**. If High, flips the waveform about the x axis.

Bit 11: Reserved.

Bits 10–0: **Waveform Increment**. The waveform increment (TWAVEINC) of the LFO. The LFO frequency can range from 21.5 Hz for 7FFh to 95 seconds for 001h. The synthesizer uses Equation 7-11 to determine the LFO frequency.

Equation 7-11 Specifying the LFO Frequency

$$F_{LFO}(Hz) = \frac{44100}{64 \cdot 2^{16}} \cdot TWAVEINC \cong 0.010514 \cdot TWAVEINC$$

LFO Processing

The synthesizer processes the LFO parameters in local memory within the InterWave frame structure to update those parameters and to produce a final LFO value.

Frames and LFO Processing

The synthesizer updates one LFO every frame. Therefore, it takes 64 frames, or one *LFO frame*, to update all of the LFOs.

Every 8 frames, the synthesizer updates the current position for the depth of one LFO. Therefore, during one LFO frame, the synthesizer updates the depth position for 8 LFOs. It takes 8 LFO frames, or one *ramp frame*, to update the depth of all 64 LFOs.

Processing an LFO requires four accesses to local memory if not updating the depth position. The first three accesses read the CONTROL word, the currently selected DEPTH word, and the currently selected TWAVE word. The fourth access writes back a newly calculated TWAVE value.

When updating the current position for the depth, an additional read access obtains DEPTHFINAL and DEPTHINC and an additional write access writes back a new value for DEPTH.

Ramp Updates

For any given LFO, once each ramp frame the synthesizer compares the DEPTH value to DEPTHFINAL * 32. If the two values are equal, no ramping occurs.

If DEPTH is less than DEPTHFINAL * 32, then the synthesizer calculates DEPTH + DEPTHINC and compares that sum to DEPTHFINAL * 32. If the sum is less than DEPTHFINAL * 32, the synthesizer writes the sum to DEPTH. If the sum is greater than DEPTHFINAL * 32, the synthesizer writes DEPTHFINAL * 32 to DEPTH.

If DEPTH is greater than DEPTHFINAL * 32, then the synthesizer calculates DEPTH – DEPTHINC and compares the difference to DEPTHFINAL * 32. If the difference is greater than DEPTHFINAL * 32, the synthesizer writes the difference to DEPTH. If the difference is less than DEPTHFINAL * 32, the synthesizer writes DEPTHFINAL * 32 to DEPTH.

Use Equation 7-12 to calculate the time required for a ramp.

Equation 7-12 Calculating Ramp Time

$$Ramptime = \frac{|DEPTHFINAL \cdot 32 - DEPTH|}{DEPTHINC \cdot 86.13}\text{sec}$$

The Final LFO Value

The final LFO value modifies either the frequency or the volume of the voice. The synthesizer stores the final value in the Synthesizer Frequency LFO register (SFLFOI) for frequency LFOs and the Synthesizer Volume LFO register (SVLFOI) for volume LFOs. The InterWave IC uses the following procedure to create the final LFO value. The action in step 3 depends on the value of the *Shift* bit (bit 13) of the LFO CONTROL word.

Obtains the current waveform position (TWAVE) from local memory.

1. Adds TWAVEINC to TWAVE and writes the result back to TWAVE in local memory.
2. If *Shift* is Low:

   If TWAVE bit 14 is High, it inverts TWAVE bits 13–0 to create the LFO waveform magnitude. Otherwise, it uses TWAVE bits 14–0 as the waveform magnitude. In either case, TWAVE bit 15 is exclusive or'd with the *Invert* bit (bit 12) of the CONTROL word to determine the sign bit (bit 15).

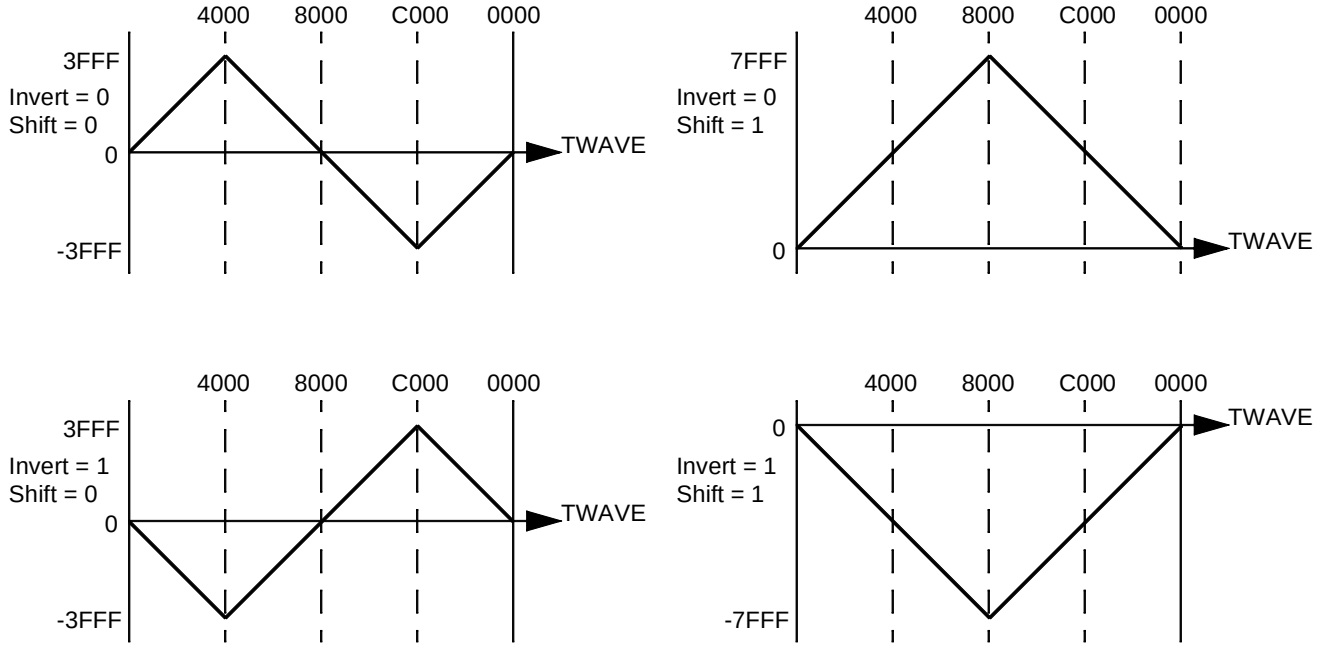
If *Shift* is High:

If TWAVE bit 15 is High, it inverts TWAVE bits 14–0 to create the LFO waveform magnitude. Otherwise, it uses TWAVE bits 14–0 as the waveform magnitude. In either case, the *Invert* bit (bit 12) of the CONTROL word is used as the sign bit (bit 15).

3. Multiplies the 14-bit LFO waveform magnitude value by DEPTH, then combines the 7 most-significant bits of the result with the LFO waveform magnitude sign bit (bit 15) to create the two's-complement final LFO value.
4. Writes the final LFO to the appropriate register.

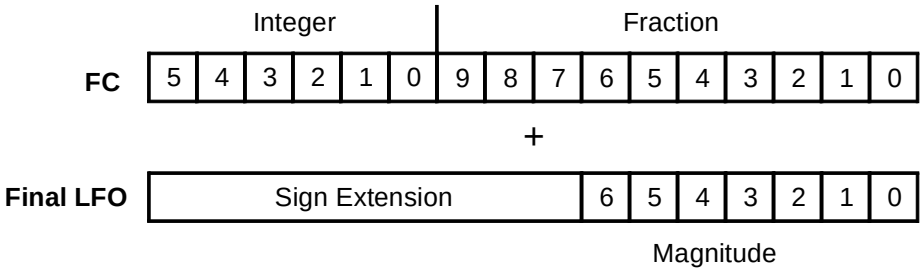
The synthesizer adds TWAVEINC to TWAVE every LFO frame. The magnitude of the LFO waveform is multiplied by the DEPTH to become the final LFO value. The *Shift* and *Invert* bits of the LFO CONTROL word determine which of the four possible waveforms are used. The waveforms are illustrated in Figure 7-10.

Figure 7-10 The Four Possible LFO Waveforms



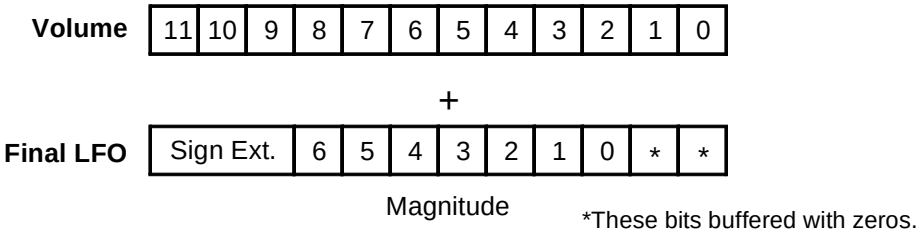
The synthesizer stores the final LFO value as an 8-bit two's-complement value in the appropriate voice-specific register for adding to the frequency or volume controls for the voice. For vibrato, the synthesizer adds the final LFO value to FC as shown in Figure 7-11. This addition allows for a maximum vibrato depth of 12.4 percent (if FC is 1).

Figure 7-11 Adding Final LFO Value to FC—Vibrato



For tremolo, the synthesizer adds the final LFO value to Volume as shown in Figure 7-12. This addition allows for a maximum tremolo depth of 12 dB.

Figure 7-12 Adding Final LFO Value to Volume—Tremolo



Delay-Based Effects

Software can program up to 8 of the 32 synthesizer voices for effects processing, producing such effects as reverb, echo, chorus, and flanging. To specify a voice as an effects processor, set the *Effects Processor Enable* bit of the Synthesizer Mode Select register (SMSI[0]) High. When programmed as an effects processor, the input for the voice comes from the effects accumulator specified in Table 7-12, “Effects Accumulator Output Links,” on page 7-28.

When a voice performs effects processing, it writes the output from one of the effects accumulators to local memory as wavetable data and then reads that data back some number of frame times later, creating a delay. The Synthesizer Effects Address High/Low registers (SEAHI and SEALI) hold the current write address, which loops between the same START and END address boundaries as the read address (only forward looping is allowed). As with all voices, the Synthesizer Address High/Low registers (SAHI and SALI) hold the current read address. The difference between the write and read addresses defines the amount of delay in the effects processor. Each integer delta between the read and write addresses of 1 introduces a time delay of 1 frame, or 1/44,100 seconds—the reciprocal of the frame rate. The distance between the START and END address boundaries defines the maximum delay available. For more details, see “Address Control” on page 7-9.

The write address always increments by 1. The read increments by an average of 1 but can have variations in time added by an LFO. These variations in time generate chorus

and flange effects. The volume components in the left and right path determine how much of the effect is heard and the stereo position of the effects processor's output.

Three bits in the Synthesizer Address Control register (SACI) and the FC value set in the Synthesizer Frequency Control register (SFCI) control the looping mode for write addressing:

- *Loop Enable* bit (SACI[3]) High
- *Bidirectional Loop Enable* bit (SACI[4]) Low
- *Direction* bit (SACI[6]) Low
- FC = 1.0 (SFCI = 0400h)

The mode of looping for read addressing must be the same as for write addressing, plus the *Enable PCM Operation* bit of the Synthesizer Volume Control register (SVCI[2]) must be set high.

The *Alternate Effects Path* bit of the Synthesizer Mode Select register (SMSI[4]) determines which of two split-off points are used for the effects signal path. When SMSI[4] is Low, the effects signal splits off after interpolation and before the VOL(L) and VOL(LFO) attenuators. When SMSI[4] is High, the effects signal splits off after the VOL(L) and VOL(LFO) attenuators. In either case, the effects signal goes through the effects volume attenuator (EVOL) before being fed back to the effects accumulators.

Voice Accumulation

Each active voice sums into the left and right accumulators and into selected effects accumulators once during each frame, so that the accumulators represent the combined activity of all voices in a frame. See Figure 7-1 and Figure 7-2 for diagrams of the accumulation paths.

Signal Voice Accumulation

Summing into the left and right output accumulators takes place automatically—no action is necessary by the programmer. This is the final step before a voice's output is sent to the dedicated synthesizer DAC in the codec module for conversion to analog audio signals.

Effects Accumulation

To send a voice's output through delay-based effects, enable its EVOL output to any or all of the eight effects accumulators. Choose the accumulators in the Synthesizer Effects Output Accumulator Select (SEASI) register. Every voice can direct its effects path to any, all, or none of the effects accumulators.

When a voice acts as an effects processor, its input comes only from the accumulator specified in Table 7-12. For example, if software programs voice 12 to act as an effects processor voice, the voice is linked to effects accumulator 4. Any voice can direct its effects path to be processed by voice 12 by setting its Synthesizer Effects Output Accumulator Select register (SEASI) to 10h, which directs the voice's effects path to effects accumulator 4.

Table 7-12 Effects Accumulator Output Links

Effects Accumulator	Effects Processor Voice Number			
0	0	8	16	24
1	1	9	17	25
2	2	10	18	26
3	3	11	19	27
4	4	12	20	28
5	5	13	21	29
6	6	14	22	30
7	7	15	23	31

Loading Patches

When a program uses DRAM to store wavetable data, musical instrument patches can be swapped in and out as needed.

For details of loading data into local memory, see Chapter 8, “Local Memory Control.”

Digital Audio Files and PCM Operation Mode

To play back a long piece of digitally recorded sound while using only a small block of memory, set the *Enable PCM Operation* bit of the Synthesizer Volume Control register (SVCI[2]) High. PCM operation allows the address control logic to cause an interrupt at an address boundary, but to continue moving the address in the same direction unaffected by the address boundary.

Figure 7-5 illustrates the looping technique for playing back digitally recorded sound. Store the START, END1, and END2 address in the Synthesizer Address Start High/Low registers (SASHI and SASLI) and Synthesizer Address End High/Low registers (SAEHI and SAELI).

For the step-by-step procedure to play back files in PCM mode, see “Playing Digital Audio Files in PCM Operation Mode” on page 7-34.

Effects Digital Signal Processor Interface

Some applications may need to perform audio functions in addition to those provided by the InterWave IC. Two examples of such additional functions are combining the codec with a digital signal processor (DSP) IC to perform high fidelity audio compression or speech recognition, and adding more complex effects processing to the synthesizer. In the first case, the data flows between the DSP and the codec. In the second case, the data flows between the DSP and the synthesizer. Two separate data paths are provided for these applications.

Serial DSP Interface

The InterWave IC provides a serial port that allows data transfers between the codec and an external DSP IC or between the synthesizer and the DSP IC. For details of this interface, see Chapter 6, “Codec/Mixer.”

Effects DSP

The InterWave IC provides a parallel path that allows the output of the effects accumulators to be written as wavetable data to local memory in the normal manner, except that an external DSP intercepts the writes, returning processed data to local memory as wavetable data for the synthesizer. The local memory interface generates timing strobes to facilitate this process.

Since the InterWave local memory controller operates as the timing master, the logic connecting the DSP IC to the local memory bus must provide synchronization for the DSP's internal timing.

GUS Frame Expansion

The synthesizer register set is a superset of that used in the Advanced Gravis UltraSound (GUS) synthesizer. The GUS synthesizer processes up to 14 voices at 44.1 kHz, after which the frame rate slows by 1.6 µs per added voice. Because the InterWave synthesizer runs at 44.1 kHz even when all 32 voices are being used, the GUS-compatible mode allows matching the frame timing to that of the GUS synthesizer.

When GUS-compatibility mode is enabled (*Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) set Low), a 44.1-kHz sampling rate is maintained only for 14 or fewer active voices. If a 15th voice is active, 1.6 µs is added to the sample period, resulting in a sampling rate of 41.2 kHz. This same process continues up to a maximum number of 32 voices, resulting in a minimum sampling rate of 19.4 kHz. Use Equation 7-13 to determine the sampling rate when the InterWave IC is in GUS-compatible mode.

Equation 7-13 GUS-Compatible Sample Period

$$Sample\ period \cong AV \cdot 1.6\,\mu\text{sec}$$

AV is equal to the number of active voices, which is controlled by the Synthesizer Active Voices (SAVI) register. *AV* can range in value from 14 to 32.

When the sampling rate changes, all voice frequency control values must be adjusted to maintain the true pitch of a tone. Slower sampling rates also degrade the audio quality of the tone.

Programming Tips

Software performs synthesizer programming by writing to various registers, which control all aspects of voice processing. The register data bus links the registers to the other modules in the IC.

This section discusses the following fundamentals of programming the synthesizer:

- Accessing voice-specific registers
- Using signal voices to play a note
- Using effects-processor voices
- Playing a digital audio file
- Processing volume envelope segments

Programming Voice-Specific Registers

The synthesizer module has two types of indirect registers: global and voice-specific. Global registers affect the operation of all voices; voice-specific registers affect the operation of only one voice. Access to global registers is identical to other indirect registers. Access to the voice-specific registers requires special considerations.

The Register Array

Voice-specific register values within the synthesizer module reside in a dual-port RAM called the *register array*. One side of the register array is available to the SBI module for voice programming and the other side is available to the synthesizer engine for voice processing.

When the synthesizer begins to generate a voice, the synthesizer module reads the voice's programmed values from the register array. At the end of voice generation, the synthesizer module writes back the self-modifying register values to the register array. When the SBI module attempts to read the register array, it must wait until the synthesizer module is not reading or writing the register array. To speed the read access of the register array, the read indexes of the synthesizer indirect registers are different from the write indexes. The separate read and write indexes allow the InterWave IC to start fetching the data to be read before software actually reads the general data port. In the case of fast bus accesses, the synthesizer uses IOCHRDY during the read of the data port registers to hold the bus until the register array can respond.

In the case of a write to the register array, the write must wait until the synthesizer is not reading or writing any voice and the voice that is being modified by the write is not being processed by the synthesizer module. The second condition insures that data written to a self-modifying register is not changed when the synthesizer module writes to the register array at the end of voice processing. Writes to the register array are buffered. The synthesizer uses IOCHRDY to hold the bus only if the register array has not taken the buffered data before the next write to the Synthesizer Voice Select register (SVSR).

Special care must be taken when writing to paired registers for an active voice. If the synthesizer reads the register array between writes of pairs of voice-specific registers, an unwanted action may be taken by the synthesizer module. The voice-specific register pairs include:

- Synthesizer Address Start High/Low registers (SASHI and SASLI)
- Synthesizer Address End High/Low registers (SAEHI and SAELI)
- Synthesizer Address High/Low registers (SAHI and SALI)
- Synthesizer Effects Address High/Low registers (SEAHI and SEALI)
- Synthesizer Left/Right Offset registers (SLOI and SROI)

Accessing Voice-Specific Registers

To gain access to a voice-specific register, the voice number must be written to the Synthesizer Voice Select register (SVSR). Software triggers a read of a voice-specific register by writing a read address to the General Index register (IGIDXR). When the write to IGIDXR occurs, the InterWave IC starts to fetch the data from the specified index and may actually have the data waiting in the general data port before the port is read. Software triggers a write to a voice-specific register by writing to the General 16-Bit I/O Data Port (I16DP) or the General 8-Bit Data Port (I8DP) after writing the necessary values to SVSR and IGIDXR.

To ease the number of accesses needed to program a voice, set the *Auto Increment* bit of the Synthesizer Voice Select register (SVSR[7]) High to allow the value in IGIDXR to auto-increment with every write to I8DP or I16DP.

These I/O addressing features provide several different ways of accessing voice-specific registers.

Standard Access for Writing and Reading

Use this procedure for writing to or reading from a single register for a specific voice.

1. **Write the voice number to the Synthesizer Voice Select register (SVSR).**
2. **Write the register index value to the General Index register (IGIDXR).**
3. **Write to or read from the General 16-Bit Data Port (I16DP) or the General 8-Bit Data Port.**

Row Access for Writing and Reading

Use this procedure for writing to or reading from several registers for a specific voice.

1. **Write the voice number to the Synthesizer Voice Select register (SVSR).**
2. **Write the register index value to the General Index register (IGIDXR).**
3. **Write to or read from the General 16-Bit Data Port (I16DP) or the General 8-Bit Data Port.**
4. **Repeat steps 2 through 3 for each required register for the selected voice.**

Column Access for Writing

Use this procedure for writing to the same register for several voices.

1. **Write the voice number to the Synthesizer Voice Select register (SVSR).**
2. **Write the register index value to the General Index register (IGIDXR).**
3. **Write to or read from the General 16-Bit Data Port (I16DP) or the General 8-Bit Data Port.**
4. **Write a new voice number to the Synthesizer Voice Select register (SVSR).**
5. **Write to the General 16-Bit Data Port (I16DP) or the General 8-Bit Data Port.**
6. **Repeat steps 4 through 5 for each voice.**

Auto-Increment Access for Writing

Use this procedure for writing to all of the registers for a specific voice.

1. **Write the voice number to the Synthesizer Voice Select register (SVSR).**
2. **Set the Auto Increment bit (SVSR[7]) high.**

   (Steps 1 and 2 can be performed by the same write action.)
3. **Write the register index value for the register with the lowest index value to the General Index register (IGIDXR).**
4. **Write to the General 16-Bit Data Port (I16DP) or the General 8-Bit Data Port.**
5. **Repeat step 4 for each successive register for the selected voice.**

Using Signal Voices

The top section of Figure 7-2 illustrates the data paths for a signal voice.

Repeat the following steps for each active voice:

1. **Decide how many voices are needed to produce the desired sound, and whether its output is to be used for effects.**

2. **Load the wavetable data for the sound into local memory.**

   See Chapter 8, "Local Memory Control."

3. **Write the value of the voice to be programmed to the Synthesizer Voice Select (SVSR) register.**

4. **Write the starting and ending addresses of the data to the Synthesizer Address Start High/Low registers (SASHI and SASLI) and the Synthesizer Address End High/Low registers (SAEHI and SAELI).**

   Software can read wavetable data from local memory in loops, which decreases the amount of memory required for a sustained, unchanging note. Control looping with the Synthesizer Volume Control (SVCI) and Synthesizer Address Control (SACI) registers.

5. **Write the address where the voice should start (i.e., the current address) to the Synthesizer Address High/Low registers (SAHI and SALI).**

   Typically, a voice starts at the beginning of its wavetable data. If this is the case, write the same values to the Synthesizer Address Start registers and to the Synthesizer Address registers.

   Software can also loop through the wavetable data in various ways, allowing, for example, long stretches of repetitive data to be played from only a small block of local memory. See "Address Looping" on page 7-9.

6. **Determine the sampling rate for the voice. Write the appropriate data to the Synthesizer Frequency Control register (SFCI).**

   SFCI determines the rate (FC) at which the wavetable data is accessed. An FC value of 1.0 indicates the data is to be accessed at the same rate as it was recorded.

7. **Add the volume envelope to the voice.**

   For information about processing a volume envelope segment, see "Processing Volume Envelope Segments" on page 7-34.

   The results of the envelope segment are put into the Synthesizer Volume Level register. This is the VOL(L) component referred to in Figure 7-2.

8. **Add tremolo to the voice if desired.**

   The synthesizer adds the tremolo component, VOL(LFO), by continuously varying the value in the Synthesizer Volume Level register (SVLI). The Synthesizer Volume LFO register (SVLFOI) contains the tremolo value, which is the output of a programmable LFO.

9. **Position the voice in the stereo field with PAN, or with LOFF and ROFF.**

   For details, see "Stereo Positioning—Offset and Pan" on page 7-19.

10. If the voice's output is to be used for delay-based effects, set the effects volume (EVOL) in the Synthesizer Effects Volume register (SEVI).

Because the value is played back some delay later by an effects-processor voice and summed with the other voices in that later frame, choose the EVOL level for the amount of the delayed effect to be heard at that later time.

11. Determine the accumulator paths.

Finally, the synthesizer sums a signal voice's output with all other active voices in the frame. There are three voice accumulators:

- Left and Right Accumulators: Summing into these output accumulators takes place automatically—no action is necessary by the programmer. This is the final step before the synthesizer sends a voice's output to the synthesizer DAC in the codec module for conversion to analog audio signals.
- Effects Accumulator: Choose the receiving effects accumulators, if any, in the Synthesizer Effects Output Accumulator Select register (SEASI).

Note: Each effects accumulator's output is hardware-linked to four specific voices, so choose an accumulator linked to a voice known to be available during effects processing. See Figure 7-12 for a map of the links.

12. Flag the voice as the active voice in the Synthesizer Mode Select register (SMSI).

Using Effects-Processor Voices

To use a voice as an effects processor, use the following procedure:

1. **Determine in advance which voices are to be used as effects processors.**

   The outputs of the effects accumulators are linked in hardware to specific voices. See Table 7-12 for the mapping.

2. **Clear the local memory locations to be used for the effects processing.**

   See Chapter 8, "Local Memory Control."

3. **Flag the voice as an effects processor by setting the *Effects Processor Enable* bit in the Synthesizer Mode Select register (SMSI[0]).**

4. **Select the effects signal path by setting the *Alternate Effects Path* bit (SMSI[4]).**

   This step determines which volume components affect the effects signal path—see Figure 7-2. Steps 3 and 4 can be performed by the same write operation.

5. **Write the starting and ending addresses of the voice's local memory area to the Synthesizer Address Start High/Low registers (SASHI and SASLI) and the Synthesizer Address End High/Low registers (SAEHI and SAELI).**

1. **Write the address where the voice should start (i.e., the current address) to the Synthesizer Address High/Low registers (SAHI and SALI).**

   As with a signal voice, this address is often the same value that is in the Synthesizer Address Start High/Low registers (SASHI and SASLI).

2. **Determine the amount of delay to be added by this effects processor.**

   Determine the difference between the read address and the write address to achieve the desired delay (each increment equals 1/44,100 seconds). Write the higher address (the write address) to the Synthesizer Effects Address High/Low registers (SEAHI and SEALI).

3. **Determine how much of the effect is to be heard.**

   Volume segments, tremolo, and panning are added just as with a signal voice. This path, with full envelope generation, sums into the left and right accumulators with all other active voices in the frame.

4. **Set the effects volume (EVOL) to be fed back to the effects accumulators.**

   As with a signal voice, set EVOL with the Synthesizer Effects Volume (SEVI) register. The value used in this step allows an effect to build up or decay with time.

5. **Flag the voice as the active voice in the Synthesizer Voice Select (SVSR) register.**

Playing Digital Audio Files in PCM Operation Mode

Software can use any or all of the 32 synthesizer voices to play back and mix multiple digital audio files, such as **.wav** files.

For more background on playing back files in this mode, see “Digital Audio Files and PCM Operation Mode” on page 7-28. For a graph of the addressing used in PCM operation, see Figure 7-5.

The following steps illustrate playing back an audio file:

1. **Using DMA or Programmed I/O (PIO), store the first block of recorded data in local memory from address START to END1.**
2. **Set START and END1 as address boundaries with SVCI[2]=1, SACI[3]=0, SACI[4]=0, and SACI[6]=0 and start the voice.**
3. **Using DMA or PIO, store the next block of recorded data in local memory from address END1 to END2.**
4. **When the voice causes an interrupt for crossing END1, change the address boundary from END1 to END2 and set SACI[3]=1.**
5. **Using DMA or PIO, store the next block of recorded data in local memory from address START to END1.**
6. **When the voice causes an interrupt for crossing END2, change the address boundary from END2 to END1 and set SACI[3]=0.**
7. **Repeat steps 3 through 6 until the recorded data has completed playing.**

Processing Volume Envelope Segments

To set up the initial envelope segment (typically done from the main program), use the following procedure. This procedure assumes the *Stop 1* bit (SVCI[1]) is already set to 1.

1. **If the target volume is greater than the current volume, set the end volume equal to the target volume.**
   If the target volume is less than or equal to the current volume, set the start volume equal to the target volume.

2. **Set the volume rate and increment.**

3. **Set SVCI as follows:**

Volume IRQ	(bit 7) = 0
Direction	(bit 6) = 0, if target volume > current volume = 1, if target volume <= current volume
Volume IRQ Enable	(bit 5) = 1
Bidirectional Loop Enable	(bit 4) = don't care
Loop Enable	(bit 3) = 0
Enable PCM Operation	(bit 2) = do not change
Stop 1	(bit 1) = 0
Stop 0	(bit 0) = 0

For each subsequent volume envelope segment, use the following procedure (step 2 through step 7 are typically done from a volume interrupt handler):

1. **Poll SVIRI to determine if a volume interrupt for this voice has been triggered. If so, proceed with step 2.**

2. **Follow step 2 through step 4 for an initial segment.**

3. **Read SVII to clear the interrupt.**

CHAPTER

8 LOCAL MEMORY CONTROL

The InterWave local memory control module (also called the local memory controller or LMC) transfers data between local memory and the synthesizer, system bus interface (SBI), or codec module. Local memory can include DRAM and ROM.

This chapter covers the following topics:

- Local memory control basics
- Data paths
- Register overview
- Initialization
- Interrupts
- Local memory configuration
- Accessing local memory
- DMA data transfers
- Local memory record and playback FIFOs
- Programming tips and examples

Local Memory Control Basics

The InterWave IC supports up to 16 Mbytes of DRAM and 16 Mbytes of ROM. InterWave local memory control provides the following capabilities:

- Specifying the type and size of local memory, as described in “Local Memory Configuration” on page 8-6
- Storing of wavetable information in local memory for use with the InterWave synthesizer
- Storing of digitized special effects information from the synthesizer’s digital signal processor (DSP) interface
- Establishing and controlling large FIFOs in local memory for record and playback functions
- Supporting DMA transfers to and from system memory for wave file recording and playback, and for accessing the external device (e.g., CD-ROM)

As with the other functional areas of the InterWave IC, programmable registers provide configuration of and access to local memory. The “Register Overview” on page 8-2 outlines the programmable register functions of the local memory control module.

Local Memory Access

A priority encoder and state machine determine which of the possible sources of local memory cycles are granted access to local memory (detailed in “Accessing Local Memory” on page 8-8). The possible sources are the synthesizer, the system bus interface (SBI),

and the codec, with the synthesizer as the highest-priority user. The priority encoder also controls the DRAM refreshing, both during normal and suspend-mode operation.

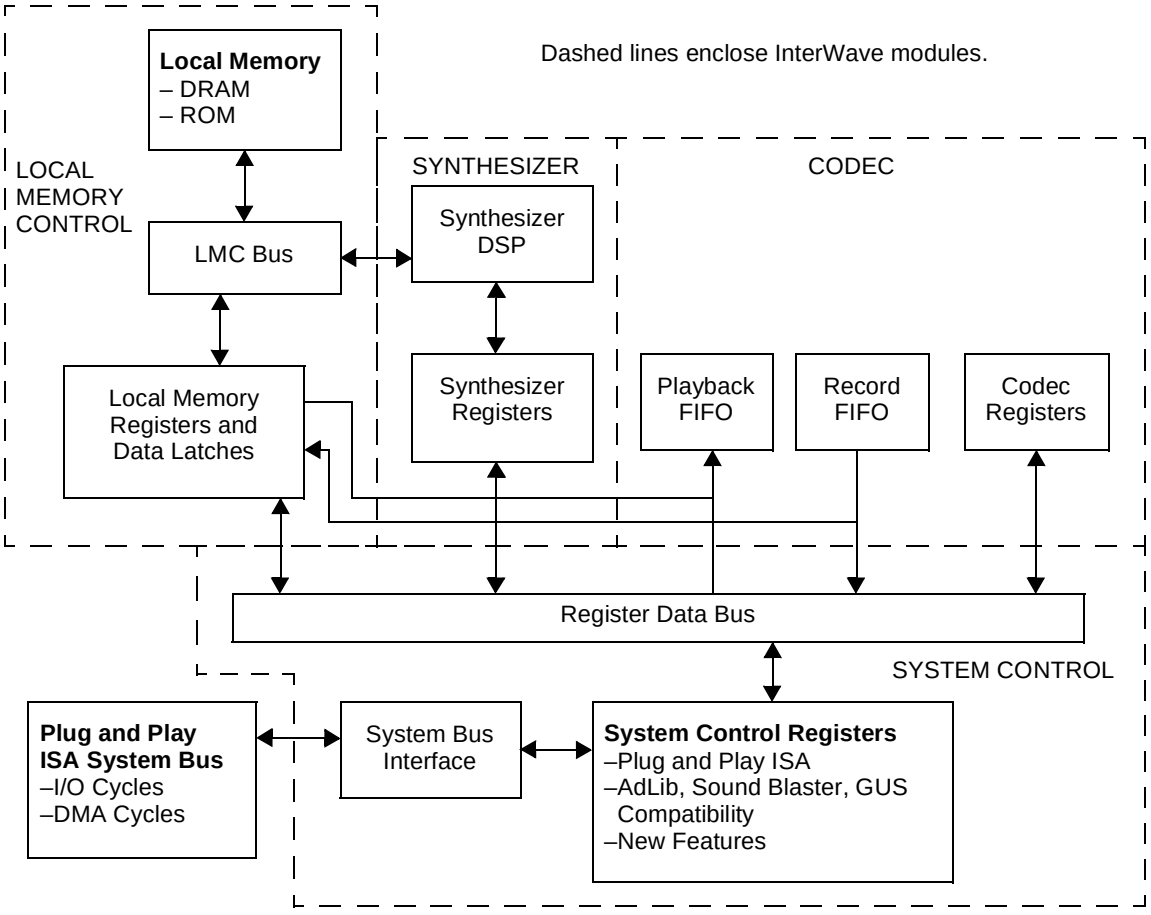
Frame-Expansion

Frame-expansion mode allows compatibility with the UltraSound sound board (GUS) by adjusting the memory-access frame rate when more than 14 synthesizer voices are active at the same time. For more information, see “GUS Frame Expansion” on page 7-29.

Data Paths

Figure 8-1 shows the data flow between local memory DRAM and ROM, the local memory controller, the system bus, and the other InterWave modules.

Figure 8-1 Local Memory Control Data Paths



Register Overview

The following tables outline the local memory control functions available through the InterWave programmable registers. For detailed information about the registers, see the reference page listed in the last column of each table. These references point to the applicable section in Chapter 15, “Local Memory Control Registers.” For information about how to program the registers, see “Accessing InterWave Registers” on page 4-2.

Table 8-1 through Table 8-2 groups the local memory control functions into the following areas:

- Control functions
- DMA and IRQ functions

Table 8-1 Local Memory Control Functions

Function	Register and Bit Field	Reference
Set the I/O transfer local memory address pointer	LMAHI, LMALI	15-3
Read a 16-bit value from local memory (I/O transfer)	LMSBAI	15-4
Set the suspend mode DRAM refresh rate	LMCFI[11:10]	15-4
Set the normal mode DRAM refresh rate	LMCFI[9:8]	15-4
Specify the ROM configuration	LMCFI[7:5]	15-4
Specify the DRAM configuration	LMCFI[3:0]	15-4
Invert the most significant bit of I/O transfer data from system to local memory	LMCI[3]	15-5
Set data width for system to local memory I/O transfer to 16 bits	LMCI[2]	15-5
Select DRAM or ROM for I/O transfer	LMCI[1]	15-5
Specify auto-increment mode for I/O transfers	LMCI[0]	15-5
Specify the record FIFO base local memory address	LMRFAI	15-6
Specify the playback FIFO base local memory address	LMPFAI	15-6
Enable the local memory record FIFO (LMRF)	LMFSI[12]	15-6
Specify the local memory record FIFO (LMRF) size	LMFSI[11:8]	15-6
Enable the local memory playback FIFO (LMPF)	LMFSI[4]	15-6
Specify the local memory playback FIFO (LMPF) size	LMFSI[3:0]	15-6

Table 8-2 Local Memory DMA and IRQ Functions

Function	Register and Bit Field	Reference
Invert the most significant bit of GUS-compatible DMA data from system to local memory	LDMACI[7]	15-1
Read status of DMA terminal count (TC) IRQ	LDMACI[6]	15-1
Set data width to 16 bits for GUS-compatible DMA between system and local memory	LDMACI[6]	15-1
Enable DMA TC IRQ	LDMACI[5]	15-1
Select DMA transfer rate	LDMACI[4:3]	15-1
Determine data width of DMA transfers between system and local memory	LDMACI[2] or LMCI[6]	15-1, 15-5
Select direction of DMA transfers between system and local memory	LDMACI[1]	15-1
Enable GUS-compatible DMA transfers	LDMACI[0]	15-1
Set the GUS-compatible DMA transfer local memory address counter	LDSAHI[7:4], LDSALI, LDSAHI[3:0]	15-2
Invert the most significant bit of interleaved DMA data from system to local memory	LDICI[10]	15-7
Enable interleaved DMA transfer between system and local memory	LDICI[9]	15-7
Set data width to 16 bits for interleaved DMA between system and local memory	LDICI[8]	15-7
Specify the number of interleaved DMA tracks	LDICI[7:3]	15-7
Specify the size on interleaved DMA tracks	LDICI[2:0]	15-7
Specify the base local memory address for interleaved DMA transfers	LDIBI	15-7

Initialization

This section discusses what steps software must take to prepare the local memory control module for use. After the InterWave IC is reset, the local memory controller begins operation with its registers in their default states. First, software must configure the InterWave IC for Plug and Play (PNP) operation, which is discussed in Chapter 5, “System Control Functions.” Next, software may need to set some of the local memory registers with non-default values. Some settings, such as DRAM and ROM bank configuration, need to be set only once, if at all. These configuration settings are typically set by a driver or initialization program.

Other settings, such as auto incrementing of the local memory address and whether to use ROM or DRAM, can change, possibly several times, during the progress of an application program.

For more information on the LMC module’s power-up default modes, see Chapter 15, “Local Memory Control Registers.”

What to Initialize

The local memory initialization process should check the default reset state of every register being used, changing the register settings where necessary. The following settings are important in every program.

Refresh Timing

Specify refresh rates in the LMC Configuration register (LMCFI).

Normal Operating Mode

The default refresh rate for DRAM is 15 µs. Change this value only if the DRAM used permits longer refresh periods.

Suspend Mode

The default is no refresh during suspend mode.

For complete details on the refresh modes available, see “DRAM Refresh Rates” on page 8-7.

Memory Configuration

Specify the sizes of the four DRAM and four ROM memory banks in the LMC Configuration register (LMCFI). If necessary, change the settings to match the local memory hardware being used.

DRAM The default is one bank of 256 Kbytes.

ROM The default is four banks, 128K by 16 bits each.

See “Local Memory Configuration” on page 8-6 for complete details.

Address Registers

The InterWave IC sets all memory address registers to zero on power up or reset. For this reason, always write the starting and ending addresses to the appropriate registers for the operation being programmed. Table 8-1 and Table 8-2 summarize the address registers used by the LMC module.

Follow this procedure with registers in all modules. Thus, when the LMC provides data for a synthesizer operation, the applicable operation-specific address registers are:

- Synthesizer Address Start High (SASHI)
- Synthesizer Address Start Low (SASLI)
- Synthesizer Address End High (SAEHI)
- Synthesizer Address End Low (SAELI)

Address Auto-Increment Mode

Software can set the local-memory address to increment automatically after each access through the LMC Byte Data (LMBDR) and LMC 16-Bit Access (LMSBAI) registers. Set auto-increment in the LMC Control (LMCI) register.

The **IwavePokeBlock** and **IwavePeekBlock** DDK functions turn on auto-increment mode to read or write entire blocks of data to or from local memory.

Returning from Suspend Mode

If the InterWave IC might be placed in suspend mode while the program is running, initialize the suspend mode refresh rate (the default is no refresh during suspend mode).

When ordered to suspend, the LMC completes the current frame. Then, it leaves the memory-access logic in a state that does not interfere with refresh cycles while suspended. When suspend is released, operation resumes automatically with the next frame.

It is not possible to access the InterWave IC from the ISA bus while the hardware is in suspend mode. Therefore, software must delay access to the IC by about 10 ms after returning from suspend mode.

Interrupts

The LMC module does not generate or process interrupts. For an overview of the InterWave interrupt structure, see “Interrupt Structure” on page 4-8.

Local Memory Configuration

The InterWave LMC can address up to 16 Mbytes of local memory in four banks of 4 Mbytes each. Local memory can consist of DRAM, ROM, or a combination of both.

This capacity allows the storage of large amounts of sound data (i.e., wavetable data). It greatly reduces the usage of system RAM and alleviates the need for repeated, time-consuming file I/O transfers.

The InterWave IC does not detect the amount or type of memory attached to it. Thus, the memory configuration must be loaded into the LMC Configuration Register (LMCFI) by vendor software *during initialization*. Different fields within this register specify DRAM and ROM configurations.

DRAM Banks

Table 8-3 lists the valid configurations for DRAM. Specify the configuration in the *DRAM Configuration* field of the LMC Configuration register (LMCFI[3:0]). The default value of LMCFI is 0000h and therefore the IC powers up configured for one bank of 256 Kbytes of DRAM. For memory configurations other than the default, the correct values must be written to LMCFI[3:0].

The InterWave DDK provides two functions that assist in determining and setting the memory configuration for DRAM:

- **IwaveMemSize**—returns the size of local DRAM, in kilobytes
- **IwaveMemCfg**—determines the DRAM configuration and sets up LMCFI accordingly

These functions work only with DRAM. ROM must always be configured explicitly in LMCFI.

Note: Vendors of InterWave-based sound hardware may have to incorporate some form of IwaveMemCfg into device drivers or initialization programs because they may support more than one possible memory configuration in their hardware.

Table 8-3 DRAM Bank Configurations (values are in bytes)

LMCFI[3:0]	Bank 3	Bank 2	Bank 1	Bank 0	Total
0	—	—	—	256K	256K
1	—	—	256K	256K	512K
2	256K	256K	256K	256K	1M
3	—	—	1M	256K	1.25M
4	1M	1M	1M	256K	3.25M
5	—	1M	256K	256K	1.5M
6	1M	1M	256K	256K	2.5M
7	—	—	—	1M	1M
8	—	—	1M	1M	2M
9	1M	1M	1M	1M	4M
10	—	—	—	4M	4M
11	—	—	4M	4M	8M
12	4M	4M	4M	4M	16M
13–15	reserved				

DRAM Refresh Rates

Set the DRAM refresh rates in the *Suspend Mode Refresh Rate* and *Normal Mode Refresh Rate* fields of the LMC Configuration register (LMCFI[11:10] and LMCFI[9:8]). Table 8-4 lists the refresh rates available during suspend and normal modes. The DRAM refresh rate should be set by a driver or initialization program.

Table 8-4 DRAM Refresh Rates

Value	LMCFI[11:10]—Suspend Mode	LMCFI[9:8]—Normal Mode
0 0	No refresh	15 µs refresh rate
0 1	62 µs refresh rate	62 µs refresh rate
1 0	125 µs refresh rate	125 µs refresh rate
1 1	Self-timed refresh	No refresh

ROM Banks

If present, each of the four 16-bit-wide banks of ROM must be the same size.

Table 8-5 lists the allowable configurations for ROM. Set the configuration in the *ROM Configuration* field of the LMC Configuration register (LMCFI[7:5]).

Table 8-5 ROM Bank Configurations (values in bits)

LMCFI[7:5]	Bank Size	Possible Total
0	128K x 16	512K x 16
1	256K x 16	1M x16
2	512K x 16	2M x 16
3	1M x 16	4M x 16
4	2M x 16	8M x 16
5–7	reserved	

Accessing Local Memory

This section explains several important aspects of accessing local memory.

Address Translation

Table 8-6 shows how the InterWave IC translates address values written to the bit fields of the various synthesizer and local memory address registers (logical addresses, indicated by A[23:0]) into actual, or real, local memory addresses (indicated by RLA[23:0]). For more information, see the appropriate sections in Part , "InterWave Registers Reference."

Table 8-6 Local Memory Address Translations

Enhanced Mode Bit (SGMI[0])	Access Width	SUAI, SASHI, SASLI, SAHI, SALI, SAEHI, SAELI, SEAHI, SEALI, LDSALI, LDSAHI	LDIBI, LMRFAI, LMPFAI, LMALI, LMAHI, SLFOBI
0	8-bit	RLA[23:0]=(0,0,0,0,A[19:0]	RLA[23:0]=A[23:0]
0	16-bit	RLA[23:0]=(0,0,0,0,A[19:18],(A[16:0]*2))	RLA[23:0]=A[23:0]
1	8-bit	RLA[23:0]=A[23:0]	RLA[23:0]=A[23:0]
1	16-bit	RLA[23:0]=(A[22:0]*2)	RLA[23:0]=A[23:0]

Table 8-6 clearly indicates which registers are affected by the address translations that take place within the InterWave IC. Accesses to local memory are carried out either by the synthesizer or by the software through DMA or programmed I/O transfers. For DMA transfers, the selected DMA channel determines the access width. DMA channels 4, 5, 6, or 7 carry out 16-bit accesses. All other available DMA channels use 8-bit access. For synthesizer accesses, SACI[2] determines the access width. When accessing local memory through programmed I/O (using the LMAHI and LMALI address registers), no translations are necessary.

Note: It is imperative to observe the address translations shown in Table 8-6. Software must always perform a translation opposite to that done by the InterWave IC for 16-bit accesses.

The **IwaveAddrTrans** DDK function provides details for obtaining the correct logical addresses to be written to the registers for 16-bit accesses. I/O memory-transfer cycles are not affected.

Programmed I/O Cycles to Local Memory

Programmed I/O cycles to local DRAM are independent of data type and mode of operation; no address translations are required. The LMC maps the address specified in the LMAHI and LMALI registers directly to that local memory address.

Downloading or uploading blocks of data through I/O cycles is straightforward with the **IwavePokeBlock** and **IwavePeekBlock** DDK functions for bytes or **IwavePokeBlockW** and **IwavePeekBlockW** for words (16 bits). These functions use the auto-increment mode set in the *Auto Increment* bit of the LMC Control register (LMCI[0]) for efficient data transfers.

16-Bit Synthesizer Transfers

The InterWave IC can operate in either GUS-compatibility or enhanced mode. The 16-bit synthesizer accesses to DRAM must take the internal address translations into account, as follows:

- **GUS-compatibility mode**
  Only address bits 19–0 are used, allowing access to only 1 Mbyte of local memory. Table 8-6 shows that for a 16-bit synthesizer access, bits 16–0 written to the address registers are shifted left by one, bit 17 is dropped, and bits 19–18 remain unchanged. Therefore, software must carry out the inverse translation before writing the address to any register that points to a location in local memory. For instance, if location 601E0h is to be accessed, the software must write 500F0h or 700F0h to the address registers (both values translate to the same memory location). Because GUS-compatibility mode shifts bits16–0 and drops bit 17, transfers cannot cross 256-Kbyte boundaries.
- **Enhanced mode**
  The IC performs a simple left shift of address bits 22–0. This shift must be anticipated by software before writing address values to any register that points to a location in local memory.

As a result of the address translation performed by the LMC, 16-bit accesses to DRAM are always aligned to an even byte.

DMA transfers in GUS-Compatible Mode

For 16-bit accesses of local memory, the application software should specify only address bits 19–4 when writing to the LMC DMA Start Address Low register (LDSALI), because bits 3–0 are set to 0000 binary automatically. Therefore, if the application needs to start a transfer at location 601E0h, it should write 500Fh to LDSALI. The LMC translates this value as shown in Table 8-6.

The address translation imposes the following restrictions:

- 16-bit DMA transfers must always start at an address aligned to a 32-byte boundary.
- 8-bit DMA transfers always start at an address aligned to a 16-byte boundary.

8-bit DMA transfers require no address translation. For example, to start a transfer at 601E0h, just write 601Eh to the LMC DMA Start Address register (LDSALI). The four least significant bits of the address are automatically set to 0000 binary.

In GUS-compatibility mode, a 16-bit DMA transfer can not cross a 256-Kbyte boundary because of the address translation mechanism.

DMA Transfers in Enhanced Mode

In enhanced mode, the address translation of the register address values requires software to shift the desired DMA starting address right by one bit before writing the address to the registers. The address translation forces 16-bit DMA transfers to be aligned to even byte boundaries. For example, to start a transfer at 0601E0h, load 0300F0h into the LMC DMA Start Address Low (LDSALI) and LMC DMA Start Address High (LDSAHI) registers.

8-bit transfers are byte aligned and require no reverse translation by the software. Therefore, to start a transfer at 060130h, load that value without change into LDSALI and LDSAHI.

The size of a DMA transfer in enhanced mode is limited only by the amount of DRAM present. A 24-bit address can access up to 16M one-byte data samples for 8-bit transfers and up to 8M two-byte (word) data samples for 16-bit transfers.

The **IwaveDmaXfer** DDK function details all the steps necessary to perform DMA transfers.

Local Memory Management

Management of local memory differs between GUS-compatibility and enhanced modes.

GUS-Compatible Mode

In the GUS-compatibility mode, the maximum amount of local memory available to an application is 1 Mbyte. Because 16-bit accesses are restricted to 256K boundaries, the LMC breaks the available local memory into, at most, four pools of 256K each. For a demonstration of local memory management in GUS-compatibility mode, examine the DDK local memory functions. The DDK routines always allocate blocks of memory that are aligned to 32-bit boundaries. This alignment serves for both 8-bit and 16-bit DMA transfers, because 8-bit transfers require 16-byte alignment and 16-bit DMA transfers require 32-byte alignment. An address aligned to a 32-bit boundary is also aligned to a 16-bit boundary.

Enhanced Mode

In the enhanced mode, all installed local memory, up to 16 Mbytes, is available to an application. Local memory can be managed in a single pool because the limitations imposed by address translations in GUS-compatibility mode do not apply in enhanced mode.

DDK Local Memory Management Functions

The DDK memory-management functions always return addresses aligned to even bytes, and always round sizes up to the next even byte. See the following DDK functions:

- **IwaveMemAlloc**—allocate memory
- **IwaveMemFree**—deallocate memory
- **IwaveMaxAlloc**—determine the maximum allocatable block size

Memory-Access Priorities

The InterWave IC does not track local memory usage. The DDK provides a set of memory-management functions that keep track of memory usage by an application.

The LMC generates access cycles to local memory based on a priority mechanism that arbitrates between the various requests for memory access. Software cannot control these priorities. The synthesizer's memory access requests have high priority and processing many active voices may slow other memory accesses (i.e., codec record and playback, system bus I/O). For details, see "Accessing Local Memory" on page 8-8.

The prioritizing of memory access requests is different depending upon which cycle type —SYNTH, EVEN, or ODD—is being executed. The LMC grants access requests as shown in Table 8-7.

For more information on memory access cycles and priorities, see the *InterWave IC Hardware Designer's Guide* available from AMD.

Table 8-7 Priorities of Access Cycles

Priority	SYNTH	EVEN	ODD
1	Synth patch access	Effects access	Synth LFO access
2	Refresh request	Codec playback FIFO	DMA cycle
3	DMA cycle	Codec record FIFO	SBI I/O cycle
4	SBI I/O cycle	Refresh request	Codec playback FIFO
5	Codec playback FIFO	DMA cycle	Codec record FIFO
6	Codec record FIFO	SBI I/O cycle	Refresh request

DMA Data Transfers

The InterWave IC is capable of two kinds of DMA transfer: normal and interleaved.

Normal Mode

Specify a DMA transfer between local memory and system memory with the LMC DMA Control register (LDMACI). Specify the beginning address of the transfer in local memory in the LMC DMA Start Address High (LDSAHI) and LMC DMA Start Address Low (LMDSALI) registers.

The DMA request signal generated by the LMC goes to the DMA logic described in Chapter 5, "System Control Functions" to become a DMA request signal out to the ISA bus. Similarly, the DMA logic receives the DMA acknowledge signal from the ISA bus and passes it to the LMC.

Interleaved Mode

Figure 8-2 shows how it is possible to separate data into tracks during an interleaved DMA transfer into local memory. Assume that *n* tracks of interleaved audio data are stored in system memory, where *n* can be from 1 to 32, as programmed in the *Number of Interleaved Tracks* field of the LMC DMA Interleave Control register (LDICI[7:3]). Specify the size of each track in the *Size of Interleaved Tracks* field (LDICI[2:0]), where the number of bytes in each track is 2^(9 + LDICI[2:0]) (ranging from 512 to 64K). The way in which data is transferred varies, based on the DMA channel width and the sample width, as shown in Table 8-8.

Figure 8-2 DMA Data Interleaving

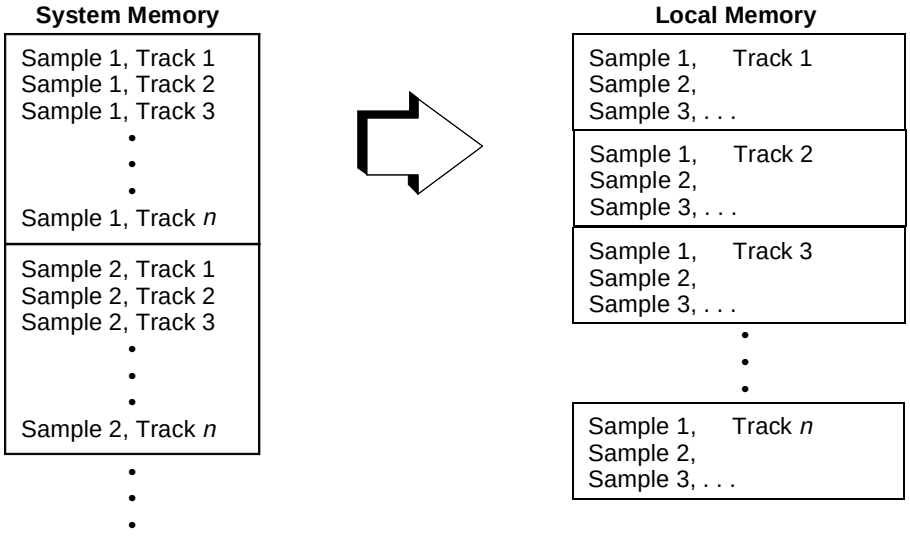
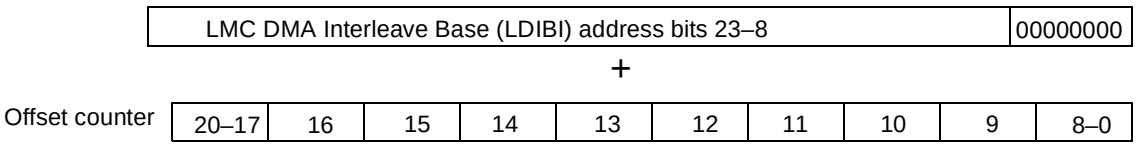


Table 8-8 Interleaved DMA Transfer Modes

DMA Channel	Sample Size	Description
8-bit	8-bit	Each DMA request-acknowledge cycle transfers one byte that is placed in the current track number. The track number increments with each byte transferred.
8-bit	16-bit	Each DMA request-acknowledge cycle transfers two bytes that are placed at the current track number. The track number increments with each 16-bit value transferred.
16-bit	8-bit	Each DMA request-acknowledge cycle transfers two bytes. The lower byte is placed in the current track number, the track number is incremented, and the upper byte is placed in the new track number. The track number then increments again.
16-bit	16-bit	Each DMA request-acknowledge cycle transfers one 16-bit value that is placed in the current track number. The track number increments with each 16-bit value transferred.

The local memory address for the interleaved DMA function is implemented by ORing the LMC DMA Interleave Base register (LDIBI) and an offset counter, as shown in Figure 8-3. The address generated is real regardless of the DMA channel width. It points to a byte in local DRAM.

Figure 8-3 Interleaved DMA Address Generation



The LMC clears each offset counter with each write to LDIBI. The most significant bits, starting at the bit defined by the *Size of Interleaved Tracks* field of the LMC DMA Interleave Control register (LDICI[2:0]), increment with each sample transferred. After transferring the number of samples specified by the *Number of Interleaved Tracks* field (LDICI[7:3]), the LMC increments the least significant bits and clears the most significant bits. If LDICI[7:3] is set to zero, then this DMA function operates like a single track transfer with a roll-over point specified by LDICI[2:0].

To execute an interleaved DMA transfer, do the following steps:

Specify the number and the size of the tracks, the data width, and whether to invert the most significant bit of each sample with a single I/O write to the LMC DMA Interleave Control register (LDICI).

1. Write bits 23–8 of the base address for the transfer to the LMC Interleave Base Address register (LDIBI). Notice that the 8 lower bits of this address are zero.
2. Trigger the DMA transfer by setting the *Interleaved DMA Enable* bit (LDICI[9]) High. The LMC automatically clears this bit after the transfer is complete.

The **IwaveDmaIleaved** DDK function implements these steps.

Local Memory Record and Playback FIFOs

The local memory record and playback FIFOs (LMRF and LMPF) are large first-in-first-out buffers allocated in local DRAM. Use the LMPF to automatically transfer data from DRAM to the codec playback FIFO. Use the LMRF to automatically transfer data from the codec record FIFO to local DRAM. In addition, software can move data from the synthesizer, through the LMRF and out to the system bus interface (SBI). The local memory FIFO sizes can range from 8 bytes to 256 Kbytes. The CPU writes the LMPF data to DRAM and reads the LMRF data from DRAM through normal I/O accesses.

Programming Tips and Examples

This section provides information about how to accomplish the following LMC tasks in an application program:

- Configuring local memory
- Transferring data using I/O cycles
- Transferring data from playback FIFO to codec using DMA
- Transferring data between system and local memory using DMA

Configuring Local Memory

To ensure proper access to local memory, the software must tell the InterWave IC how much local memory is installed and how that memory is distributed across all four banks. Normally, this configuration process occurs once during device initialization performed by a device driver and the IC retains the configuration until reset or power down time.

To configure local memory according to the options listed in Table 15-2, “DRAM Configuration Selection,” on page 15-5, the DDK provides the **IwaveMemCfg** function. This function performs the following three steps:

Determines the size of local memory in a non-destructive fashion (the contents of local memory is left intact) and stores the size in kilobytes in the `size_mem` member of the global `iw` variable.

3. Determines the amount of memory present within each of the four memory banks and from that information determines the configuration as indicated in Table 15-2.

4. Updates the contents of the *DRAM Configuration* field of the LMC Configuration register (LMCFI[3:0]).

IwaveMemCfg determines the configuration of DRAM attached to the InterWave. Because ROM can not be probed reliably like DRAM, the configuration of ROM must be hard coded into the initialization device driver for the particular sound card.

The **IwaveMemSize** function performs the same sizing of local memory as **IwaveMemCfg**, but it does not update LMCFI.

For more information about local memory configuration, see “Initialization” on page 8-4.

Transferring Data Using I/O Cycles

The following steps describe how to use programmed I/O to transfer a *patch* from the system bus to local memory. In this example, the patch is to be used by the synthesizer.

1. **Set Auto Increment in the LMC Control register (LMCI[0]=1).**

   This setting causes the address in the LMC Address registers to increment automatically after every access.

2. **Write the starting local memory address to the LMC Address High (LMAHI) and LMC Address Low (LMALI) registers.**

3. **Write the series of data bytes to the LMC Byte Data register (LMBDR) or 16-bit words to the LMC 16-Bit Access register (LMSBAI).**

   If using 8-bit access through LMBDR, the address in the registers increments by one after each write, leaving the LMC pointing to the next address and ready for the next data byte into LMBDR.

   If using 16-bit access through LMSBAI, the address in the registers increments by two after each write.

   The **IwavePokeBlock** and **IwavePeekBlock** DDK functions can greatly simplify downloading and uploading data with programmed I/O cycles.

4. **Write the beginning address of the loaded data to the Synthesizer Address Start High (SASHI) and Synthesizer Address Start Low (SASLI) registers.**

   See Chapter 7, “Synthesizer,” for details.

Transferring Data Between System and Local Memory Using DMA

When performing DMA transfers between the system memory and local memory, the software must take into account the address translations that occur before the programmed addresses become *real* local-memory addresses. As shown in Table 8-6 on page 8-8, these translations occur for 16-bit data access. The data-access width is determined by the DMA channel in use, whether in enhanced mode or GUS-compatibility mode. The DDK **IwaveAddrTrans** function performs this address translation.

An application that uses DMA transfers to move data to and from local memory must first define a structure of type DMA to hold information about the DMA channel to be used. After the structure has been defined, it must be registered so that DDK functions can access it. If the application needs to do any special processing when the DMA terminal count interrupt

occurs, the application must also register an IRQ structure and a callback. The **IwaveSetInterface** function registers DMA and IRQ structures. The **IwaveSetCallback** function registers callbacks.

The process of registering a DMA or IRQ structure consists of setting a pointer to the particular structure inside the iw variable, which is defined in the **iwcore.h** DDK header file. This iw variable is required in all DDK-based application programs.

The application must reserve an adequate amount of system memory to write to (uploading) or read from (downloading) during the transfer. This RAM must be reserved in DOS conventional memory as the DMA controller requires it. If the application is a protected mode program then it must make special arrangements to reserve conventional memory.

The program in Sample 8-1 illustrates the execution of a DMA transfer using the DDK **IwaveDmaXfer** function, as well as several aspects of any program using the DDK. The program downloads 16 bytes of data to InterWave local memory starting at location 00h. The program registers a callback function called **DmaCallback** whose only purpose is to set a flag indicating the completion of the transfer.

The program in Sample 8-1 performs the following steps:

1. **Defines a callback function (DmaCallback).**

   It is not necessary for an application to have a callback but one is shown here as an example. The callback function is called by the actual interrupt handler when the transfer is completed. The callback in this program only sets a flag when called; however, a callback can be defined by the application to perform any necessary task.

2. **Opens the DDK by issuing a call to IwaveOpen.**

   This important step *must* be carried out before any communication with the sound card is attempted.

3. **Registers a DMA structure and an IRQ structure by calling IwaveInitStructs.**

   After initializing the DDK and InterWave board, the program registers the DMA as well as the IRQ structures through a call to the **IwaveSetInterface** function. This call is the only means by which the DDK knows about these structures.

   The following items must be specified to execute a transfer to or from local memory:

   DMA Buffer Allocate a DMA buffer and store the pointer in the pc_ram member of the DMA structure variable. This is the address to be used by the DMA controller.

   Local Address Specify the local memory address where the data are to be stored in the local member of the DMA structure variable. This is the base address for the data inside the InterWave IC.

   DMA Type Specify the direction of transfer in the type member of the DMA structure variable. Set this member to DMA_READ to download data to the InterWave local memory or set it to DMA_WRITE to upload data from local memory to the system. This member is a specification for the DMA controller and not the InterWave IC. If the DMA controller is to operate in auto-initialization mode, use the AUTO_READ and AUTO_WRITE symbolic constants.

Control Information

Specify the direction of transfer for the InterWave IC and whether to invert the most significant bit (MSB) of each sample for local memory DMA. To specify a DMA transfer to local memory (download), use the `DMA_DOWN` symbolic constant. To specify a DMA transfer from local memory (upload), use the `DMA_UP` symbolic constant. To specify the inversion of the MSB of each sample, use the `DMA_INV` symbolic constant. This information must be specified in the `cur_control` member of the **DMA** structure variable.

4. **Register the DmaCallback function to be called when the DMA transfer is completed and the terminal count interrupt is reflected in LDMACI[6].**

   This callback function sets a global flag to indicate DMA completion.

   Note that two of the arguments in the call to **IwaveSetCallback** are set to NULL because the program has no need for the second DMA channel or the MIDI interrupt.

   IwaveOpen installs a default callback function called **IwaveDefFunc** for all possible callbacks within the IWAVE variable. Therefore, if an application does not define a callback for a particular interrupt event and the interrupt occurs, **IwaveDefFunc** is called. This function simply executes a return.

5. **Reserve space in system memory for the data and then fill it with value 0 through value 15 in descending order.**

6. **Specify certain members of the DMA structure.**

   The pointer to system memory is specified in the structure member `pc_ram`. The type of DMA operation is specified in member `type`. The `cur_control` member contains the direction of the DMA transfer. The `cur_control` member affects the setting of the LMC DMA Control (LDMACI) register.

1. **Start the DMA transfer with a call to the IwaveDmaXfer function.**

   This function takes as arguments a pointer to the DMA `play` structure and the number of bytes to be downloaded. This function in turn calls other functions that program the DMA controller in the PC and the InterWave and eventually begins the transfer.

2. **Block other DMA activity until the transfer is complete with a call to the IwaveDmaWait function.**

   This function polls the `flag` member of the `iw` variable to determine when the transfer is done.

3. **Test the transfer by reading local memory with calls to the IwaveMemPeek function and by printing the results to the display.**

4. **Release the system memory.**

5. **Shut down the InterWave IC and the DDK with a call to the IwaveClose function.**

   The shutdown process performed by **IwaveClose** includes restoring the system's interrupt vector table to its original state. Failure to perform this step may result in the system hanging, crashing, or exhibiting erratic behavior.

Sample 8-1 System-to-Local Memory DMA Transfer Program

```
/*////////////////////////////////////////////////////////////////////////////
/ FILE: ldma.c
/
/ REMARKS: This program is an illustration of the steps needed to conduct a DMA
/ transfer to local memory. Note how the transfer is described to the DDK, how to
/ register a callback, and the DMA and IRQ structures.
/
////////////////////////////////////////////////////////////////////////////*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "iwdefs.h"
#include "iwprotos.h"
#include "iwcore.h"

BYTE dmaflag = 0;                /* callback flag */

/* step 1 */
void DmaCallback()
{
  dmaflag++;                     /* flag completion of DMA xfer */
}

void main()
{
  IRQ irq1;                      /* variable for interrupt service */
  DMA dma1;                      /* variable for DMA service */
  BYTE far *ptr, i;

  /* step 2 */
  IwaveOpen(14, GUS_MODE);    /* Initialize DDK and sound board */

  /* step 3 */
  IwaveSetInterface(&dma1, NULL, &irq1, NULL); /* register DMA and IRQ structures */

  /* step 4 */
  IwaveSetCallback(DmaCallback, PLAY_DMA_HANDLER);  /* register callback */

  /* step 5 */
  if ((dma1.pc_ram=farmalloc(16)) == NULL) {     /* allocate memory for data */
    printf("Failed to allocate DMA buffer\n");
    exit(-1);
  }
  dma1.local = 0L;               /* where data is in Local Mem */
  ptr = (BYTE far *)dma1.pc_ram;

  for (i = 0; i <= 15; i++)      /* load buffer with some data */
    *ptr++ = i;

  /* step 6 */
  /* describe transfer to DDK */
  dma1.cur_control |= DMA_DOWN;  /* tell InterWave IC to download */
  dma1.cur_control &= ~DMA_INV;  /* pass samples with no inversion */
  dma1.type = DMA_READ;          /* tell DMA controller to download */
```

```
  /* step 7 */
  if (IwaveDmaXfer(&dma1,16) != DMA_OK) {
    printf("DMA Failure\n");
  }
  else {
    /* step 8 */
    IwaveDmaWait();                /* block until DMA is done */

    /* step 9 */
    ptr = (BYTE far *)dma1.pc_ram;
    for (i = 0; i <= 15; i++)
      if (IwaveMemPeek(dma1.local + (ADDRESS)i) != *ptr++) {
        printf("DMA Error(%u)\n", i);
        break;
      }
  }

  printf("amount sent %u\n", dma1.amnt_sent);

  /* step 10 */
  farfree(dma1.pc_ram);          /* free buffer */

  /* step 11 */
  IwaveClose();                  /* close down DDK and board */
}
```

CHAPTER

9 GAME AND MIDI PORTS

The InterWave IC contains ports for supporting two standard PC joysticks and for sending and receiving Musical Instrument Digital Interface (MIDI) data.

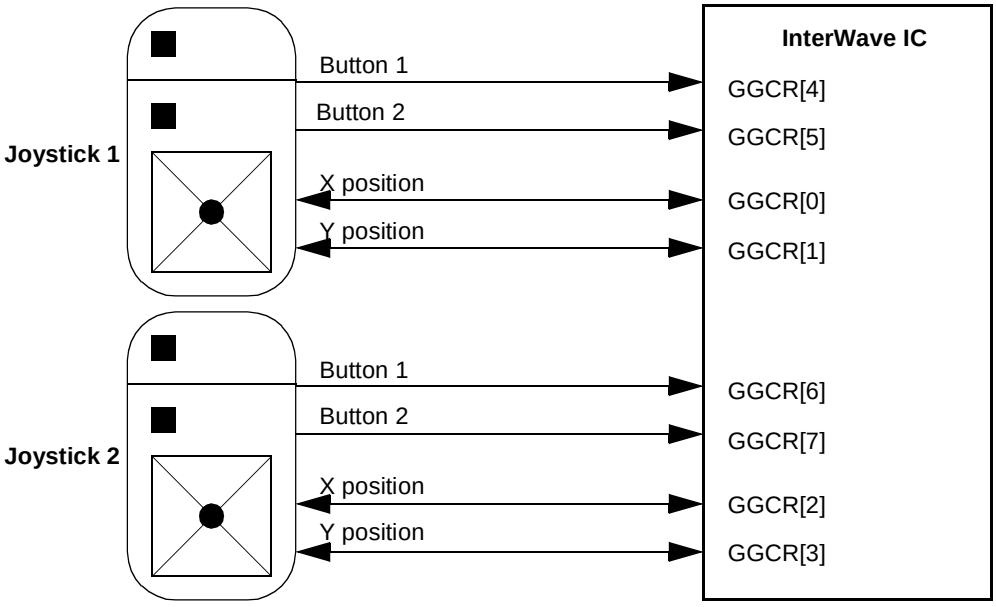
This chapter covers the following topics:

- Game port basics
- MIDI port basics
- Game port and MIDI port register overview
- Programming tips and examples

Game Port Basics

The game port on the InterWave IC provides the functions found in standard game ports in PCs. It supports X and Y position inputs for two joysticks plus four button inputs. The button inputs can be used for two, two-button joysticks or one, four-button joystick. By reading the game port through the game port registers, you can determine the X and Y position of the joysticks and the state of each of the buttons. Figure 9-1 shows the game port connections. On the Am78C202, the second joystick X and Y position connections are deleted.

Figure 9-1 Game Port Connections



Joystick Buttons

Bits 7–4 of the Game Control register (GGCR) reflect the current state of the button connected to it, as illustrated in Figure 9-1. These bits are normally High. When a button is pushed, the corresponding bit goes Low and stays low until the button is released.

Joystick X/Y Position

Bits 3–0 of the Game Control register (GGCR) provide a means of reading the joystick's X and Y position, as illustrated in Figure 9-1. Writing to the GGCR register sets bits 3–0 High. The joystick X/Y position lines are then compared to the joystick trim DAC to determine when the joystick X/Y capacitors charge up to the trim DAC voltage. When each of these lines reaches the trim DAC voltage, the corresponding bit in GGCR drops to Low. By timing how long the bit takes to drop to Low, a program can determine the X/Y position.

Joystick Trim DAC

The joystick trim DAC provides a voltage reference used as a threshold to determine when the joystick X/Y position bits should drop to Low. This threshold voltage can be adjusted for different systems by writing a 5-bit value to the Joystick Trim DAC register (GJTDI[4:0]). Software typically uses a counter based on a system clock to determine how long it takes for the X or Y position capacitors to charge. In a fast system, if the threshold comparison voltage is too high, the counter may overflow before the capacitor charges to the threshold. Setting the threshold voltage allows the software to use the maximum range of voltage for the best joystick position resolution without overflowing the joystick position counter. Some hardware vendors provide a utility program to test the joystick for the best joystick threshold voltage.

Game Port Register Overview

Table 9-1 lists the game port functions that can be accomplished through the InterWave programmable registers. For detailed information about the registers, see the reference page listed in the last column of the table. These references point to the applicable section in Chapter 16, "Game Port and MIDI Port Registers."

Table 9-1 Game Port Functions

Function	Register and Bit Field	Reference
Read the state of the joystick buttons	GGCR[7:4]	16-1
Read the state of the joystick position flags	GGCR[3:0]	16-1
Specify the joystick trim DAC level	GJTDI[4:0]	16-1

MIDI Port Basics

The Musical Instrument Digital Interface (MIDI) standard specifies a low-performance local area network (LAN) and describes the data that are passed onto the LAN. These data are geared toward controlling musical instruments such as synthesizers. The port allows for the connection of MIDI-compatible devices such as keyboards. The MIDI port can receive and transmit serial data at digital levels; external circuitry is required for the interface to the MIDI LAN.

MIDI UART

The MIDI interface hardware is, in essence, a UART.

From a software standpoint, the MIDI UART can be programmed to look like a Motorola MC6850 with limited programmability or a MPU-401 UART. The UART supports these basic functions:

- Bit rate = 31.25 kHz ±1%
- 1 start bit (0)
- 8 data bits
- 1 stop bit (1)
- DCD/, CTS/, RTS/ not implemented

In addition, a 16-byte MIDI receive FIFO has been included.

The MIDI port module can be programmed to generate an interrupt to the system control module when data enters the MIDI Receive Data register (GMRDR) or when the transmission of data is complete.

MIDI Receive FIFO and Register

A 16-byte receive FIFO sits between the UART and the MIDI Receive Data register (GMRDR). When GMRDR contains data, the MIDI port module generates an interrupt (if enabled). Reading the register clears the interrupt. If more MIDI data is received before the byte is read from the register, the new data is placed in the receive FIFO. If, after GMRDR is read, the FIFO contains more data, the module transfers the next byte from the FIFO to the register and generates another interrupt. Therefore, the IRQ assigned to the MIDI interrupt is cleared when the data is read but can be triggered again immediately as the data is passed from the FIFO to the register.

MIDI Loop Back

You can loop the contents of the MIDI Transmit Data register (GMTDR) directly back into the MIDI Receive Data register (GMRDR). When in loop-back mode, GMTDR still functions normally; that is, it transmits the looped data. However, in loop-back mode, the MIDI port cannot receive external data.

MIDI Port Register Overview

Table 9-1 lists the MIDI port functions that can be accomplished through the InterWave programmable registers. For detailed information about the registers, see the reference page listed in the last column of the table. These references point to the applicable section in Chapter 16, “Game Port and MIDI Port Registers.”

Table 9-2 MIDI Port Functions

Function	Register and Bit Field	Reference
Enable the MIDI receive data IRQ	GMCR[7]	16-2
Enable the MIDI transmit data IRQ	GMCR[6:5]	16-2
Reset the MIDI port	GMCR[1:0]	16-2
Read the status of the MIDI IRQ	GMSR[7]	16-3
Determine if MIDI receive FIFO is full	GMSR[5]	16-3
Determine if there has been a MIDI framing error	GMSR[4]	16-3
Determine if MIDI Transmit Data register (GMTDR) is available	GMSR[1]	16-3
Determine if MIDI Receive Data register (GMRDR) is full	GMSR[0]	16-3
Transmit MIDI data	GMTDR	16-4
Receive MIDI data	GMRDR	16-4
Write to the MIDI receive FIFO	GMRFAI	16-4

Programming Tips and Examples

This section provides information about how to accomplish the following game port and MIDI port tasks in an application program:

- Read the joystick X/Y position
- Read the joystick buttons
- Receive MIDI data
- Transmit MIDI data
- Loopback MIDI data

Reading the Joystick X/Y Position

Before the X and Y position of a joystick can be read, the joystick must be calibrated. A typical joystick calibration procedure consists of reading the capacitor charge times at the extreme X and Y positions of the joystick. These charge times then provide a maximum range against which subsequent charge times can be compared to determine the relative X or Y position.

To read the joystick position, use the following procedure. This procedure assumes that a threshold value has already been written to the Joystick Trim DAC register (GJTDI[4:0]).

1. **Write any value to the Game Control register.**

   This write sets the GGCR joystick X/Y position bits High and allows an external capacitor to be charged through the X and Y position lines to the joystick.

2. **Poll the Game Control port and time how long it takes for the GAMIO pins to drop to 0.**

   That time is how long it takes to charge the external capacitor to the threshold voltage through the X and Y position lines. Compare this time to the time obtained during calibration to determine the joystick's relative position.

CHAPTER

10 LEGACY SOUND CARD COMPATIBILITY AND EMULATION

The InterWave IC supports application software written to operate with legacy sound cards and synthesizers through two kinds of facilities: registers that mimic registers found in the earlier device, and general purpose registers that can be located in I/O address space to capture information written by the application software and to send information back to the application.

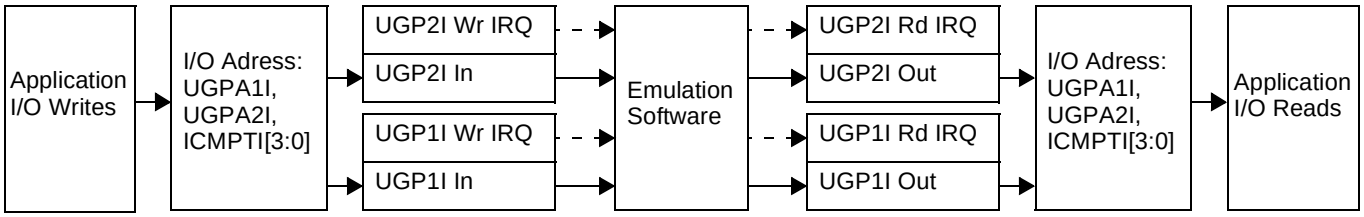
MPU-401 Emulation Basics

The InterWave IC supports MPU-401 emulation with two MPU-401 emulation registers and with the use of the general purpose registers.

General Purpose Registers

Two 8-bit general purpose registers in the IC are used for MPU-401 emulation and to support other emulation software. The general purpose registers, referred to as UGP1I and UGP2I, can be located anywhere in the ISA 10-bit I/O address space through ICMPTI[3:0], UGPA1I, and UGPA2I. Each general purpose register actually represents two registers: one that is read from by the application and one that is written to by the application. The application writes to one of these registers at the emulation address, causing an interrupt if the interrupt is enabled. The interrupt signals the emulation software to read the register through a back-door access location in the GUS Hidden Register Data Port (UHRDP). The emulation software can then write to the same back-door location to update the general purpose register before it is read by the application. Figure 10-1 illustrates the flow of information through the general purpose registers. The dashed arrows in the illustration stand for IRQ enables.

Figure 10-1 Data Flow Through the General Purpose Registers

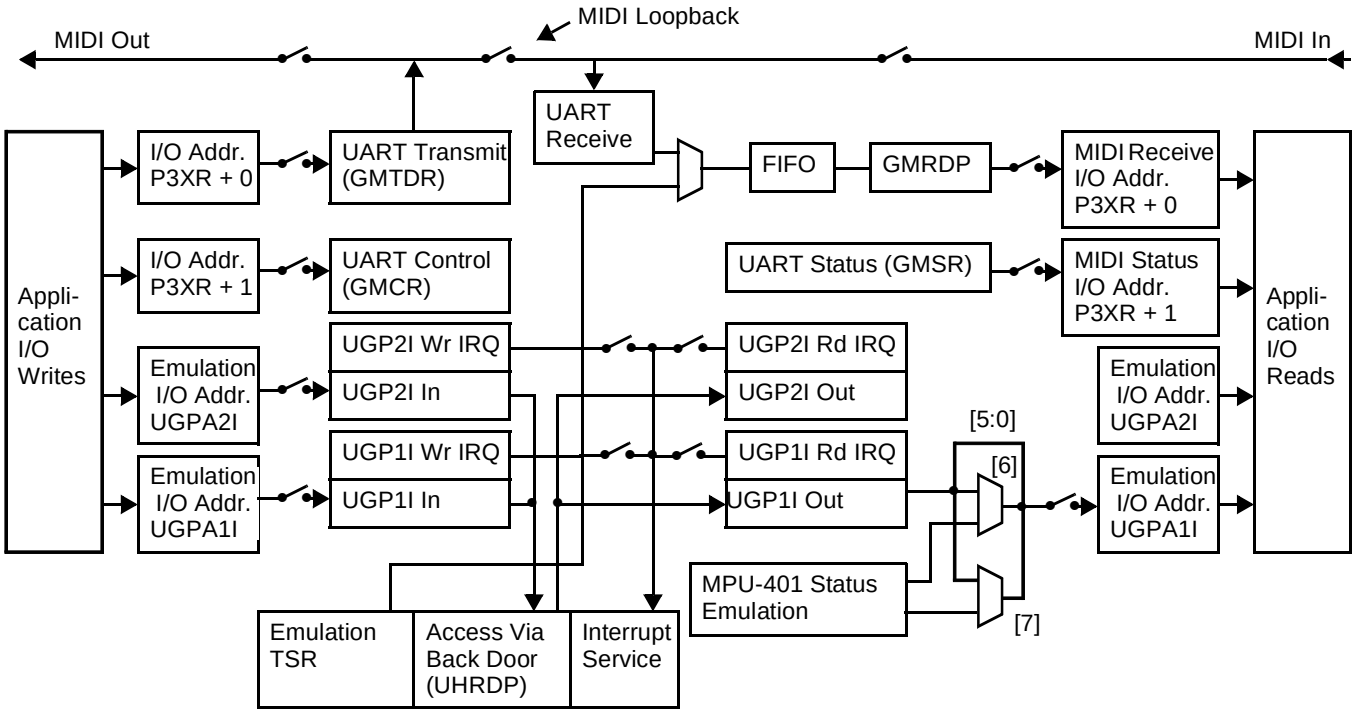


Several controls have been added to the general purpose registers as found in the GUS to support MPU-401 emulation. Such emulation assumes there is an MPU-401emulation terminate-and-stay-resident program (TSR) running concurrently with the application (typically game software). To match the MPU-401 card, the emulation address (UGPA1I, UGPA2I, and ICMPTI[3:0]) may be set to match the MIDI UART address. The two UART addresses can be swapped so that the MIDI Receive Data and MIDI Transmit Data registers (GMRDR and GMTDR) are accessed through P3XR+0 and the MIDI Control and MIDI Status registers (GMCR and GMSR) are accessed through P3XR+1. Writing to the general purpose registers causes an interrupt (potentially NMI). Emulation software captures the interrupt, reads the data in the emulation registers through the GUS Hidden Register Data

Port (UHRDP) and uses it to determine how to control the synthesizer. The MIDI commands may also be sent to the UART so that the application can be driven by the same interrupts and observe the same status as the MPU-401 card.

Figure 10-2 shows the access possibilities for the application and the emulation TSR. In that figure, the switch symbols are the enables controlled by the MPU-401 Emulation Control A and MPU-401 Emulation Control B registers (IEMUAI and IEMUBI).

Figure 10-2 Emulation Control Registers



MPU-401 Status Emulation

Two MPU-401 status bits are generated: Data Receive Ready (bit 6) and Data Send Ready (bit 7). The intended meaning of these bits is as follows: Data Receive Ready (DRR) becomes active (Low) when the host (CPU) is free to send a new command or data byte to the UART; Data Send Ready (DSR) becomes active (Low) when there is data available in the UART's receive data register. Note that the names of these bits are derived from the perspective of the MPU-401 hardware.

DRR is set inactive (High) by the hardware whenever there is a write to either of the emulation registers through the emulation address (ICMPTI[3:0], UGPA1I, UGPA2I), if a write to that register is enabled. Writes to UGP1I[6] through the back door (UHRDP) also update the state of this bit. This bit defaults to High at reset.

DSR is set inactive (High) by the hardware when there is a read of UGP2I via the emulation address (ICMPTI[3:2], UGPA2I). Writes to UGP1I[7] through the back door (UHRDP) also update the state of this bit. This bit defaults to Low at reset.

Legacy Sound Card Emulation

The InterWave IC contains programmable registers that support software written for the AdLib sound card or the Sound Blaster sound card. Emulation of legacy sound cards follows a basic 3-step process:

4. Trap the various emulation interrupts.
5. Upon trapping an emulation interrupt, read any register that contains information pertaining to that interrupt.
6. Take appropriate actions based on the interrupt and information read.

Table 10-1 lists and briefly describes the registers involved in the emulation of legacy sound cards. For detailed information about the AdLib–Sound Blaster emulation registers, go to the reference pages listed in the last column of Table 10-1.

Table 10-1 AdLib and Sound Blaster Emulation Registers

Mnemonic	Register	Description	Reference
UISR	IRQ Status	Contains several interrupt status bits for AdLib–Sound Blaster emulation.	12-2
U2X6R	Sound Blaster 2X6	Writing to this register triggers the Write to 2X6 interrupt, whose status is reflected in UASRR[3].	12-3
UACWR	AdLib Command Write	Written to by AdLib application software to program the internal synthesizer to duplicate the AdLib sound.	12-3
UASRR	AdLib Status Read	Contains several interrupt status bits for AdLib and Sound Blaster emulation.	12-3
UADR	AdLib Data	Performs AdLib functions based on the state of various bits. Writing to this register causes an interrupt.	12-4
UACRR	AdLib Command Read	Read by AdLib application software.	12-3
UASWR	AdLib Status Write	See UASRR.	12-3
UI2XCR	Sound Blaster IRQ 2XC	The Sound Blaster 2XC register. Writing to this register causes an interrupt if the Sound Blaster interrupts are enabled.	12-5
U2XCR	Sound Blaster 2XC (no IRQ)	Allows writing to the Sound Blaster 2XC register without generating an interrupt.	12-6
U2XER	Sound Blaster 2XE	The Sound Blaster 2XE register. Writing to this register causes an interrupt if the U2XER interrupt is enabled.	12-6
URCR	Register Control	Contains an enable bit for the U2XER interrupt, and the *Toggle UI2XCR[7]* bit.	12-6
USRR	Status Read	Contains the 2XE interrupt status bit.	12-7
UICI	Interrupt Control	Contains the *AdLib–Sound Blaster to NMI* bit for sending AdLib–Sound Blaster interrupts to NMI instead of the selected IRQ channel.	12-9
UASBCI	AdLib–Sound Blaster Control	Controls the AdLib and Sound Blaster emulation hardware. Contains enabling bits for various emulation interrupts and the auto-timer mode.	12-12
UAT1I	AdLib Timer 1	Contains the load value for AdLib Timer 1. See UADR for the *Start Timer 1* bit.	12-13

Table 10-1 AdLib and Sound Blaster Emulation Registers (Continued)

Mnemonic	Register	Description	Reference
UAT2I	AdLib Timer 2	Contains the load value for AdLib Timer 2. See UADR for the *Start Timer 2* bit.	12-13
IDECI	Decode Control	Contains bits for enabling reading and writing of the Sound Blaster and AdLib registers.	12-16
IEIRQI	Emulation Control	Contains a bit for controling the state of the IRQ line selecting in PSBISI.	12-20
PSACTI	PNP AdLib–Sound Blaster Activate	Activates the AdLib-Sound Blaster emulation logical device.	12-24
PSRCI	PNP AdLib–Sound Blaster I/O Range Check	Provides a method for checking the AdLib-Sound Blaster emulation I/O addresses.	12-24
PSBISI	PNP AdLib–Sound Blaster Emulation IRQ Select	Selects the IRQ number for the Sound Blaster emulation interrupt.	12-25
PSBITI	PNP AdLib–Sound Blaster Emulation IRQ Type	Provides data back to standard PNP software concerning the type of interrupts supported by the InterWave IC.	12-26

Part 3

InterWave Registers Reference

This section provides the information needed to program each of the InterWave user-accessible registers. The registers are separated into chapters by major component:

- System control
- Codec
- Synthesizer
- Local memory control
- Game and MIDI ports

Notes: ***1.*** *In the following register definitions,* Res *or* Reserved *specifies reserved bits. Unless noted otherwise, such fields must be written with zeros. Reading a reserved bit returns an indeterminate value. A read-modify-write operation can write back the value read.*

2. *To make the following register definitions easier to understand, this book uses certain conventions when describing bit positions and bit values. For information about the conventions used, see "Typographical Conventions" on page -xix.*

CHAPTER

11 REGISTER SUMMARY

The InterWave audio IC has hundreds of user-accessible registers for controlling all aspects of its operation. This chapter describes the conventions used to name the registers and lists every user-accessible register in the InterWave IC along with its address, default value, and a page reference to the detailed register information in Part , "InterWave Registers Reference."

Information about how to program the InterWave IC through the registers can be found in Part , "Programming the InterWave IC." For information about the relocatable address spaces, direct and indirect registers, and external decoding mode, see "Accessing InterWave Registers" on page 4-2.

Register Naming Conventions

To help you understand how the InterWave IC operates, we have used the following mnemonic conventions in naming registers:

- The first character of a register's name is a code letter that specifies the InterWave functional block to which the register belongs (see Table 11-1).
- The middle two to four characters describe the function of the register.
- The final character is either R for a direct register, P for a port (to access an array of indexed registers), or I for an indirect register.

For example, the IGIDXR register breaks down as follows:

- A system control register (I)
- The General Index register (GIDX)
- A direct register (R)—which means software writes directly to the register address without having to write an index value to some other register first.

Table 11-1 Module Mnemonics

Code	Function
C	Codec digital section
G	Game port (joystick), MIDI port
I	Interface (system control, system bus interface)
L	Local memory control
P	Plug and Play ISA
S	Synthesizer
U	UltraSound (GUS compatibility, emulation of legacy sound cards)

Registers By I/O Address

Table 11-2 lists the InterWave I/O programmable registers and ports in the order of their I/O addresses. Table 11-3 lists the same registers and ports alphabetized by mnemonic name. All addresses are in hexadecimal. Default values are shown in hexadecimal (00h) or binary (0000 0000). When shown in binary, an x in a bit position means the bit value is indeterminate. A question mark in a bit position means the default bit value may be either High or Low depending on certain conditions, as described in the detailed information in Part , "InterWave Registers Reference." The last column of the table shows the page in Part 3 that contains detailed information about the register or port.

Table 11-2 InterWave Registers and Ports by I/O Address

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page only
UMCR	Mix Control	P2XR+0	—	r/w	03h	12-1
UISR	IRQ Status	P2XR+6	—	read	00h	12-2
U2X6R	Sound Blaster 2X6	P2XR+6	—	write	—	12-3
UACWR	AdLib Command Write	P2XR+8, P388AR+0	—	write	00h	12-3
UASRR	AdLib Status Read	P2XR+8, P388AR+0	—	read	00h	12-3
UADR	AdLib Data	P2XR+9, P389AR+1	—	r/w	00h	12-4
UACRR	AdLib Command Read	P2XR+0Ah	—	read	00h	12-3
UASWR	AdLib Status Write	P2XR+0Ah	—	write	00h	12-3
UHRDP	GUS Hidden Register Data Port	P2XR+0Bh	—	r/w	—	12-5
UI2XCR	Sound Blaster IRQ 2XC	P2XR+0Ch	—	r/w	00h	12-5
U2XCR	Sound Blaster 2XC (no IRQ)	P2XR+0Dh	—	write	00h	12-6
U2XER	Sound Blaster 2XE	P2XR+0Eh	—	r/w	00h	12-6
URCR	Register Control	P2XR+0Fh	—	r/w	00h	12-6
USRR	Status Read	P2XR+0Fh	—	read	01h	12-7
UDCI	DMA Channel Control	P2XR+0Bh	UMCR[6]=0, URCR[2:0]=0	r/w	00h	12-8
UICI	Interrupt Control	P2XR+0Bh	UMCR[6]=1, URCR[2:0]=0	write	07h	12-9
UGP1I	General Purpose Register 1 (Back Door)	P2XR+0Bh	URCR[2:0]=1	r/w	00h	12-10
UGP2I	General Purpose Register 2 (Back Door)	P2XR+0Bh	URCR[2:0]=2	r/w	00h	12-10
UGP1I	General Purpose Register 1 (Emulation Address)	UGPA1I		r/w	00h	12-10
UGP2I	General Purpose Register 2 (Emulation Address)	UGPA2I		r/w	00h	12-10
UGPA1I	General Purpose Register 1 Address	P2XR+0Bh	URCR[2:0]=3	r/w	00h	12-11
UGPA2I	General Purpose Register 2 Address	P2XR+0Bh	URCR[2:0]=4	r/w	00h	12-11
UCLRII	Clear Interrupt	P2XR+0Bh	URCR[2:0]=5	write	—	12-11
UJMPI	Jumper	P2XR+0Bh	URCR[2:0]=6	r/w	06h	12-11
GGCR	Game Control	P201AR+0	—	r/w	x0h	16-1
GMCR	MIDI Control	P3XR+0	—	r/w	0x0x xxx0	16-2
GMSR	MIDI Status	P3XR+0	—	read	0x00 xx10	16-3
GMTDR	MIDI Transmit Data	P3XR+1	—	write	—	16-4
GMRDR	MIDI Receive Data	P3XR+1	—	read	FFh	16-4
SVSR	Synthesizer Voice Select	P3XR+2	—	r/w	00h	14-1

Table 11-2 InterWave Registers and Ports by I/O Address (Continued)

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page only
IGIDXR	General Index	P3XR+3	—	r/w	00h	12-12
I16DP	General 16-bit I/O Data Port	P3XR+(4-5)	—	r/w	—	12-12
I8DP	General 8-bit I/O Data Port	P3XR+5	—	r/w	—	12-12
SACI	Synth. Address Control (1 per voice)	P3XR+5	IGIDXR=0, 80	w, r	01h	14-8
SFCI	Synth. Frequency Control (1 per voice)	P3XR+(4-5)	IGIDXR=1, 81	w, r	0400h	14-7
SASHI	Synth. Address Start High (1 per voice)	P3XR+(4-5)	IGIDXR=2, 82	w, r	0000h	14-4
SASLI	Synth. Address Start Low (1 per voice)	P3XR+(4-5)	IGIDXR=3, 83	w, r	0000h	14-5
SAEHI	Synth. Address End High (1 per voice)	P3XR+(4-5)	IGIDXR=4, 84	w, r	0000h	14-5
SAELI	Synth. Address End Low (1 per voice)	P3XR+(4-5)	IGIDXR=5, 85	w, r	0000h	14-5
SVRI	Synth. Volume Rate (1 per voice)	P3XR+5	IGIDXR=6, 86	w, r	00h	14-10
SVSI	Synth. Volume Start (1 per voice)	P3XR+5	IGIDXR=7, 87	w, r	00h	14-9
SVEI	Synth. Volume End (1 per voice)	P3XR+5	IGIDXR=8, 88	w, r	00h	14-9
SVLI	Synth. Volume Level (1 per voice)	P3XR+(4-5)	IGIDXR=9, 89	w, r	0000h	14-10
SAHI	Synth. Address High (1 per voice)	P3XR+(4-5)	IGIDXR=A, 8A	w, r	0000h	14-6
SALI	Synth. Address Low (1 per voice)	P3XR+(4-5)	IGIDXR=B, 8B	w, r	0000h	14-6
SROI	Synth. Right Offset (1 per voice)	P3XR+(4-5)	IGIDXR=C, 8C	w, r	0700h	14-12
SVCI	Synth. Volume Control (1 per voice)	P3XR+5	IGIDXR=D, 8D	w, r	01h	14-10
SAVI	Synth. Active Voices	P3XR+5	IGIDXR=E, 8E	w, r	CDh	14-1
SVII	Synth. Voices IRQ	P3XR+5	IGIDXR=8F	read	E0h	14-2
SUAI	Synth. Upper Address (1 per voice)	P3XR+5	IGIDXR=10, 90	w, r	00h	14-4
SEAHI	Synth. Effects Address High (1 per voice)	P3XR+(4-5)	IGIDXR=11, 91	w, r	0000h	14-7
SEALI	Synth. Effects Address Low (1 per voice)	P3XR+(4-5)	IGIDXR=12, 92	w, r	0000h	14-7
SLOI	Synth. Left Offset (1 per voice)	P3XR+(4-5)	IGIDXR=13, 93	w, r	0000h	14-13
SEASI	Synth. Effects Output Accumulator Select (1 per voice)	P3XR+5	IGIDXR=14, 94	w, r	00h	14-14
SMSI	Synth. Mode Select (1 per voice)	P3XR+5	IGIDXR=15, 95	w, r	02h	14-14
SEVI	Synth. Effects Volume (1 per voice)	P3XR+(4-5)	IGIDXR=16, 96	w, r	0000h	14-13
SFLFOI	Synth. Frequency LFO (1 per voice)	P3XR+5	IGIDXR=17, 97	w, r	00h	14-8
SVLFOI	Synth. Volume LFO (1 per voice)	P3XR+5	IGIDXR=18, 98	w, r	00h	14-11
SGMI	Synth. Global Mode	P3XR+5	IGIDXR=19, 99	w, r	00h	14-3
SLFOBI	Synth. LFO Base Address	P3XR+(4-5)	IGIDXR=1A, 9A	w, r	0000h	14-3
SROFI	Synth. Right Offset Final (1 per voice)	P3XR+(4-5)	IGIDXR=1B, 9B	w, r	0700h	14-12
SLOFI	Synth. Left Offset Final (1 per voice)	P3XR+(4-5)	IGIDXR=1C, 9C	w, r	0000h	14-13
SEVFI	Synth. Effect Volume Final (1 per voice)	P3XR+(4-5)	IGIDXR=1D, 9D	w, r	0000h	14-14
SVIRI	Synth. Voices IRQ Read	P3XR+5	IGIDXR=9F	read	E0h	14-2
LDMACI	LMC DMA Control	P3XR+5	IGIDXR=41	r/w	00h	15-1
LDSALI	LMC DMA Start Address Low (bits 19–4)	P3XR+(4-5)	IGIDXR=42	r/w	0000h	15-2
LMALI	LMC I/O Address Low (bits 15–0)	P3XR+(4-5)	IGIDXR=43	write	0000h	15-3
LMAHI	LMC I/O Address High (bits 23–16)	P3XR+5	IGIDXR=44	write	00h	15-3
UASBCI	AdLib–Sound Blaster Control	P3XR+5	IGIDXR=45	r/w	00h	12-12
UAT1I	AdLib Timer 1	P3XR+5	IGIDXR=46	r/w	00h	12-13
UAT2I	AdLib Timer 2	P3XR+5	IGIDXR=47	r/w	00h	12-13
USCI	ADC Sample Control	P3XR+5	IGIDXR=49	r/w	00h	12-13
GJTDI	Joystick Trim DAC	P3XR+5	IGIDXR=4B	r/w	1Dh	16-1
URSTI	GUS Reset	P3XR+5	IGIDXR=4C	write	xxxx x000	12-14
LDSAHI	LMC DMA Start Address High (bits 23–20, 3–0)	P3XR+5	IGIDXR=50	r/w	00h	15-3

Table 11-2 InterWave Registers and Ports by I/O Address (Continued)

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page only
LMSBAI	LMC 16-Bit Access	P3XR+(4-5)	IGIDXR=51	r/w	—	15-4
LMCFI	LMC Configuration	P3XR+(4-5)	IGIDXR=52	r/w	0000h	15-4
LMCI	LMC Control	P3XR+5	IGIDXR=53	r/w	00h	15-5
LMRFAI	LMC Record FIFO Base Address (bits 23–8)	P3XR+(4-5)	IGIDXR=54	r/w	0000h	15-6
LMPFAI	LMC Playback FIFO Base Address (bits 23–8)	P3XR+(4-5)	IGIDXR=55	r/w	0000h	15-6
LMFSI	LMC FIFO Size	P3XR+(4-5)	IGIDXR=56	r/w	0000h	15-6
LDICI	LMC DMA Interleave Control	P3XR+(4-5)	IGIDXR=57	r/w	0000h	15-7
LDIBI	LMC DMA Interleave Base Address (bits 23–8)	P3XR+(4-5)	IGIDXR=58	r/w	0000h	15-7
ICMPTI	Compatibility	P3XR+5	IGIDXR=59	r/w	1Fh	12-15
IDECI	Decode Control	P3XR+5	IGIDXR=5A	r/w	7Fh	12-16
IVERI	Version Number	P3XR+5	IGIDXR=5B	r/w	0010 0?00	12-17
IEMUAI	MPU-401 Emulation Control A	P3XR+5	IGIDXR=5C	r/w	00h	12-18
IEMUBI	MPU-401 Emulation Control B	P3XR+5	IGIDXR=5D	r/w	30h	12-19
GMRFAI	MIDI Receive FIFO Access	P3XR+5	IGIDXR=5E	write	—	16-4
	Reserved	P3XR+5	IGIDXR=5F			
IEIRQI	Emulation IRQ	P3XR+5	IGIDXR=60	r/w	0?xx xx00	12-20
LMBDR	LMC Byte Data	P3XR+7	—	r/w	—	15-1
CIDXR	Codec Index Address	PCODAR+0	—	r/w	40h	13-1
CDATAP	Codec Indexed Data Port	PCODAR+1	—	r/w	—	13-2
CSR1R	Codec Status Register 1	PCODAR+2	—	read	CCh	13-2
CPDR	Playback Data	PCODAR+3	—	write	—	13-4
CRDR	Record Data	PCODAR+3	—	read	—	13-4
CLICI	Left ADC Input Control	PCODAR+1	CIDXR[4:0]=0	r/w	000x 0000	13-4
CRICI	Right ADC Input Control	PCODAR+1	CIDXR[4:0]=1	r/w	000x 0000	13-4
CLAX1I	Left Auxiliary 1/Synthesizer Input Control	PCODAR+1	CIDXR[4:0]=2	r/w	1xx0 1000	13-5
CRAX1I	Right Auxiliary 1/Synthesizer Input Control	PCODAR+1	CIDXR[4:0]=3	r/w	1xx0 1000	13-5
CLAX2I	Left Auxiliary 2 Input Control	PCODAR+1	CIDXR[4:0]=4	r/w	1xx0 1000	13-5
CRAX2I	Right Auxiliary 2 Input Control	PCODAR+1	CIDXR[4:0]=5	r/w	1xx0 1000	13-5
CLDACI	Left Playback DAC Control	PCODAR+1	CIDXR[4:0]=6	r/w	1x00 0000	13-6
CRDACI	Right Playback DAC Control	PCODAR+1	CIDXR[4:0]=7	r/w	1x00 0000	13-6
CPDFI	Playback Data Format	PCODAR+1	CIDXR[4:0]=8	r/w	00h	13-6
CFIG1I	Configuration Register 1	PCODAR+1	CIDXR[4:0]=9	r/w	00xx 1000	13-7
CEXTI	External Control	PCODAR+1	CIDXR[4:0]=A	r/w	00xx 0x0x	13-8
CSR2I	Codec Status Register 2	PCODAR+1	CIDXR[4:0]=B	read	00h	13-9
CMODEI	Mode Select, ID	PCODAR+1	CIDXR[4:0]=C	r/w	100x 1010	13-10
CLCI	Loopback Control	PCODAR+1	CIDXR[4:0]=D	r/w	0000 00x0	13-10
CUPCTI	Upper Playback Count	PCODAR+1	CIDXR[4:0]=E	r/w	00h	13-11
CLPCTI	Lower Playback Count	PCODAR+1	CIDXR[4:0]=F	r/w	00h	13-11
CFIG2I	Configuration Register 2	PCODAR+1	CIDXR[4:0]=10	r/w	0000 xxx0	13-11
CFIG3I	Configuration Register 3	PCODAR+1	CIDXR[4:0]=11	r/w	0000 x000	13-12
CLLICI	Left Line Input Control	PCODAR+1	CIDXR[4:0]=12	r/w	1xx0 1000	13-13
CRLICI	Right Line Input Control	PCODAR+1	CIDXR[4:0]=13	r/w	1xx0 1000	13-13
CLTIMI	Lower Timer	PCODAR+1	CIDXR[4:0]=14	r/w	00h	13-14

Table 11-2 InterWave Registers and Ports by I/O Address (Continued)

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page only
CUTIMI	Upper Timer	PCODAR+1	CIDXR[4:0]=15	r/w	00h	13-14
CLMICI	Left Microphone Input Control	PCODAR+1	CIDXR[4:0]=16	r/w	1xx0 1000	13-14
CRMICI	Right Microphone Input Control	PCODAR+1	CIDXR[4:0]=17	r/w	1xx0 1000	13-14
CSR3I	Codec Status Register 3	PCODAR+1	CIDXR[4:0]=18	r/w	x000 0000	13-15
CLOAI	Left Output Attenuation	PCODAR+1	CIDXR[4:0]=19	r/w	1xx0 0000	13-16
CMONOI	Mono Input and Output Control	PCODAR+1	CIDXR[4:0]=1A	r/w	000x 0000	13-17
CROAI	Right Output Attenuation	PCODAR+1	CIDXR[4:0]=1B	r/w	1xx0 0000	13-16
CRDFI	Record Data Format	PCODAR+1	CIDXR[4:0]=1C	r/w	00h	13-17
CPVFI	Playback Variable Frequency	PCODAR+1	CIDXR[4:0]=1D	r/w	00h	13-18
CURCTI	Upper Record Count	PCODAR+1	CIDXR[4:0]=1E	r/w	00h	13-19
CLRCTI	Lower Record Count	PCODAR+1	CIDXR[4:0]=1F	r/w	00h	13-19
PCSNBR	PNP Card Select Number Back Door	201	—	write	00h	12-21
PIDXR	PNP Index Address	279	—	write	00h	12-21
PNPWRP	PNP Write Data Port	A79	—	write	—	12-21
PNPRDP	PNP Read Data Port	PNPRDP	—	read	—	12-21
PSRPAI	PNP Set Read Data Port Address	A79	PIDXR=00	write	80h	12-22
PISOCI	PNP Isolate Command	PNPRDP	PIDXR=01	read	—	12-22
PCCCI	PNP Configuration Control Command	A79	PIDXR=02	write	—	12-22
PWAKEI	PNP Wake[CSN] Command	A79	PIDXR=03	write	—	12-23
PRESDI	PNP Resource Data	PNPRDP	PIDXR=04	read	00h	12-23
PRESSI	PNP Resource Data Status	PNPRDP	PIDXR=05	read	00h	12-23
PCSNI	PNP Card Select Number	PNPRDP, A79	PIDXR=06	r/w	00h	12-23
PLDNI	PNP Logical Device Number (LDN)	PNPRDP, A79	PIDXR=07	r/w	00h	12-23
PUACTI	PNP Audio Activate	PNPRDP, A79	LDN=0, PIDXR=30	r/w	00h	12-24
PURCI	PNP Audio I/O Range Check	PNPRDP, A79	LDN=0, PIDXR=31	r/w	00h	12-24
P2X0HI	PNP P2XR[15:8]	PNPRDP, A79	LDN=0, PIDXR=60	r/w	00h	12-25
P2X0LI	PNP P2XR[7:0]	PNPRDP, A79	LDN=0, PIDXR=61	r/w	00h	12-25
P3X0HI	PNP P3XR[15:8]	PNPRDP, A79	LDN=0, PIDXR=62	r/w	00h	12-25
P3X0LI	PNP P3XR[7:0]	PNPRDP, A79	LDN=0, PIDXR=63	r/w	00h	12-25
PHCAI	PNP PCODAR[15:8]	PNPRDP, A79	LDN=0, PIDXR=64	r/w	00h	12-25
PLCAI	PNP PCODAR[7:0]	PNPRDP, A79	LDN=0, PIDXR=65	r/w	00h	12-25
PUI1SI	PNP Audio IRQ Channel 1 Select	PNPRDP, A79	LDN=0, PIDXR=70	r/w	00h	12-25
PUI1TI	PNP Audio IRQ Channel 1 Type	PNPRDP	LDN=0, PIDXR=71	read	02h	12-26
PUI2SI	PNP Audio IRQ Channel 2 Select	PNPRDP, A79	LDN=0, PIDXR=72	r/w	00h	12-25
PUI2TI	PNP Audio IRQ Channel 2 Type	PNPRDP	LDN=0, PIDXR=73	read	02h	12-26
PUD1SI	PNP Audio DMA Channel 1 Select	PNPRDP, A79	LDN=0, PIDXR=74	r/w	04h	12-27

Table 11-2 InterWave Registers and Ports by I/O Address (Continued)

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page only
PUD2SI	PNP Audio DMA Channel 2 Select	PNPRDP, A79	LDN=0, PIDXR=75	r/w	04h	12-27
PSEENI	PNP Serial EEPROM Enable	PNPRDP, A79	LDN=0, PIDXR=F0	r/w	00h	12-28
PSECI	PNP Serial EEPROM Control	PNPRDP, A79	LDN=0, PIDXR=F1	r/w	xxxx 000x	12-28
PPWRI	PNP Power Mode	PNPRDP, A79	LDN=0, PIDXR=F2	r/w	x111 1111	12-29
PSRSTI	PNP Software Reset	PNPRDP, A79	LDN=0, PIDXR=F3	write	—	12-30
PRACTI	PNP CD-ROM Activate	PNPRDP, A79	LDN=1, PIDXR=30	r/w	00h	12-24
PRRCI	PNP CD-ROM I/O Range Check	PNPRDP, A79	LDN=1, PIDXR=31	r/w	00h	12-24
PRAHI	PNP PCDRAR[15:8]	PNPRDP, A79	LDN=1, PIDXR=60	r/w	00h	12-25
PRALI	PNP PCDRAR[7:0]	PNPRDP, A79	LDN=1, PIDXR=61	r/w	00h	12-25
PATAHI	PNP PATAAR[15:8]	PNPRDP, A79	LDN=1, PIDXR=62	r/w	00h	12-25
PATALI	PNP PATAAR[7:0]	PNPRDP, A79	LDN=1, PIDXR=63	r/w	00h	12-25
PRISI	PNP CD-ROM IRQ Select	PNPRDP, A79	LDN=1, PIDXR=70	r/w	00h	12-25
PRITI	PNP CD-ROM IRQ Type	PNPRDP	LDN=1, PIDXR=71	read	02h	12-26
PRDSI	PNP CD-ROM DMA Select	PNPRDP, A79	LDN=1, PIDXR=74	r/w	04h	12-27
PGACTI	PNP Game Port Activate	PNPRDP, A79	LDN=2, PIDXR=30	r/w	00h	12-24
PGRCI	PNP Game Port I/O Range Check	PNPRDP, A79	LDN=2, PIDXR=31	r/w	00h	12-24
P201HI	PNP P201AR[15:8]	PNPRDP, A79	LDN=2, PIDXR=60	r/w	00h	12-25
P201LI	PNP P201AR[7:0]	PNPRDP, A79	LDN=2, PIDXR=61	r/w	01h	12-25
PSACTI	PNP AdLib–Sound Blaster Activate	PNPRDP, A79	LDN=3, PIDXR=30	r/w	00h	12-24
PSRCI	PNP AdLib–Sound Blaster I/O Range Check	PNPRDP, A79	LDN=3, PIDXR=31	r/w	00h	12-24
P388HI	PNP P388AR[15:8]	PNPRDP, A79	LDN=3, PIDXR=60	r/w	00h	12-25
P388LI	PNP P388AR[7:0]	PNPRDP, A79	LDN=3, PIDXR=61	r/w	08h	12-25
PSBISI	PNP AdLib–Sound Blaster Emulation IRQ Select	PNPRDP, A79	LDN=3, PIDXR=70	r/w	00h	12-25
PSBITI	PNP AdLib–Sound Blaster Emulation IRQ Type	PNPRDP	LDN=3, PIDXR=71	read	02h	12-26
PMACTI	PNP MPU-401 Activate	PNPRDP, A79	LDN=4, PIDXR=30	r/w	00h	12-24
PMRCI	PNP MPU-401 I/O Range Check	PNPRDP, A79	LDN=4, PIDXR=31	r/w	00h	12-24

Table 11-2 InterWave Registers and Ports by I/O Address (Continued)

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page only
P401HI	PNP P401AR[15:8]	PNPRDP, A79	LDN=4, PIDXR=60	r/w	00h	12-25
P401LI	PNP P401AR[7:0]	PNPRDP, A79	LDN=4, PIDXR=61	r/w	00h	12-25
PMISI	PNP MPU-401 Emulation IRQ Select	PNPRDP, A79	LDN=4, PIDXR=70	r/w	00h	12-25
PMITI	PNP MPU-401 Emulation IRQ Type	PNPRDP	LDN=4, PIDXR=71	read	02h	12-26

Registers By Mnemonic

Table 11-3 InterWave Registers and Ports by Mnemonic

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page
CDATAP	Codec Indexed Data Port	PCODAR+1	—	r/w	—	13-2
CEXTI	External Control	PCODAR+1	CIDXR[4:0]=A	r/w	00xx 0x0x	13-8
CFIG1I	Configuration Register 1	PCODAR+1	CIDXR[4:0]=9	r/w	00xx 1000	13-7
CFIG2I	Configuration Register 2	PCODAR+1	CIDXR[4:0]=10	r/w	0000 xxx0	13-11
CFIG3I	Configuration Register 3	PCODAR+1	CIDXR[4:0]=11	r/w	0000 x000	13-12
CIDXR	Codec Index Address	PCODAR+0	—	r/w	40h	13-1
CLAX1I	Left Auxiliary 1/Synthesizer Input Control	PCODAR+1	CIDXR[4:0]=2	r/w	1xx0 1000	13-5
CLAX2I	Left Auxiliary 2 Input Control	PCODAR+1	CIDXR[4:0]=4	r/w	1xx0 1000	13-5
CLCI	Loopback Control	PCODAR+1	CIDXR[4:0]=D	r/w	0000 00x0	13-10
CLDACI	Left Playback DAC Control	PCODAR+1	CIDXR[4:0]=6	r/w	1x00 0000	13-6
CLICI	Left ADC Input Control	PCODAR+1	CIDXR[4:0]=0	r/w	000x 0000	13-4
CLLICI	Left Line Input Control	PCODAR+1	CIDXR[4:0]=12	r/w	1xx0 1000	13-13
CLMICI	Left Microphone Input Control	PCODAR+1	CIDXR[4:0]=16	r/w	1xx0 1000	13-14
CLOAI	Left Output Attenuation	PCODAR+1	CIDXR[4:0]=19	r/w	1xx0 0000	13-16
CLPCTI	Lower Playback Count	PCODAR+1	CIDXR[4:0]=F	r/w	00h	13-11
CLRCTI	Lower Record Count	PCODAR+1	CIDXR[4:0]=1F	r/w	00h	13-19
CLTIMI	Lower Timer	PCODAR+1	CIDXR[4:0]=14	r/w	00h	13-14
CMODEI	Mode Select, ID	PCODAR+1	CIDXR[4:0]=C	r/w	100x 1010	13-10
CMONOI	Mono Input and Output Control	PCODAR+1	CIDXR[4:0]=1A	r/w	000x 0000	13-17
CPDFI	Playback Data Format	PCODAR+1	CIDXR[4:0]=8	r/w	00h	13-6
CPDR	Playback Data	PCODAR+3	—	write	—	13-4
CPVFI	Playback Variable Frequency	PCODAR+1	CIDXR[4:0]=1D	r/w	00h	13-18
CRAX1I	Right Auxiliary 1/Synthesizer Input Control	PCODAR+1	CIDXR[4:0]=3	r/w	1xx0 1000	13-5
CRAX2I	Right Auxiliary 2 Input Control	PCODAR+1	CIDXR[4:0]=5	r/w	1xx0 1000	13-5
CRDACI	Right Playback DAC Control	PCODAR+1	CIDXR[4:0]=7	r/w	1x00 0000	13-6
CRDFI	Record Data Format	PCODAR+1	CIDXR[4:0]=1C	r/w	00h	13-17
CRDR	Record Data	PCODAR+3	—	read	—	13-4
CRICI	Right ADC Input Control	PCODAR+1	CIDXR[4:0]=1	r/w	000x 0000	13-4
CRLICI	Right Line Input Control	PCODAR+1	CIDXR[4:0]=13	r/w	1xx0 1000	13-13
CRMICI	Right Microphone Input Control	PCODAR+1	CIDXR[4:0]=17	r/w	1xx0 1000	13-14
CROAI	Right Output Attenuation	PCODAR+1	CIDXR[4:0]=1B	r/w	1xx0 0000	13-16

Table 11-3 InterWave Registers and Ports by Mnemonic (Continued)

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page
CSR1R	Codec Status Register 1	PCODAR+2	—	read	CCh	13-2
CSR2I	Codec Status Register 2	PCODAR+1	CIDXR[4:0]=B	read	00h	13-9
CSR3I	Codec Status Register 3	PCODAR+1	CIDXR[4:0]=18	r/w	x000 0000	13-15
CUPCTI	Upper Playback Count	PCODAR+1	CIDXR[4:0]=E	r/w	00h	13-11
CURCTI	Upper Record Count	PCODAR+1	CIDXR[4:0]=1E	r/w	00h	13-19
CUTIMI	Upper Timer	PCODAR+1	CIDXR[4:0]=15	r/w	00h	13-14
GGCR	Game Control	P201AR+0	—	r/w	x0h	16-1
GJTDI	Joystick Trim DAC	P3XR+5	IGIDXR=4B	r/w	1Dh	16-1
GMCR	MIDI Control	P3XR+0	—	r/w	0x0x xxx0	16-2
GMRDR	MIDI Receive Data	P3XR+1	—	read	FFh	16-4
GMRFAI	MIDI Receive FIFO Access	P3XR+5	IGIDXR=5E	write	—	16-4
GMSR	MIDI Status	P3XR+0	—	read	0x00 xx10	16-3
GMTDR	MIDI Transmit Data	P3XR+1	—	write	—	16-4
I8DP	General 8-bit I/O Data Port	P3XR+5	—	r/w	—	12-12
I16DP	General 16-bit I/O Data Port	P3XR+(4-5)	—	r/w	—	12-12
ICMPTI	Compatibility	P3XR+5	IGIDXR=59	r/w	1Fh	12-15
IDECI	Decode Control	P3XR+5	IGIDXR=5A	r/w	7Fh	12-16
IEIRQI	Emulation IRQ	P3XR+5	IGIDXR=60	r/w	0?xx xx00	12-20
IEMUAI	MPU-401 Emulation Control A	P3XR+5	IGIDXR=5C	r/w	00h	12-18
IEMUBI	MPU-401 Emulation Control B	P3XR+5	IGIDXR=5D	r/w	30h	12-19
IGIDXR	General Index	P3XR+3	—	r/w	00h	12-12
IVERI	Version Number	P3XR+5	IGIDXR=5B	r/w	0010 0?00	12-17
LDIBI	LMC DMA Interleave Base (bits 23–8)	P3XR+(4-5)	IGIDXR=58	r/w	0000h	15-7
LDICI	LMC DMA Interleave Control	P3XR+(4-5)	IGIDXR=57	r/w	0000h	15-7
LDMACI	LMC DMA Control	P3XR+5	IGIDXR=41	r/w	00h	15-1
LDSAHI	LMC DMA Start Address High (bits 23–20, 3–0)	P3XR+5	IGIDXR=50	r/w	00h	15-3
LDSALI	LMC DMA Start Address Low (bits 19–4)	P3XR+(4-5)	IGIDXR=42	r/w	0000h	15-2
LMAHI	LMC I/O Address High (bits 23–16)	P3XR+5	IGIDXR=44	write	00h	15-3
LMALI	LMC I/O Address Low (bits 15–0)	P3XR+(4-5)	IGIDXR=43	write	0000h	15-3
LMBDR	LMC Byte Data	P3XR+7	—	r/w	—	15-1
LMCFI	LMC Configuration	P3XR+(4-5)	IGIDXR=52	r/w	0000h	15-4
LMCI	LMC Control	P3XR+5	IGIDXR=53	r/w	00h	15-5
LMFSI	LMC FIFO Size	P3XR+(4-5)	IGIDXR=56	r/w	0000h	15-6
LMPFAI	LMC Playback FIFO Base Address (bits 23–8)	P3XR+(4-5)	IGIDXR=54	r/w	0000h	15-6
LMRFAI	LMC Record FIFO Base Address (bits 23–8)	P3XR+(4-5)	IGIDXR=55	r/w	0000h	15-6
LMSBAI	LMC 16-Bit Access	P3XR+(4-5)	IGIDXR=51	r/w	—	15-4
P201HI	PNP P201AR[15:8]	PNPRDP, A79	LDN=2, PIDXR=60	r/w	00h	12-25
P201LI	PNP P201AR[7:0]	PNPRDP, A79	LDN=2, PIDXR=61	r/w	01h	12-25
P2X0HI	PNP P2XR[15:8]	PNPRDP, A79	LDN=0, PIDXR=60	r/w	00h	12-25
P2X0LI	PNP P2XR[7:0]	PNPRDP, A79	LDN=0, PIDXR=61	r/w	00h	12-25

Table 11-3 InterWave Registers and Ports by Mnemonic (Continued)

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page
P388HI	PNP P388AR[15:8]	PNPRDP, A79	LDN=3, PIDXR=60	r/w	00h	12-25
P388LI	PNP P388AR[7:0]	PNPRDP, A79	LDN=3, PIDXR=61	r/w	08h	12-25
P3X0HI	PNP P3XR[15:8]	PNPRDP, A79	LDN=0, PIDXR=62	r/w	00h	12-25
P3X0LI	PNP P3XR[7:0]	PNPRDP, A79	LDN=0, PIDXR=63	r/w	00h	12-25
P401HI	PNP P401AR[15:8]	PNPRDP, A79	LDN=4, PIDXR=60	r/w	00h	12-25
P401LI	PNP P401AR[7:0]	PNPRDP, A79	LDN=4, PIDXR=61	r/w	00h	12-25
PATAHI	PNP PATAAR[15:8]	PNPRDP, A79	LDN=1, PIDXR=62	r/w	00h	12-25
PATALI	PNP PATAAR[7:0]	PNPRDP, A79	LDN=1, PIDXR=63	r/w	00h	12-25
PCCCI	PNP Configuration Control Command	A79	PIDXR=02	write	—	12-22
PCSNBR	PNP Card Select Number Back Door	201	—	write	00h	12-21
PCSNI	PNP Card Select Number	PNPRDP, A79	PIDXR=06	r/w	00h	12-23
PGACTI	PNP Game Port Activate	PNPRDP, A79	LDN=2, PIDXR=30	r/w	00h	12-24
PGRCI	PNP Game Port I/O Range Check	PNPRDP, A79	LDN=2, PIDXR=31	r/w	00h	12-24
PHCAI	PNP PCODAR[15:8]	PNPRDP, A79	LDN=0, PIDXR=64	r/w	00h	12-25
PIDXR	PNP Index Address	279	—	write	00h	12-21
PISOCI	PNP Isolate Command	PNPRDP	PIDXR=01	read	—	12-22
PLCAI	PNP PCODAR [7:0]	PNPRDP, A79	LDN=0, PIDXR=65	r/w	00h	12-25
PLDNI	PNP Logical Device Number (LDN)	PNPRDP, A79	PIDXR=07	r/w	00h	12-23
PMACTI	PNP MPU-401 Activate	PNPRDP, A79	LDN=4, PIDXR=30	r/w	00h	12-24
PMISI	PNP MPU-401 Emulation IRQ Select	PNPRDP, A79	LDN=4, PIDXR=70	r/w	00h	12-25
PMITI	PNP MPU-401 Emulation IRQ Type	PNPRDP	LDN=4, PIDXR=71	read	02h	12-26
PMRCI	PNP MPU-401 I/O Range Check	PNPRDP, A79	LDN=4, PIDXR=31	r/w	00h	12-24
PNPRDP	PNP Read Data Port	PNPRDP	—	read	—	12-21
PNPWRP	PNP Write Data Port	A79	—	write	—	12-21
PPWRI	PNP Power Mode	PNPRDP, A79	LDN=0, PIDXR=F2	r/w	x111 1111	12-29
PRACTI	PNP CD-ROM Activate	PNPRDP, A79	LDN=1, PIDXR=30	r/w	00h	12-24
PRAHI	PNP PCDRAR[15:8]	PNPRDP, A79	LDN=1, PIDXR=60	r/w	00h	12-25
PRALI	PNP PCDRAR[7:0]	PNPRDP, A79	LDN=1, PIDXR=61	r/w	00h	12-25
PRDSI	PNP CD-ROM DMA Select	PNPRDP, A79	LDN=1, PIDXR=74	r/w	04h	12-27

Table 11-3 InterWave Registers and Ports by Mnemonic (Continued)

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page
PRESDI	PNP Resource Data	PNPRDP	PIDXR=04	read	00h	12-23
PRESSI	PNP Resource Data Status	PNPRDP	PIDXR=05	read	00h	12-23
PRISI	PNP CD-ROM IRQ Select	PNPRDP, A79	LDN=1, PIDXR=70	r/w	00h	12-25
PRITI	PNP CD-ROM IRQ Type	PNPRDP	LDN=1, PIDXR=71	read	02h	12-26
PRRCI	PNP CD-ROM I/O Range Check	PNPRDP, A79	LDN=1, PIDXR=31	r/w	00h	12-24
PSACTI	PNP AdLib–Sound Blaster Activate	PNPRDP, A79	LDN=3, PIDXR=30	r/w	00h	12-24
PSBISI	PNP AdLib–Sound Blaster Emulation IRQ Select	PNPRDP, A79	LDN=3, PIDXR=70	r/w	00h	12-25
PSBITI	PNP AdLib–Sound Blaster Emulation IRQ Type	PNPRDP	LDN=3, PIDXR=71	read	02h	12-26
PSECI	PNP Serial EEPROM Control	PNPRDP, A79	LDN=0, PIDXR=F1	r/w	xxxx 000x	12-28
PSEENI	PNP Serial EEPROM Enable	PNPRDP, A79	LDN=0, PIDXR=F0	r/w	00h	12-28
PSRCI	PNP AdLib–Sound Blaster I/O Range Check	PNPRDP, A79	LDN=3, PIDXR=31	r/w	00h	12-24
PSRPAI	PNP Set Read Data Port Address	A79	PIDXR=00	write	80h	12-22
PSRSTI	PNP Software Reset	PNPRDP, A79	LDN=0, PIDXR=F3	write	—	12-30
PUACTI	PNP Audio Activate	PNPRDP, A79	LDN=0, PIDXR=30	r/w	00h	12-24
PUD1SI	PNP Audio DMA Channel 1 Select	PNPRDP, A79	LDN=0, PIDXR=74	r/w	04h	12-27
PUD2SI	PNP Audio DMA Channel 2 Select	PNPRDP, A79	LDN=0, PIDXR=75	r/w	04h	12-27
PUI1SI	PNP Audio IRQ Channel 1 Select	PNPRDP, A79	LDN=0, PIDXR=70	r/w	00h	12-25
PUI1TI	PNP Audio IRQ Channel 1 Type	PNPRDP	LDN=0, PIDXR=71	read	02h	12-26
PUI2SI	PNP Audio IRQ Channel 2 Select	PNPRDP, A79	LDN=0, PIDXR=72	r/w	00h	12-25
PUI2TI	PNP Audio IRQ Channel 2 Type	PNPRDP	LDN=0, PIDXR=73	read	02h	12-26
PURCI	PNP Audio I/O Range Check	PNPRDP, A79	LDN=0, PIDXR=31	r/w	00h	12-24
PWAKEI	PNP Wake[CSN] Command	A79	PIDXR=03	write	—	12-23
SACI	Synth. Address Control (1 per voice)	P3XR+5	IGIDXR=0, 80	w, r	01h	14-8
SAEHI	Synth. Address End High (1 per voice)	P3XR+(4-5)	IGIDXR=4, 84	w, r	0000h	14-5
SAELI	Synth. Address End Low (1 per voice)	P3XR+(4-5)	IGIDXR=5, 85	w, r	0000h	14-5
SAHI	Synth. Address High (1 per voice)	P3XR+(4-5)	IGIDXR=A, 8A	w, r	0000h	14-6
SALI	Synth. Address Low (1 per voice)	P3XR+(4-5)	IGIDXR=B, 8B	w, r	0000h	14-6
SASHI	Synth. Address Start High (1 per voice)	P3XR+(4-5)	IGIDXR=2, 82	w, r	0000h	14-4
SASLI	Synth. Address Start Low (1 per voice)	P3XR+(4-5)	IGIDXR=3, 83	w, r	0000h	14-5
SAVI	Synth. Active Voices	P3XR+5	IGIDXR=E, 8E	w, r	CDh	14-1
SEAHI	Synth. Effects Address High (1 per voice)	P3XR+(4-5)	IGIDXR=11, 91	w, r	0000h	14-7
SEALI	Synth. Effects Address Low (1 per voice)	P3XR+(4-5)	IGIDXR=12, 92	w, r	0000h	14-7
SEASI	Synth. Effects Output Accumulator Select (1 per voice)	P3XR+5	IGIDXR=14, 94	w, r	00h	14-14
SEVFI	Synth. Effect Volume Final (1 per voice)	P3XR+(4-5)	IGIDXR=1D, 9D	w, r	0000h	14-14

Table 11-3 InterWave Registers and Ports by Mnemonic (Continued)

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page
SEVI	Synth. Effects Volume (1 per voice)	P3XR+(4-5)	IGIDXR=16, 96	w, r	0000h	14-13
SFCI	Synth. Frequency Control (1 per voice)	P3XR+(4-5)	IGIDXR=1, 81	w, r	0400h	14-7
SFLFOI	Synth. Frequency LFO (1 per voice)	P3XR+5	IGIDXR=17, 97	w, r	00h	14-8
SGMI	Synth. Global Mode	P3XR+5	IGIDXR=19, 99	w, r	00h	14-3
SLFOBI	Synth. LFO Base Address	P3XR+(4-5)	IGIDXR=1A, 9A	w, r	0000h	14-3
SLOFI	Synth. Left Offset Final (1 per voice)	P3XR+(4-5)	IGIDXR=1C, 9C	w, r	0000h	14-13
SLOI	Synth. Left Offset (1 per voice)	P3XR+(4-5)	IGIDXR=13, 93	w, r	0000h	14-13
SMSI	Synth. Mode Select (1 per voice)	P3XR+5	IGIDXR=15, 95	w, r	02h	14-14
SROFI	Synth. Right Offset Final (1 per voice)	P3XR+(4-5)	IGIDXR=1B, 9B	w, r	0700h	14-12
SROI	Synth. Right Offset (1 per voice)	P3XR+(4-5)	IGIDXR=C, 8C	w, r	0700h	14-12
SUAI	Synth. Upper Address (1 per voice)	P3XR+5	IGIDXR=10, 90	w, r	00h	14-4
SVCI	Synth. Volume Control (1 per voice)	P3XR+5	IGIDXR=D, 8D	w, r	01h	14-10
SVEI	Synth. Volume End (1 per voice)	P3XR+5	IGIDXR=8, 88	w, r	00h	14-9
SVII	Synth. Voices IRQ	P3XR+5	IGIDXR=8F	read	E0h	14-2
SVIRI	Synth. Voices IRQ Read	P3XR+5	IGIDXR=9F	read	E0h	14-2
SVLFOI	Synth. Volume LFO (1 per voice)	P3XR+5	IGIDXR=18, 98	w, r	00h	14-11
SVLI	Synth. Volume Level (1 per voice)	P3XR+(4-5)	IGIDXR=9, 89	w, r	0000h	14-10
SVRI	Synth. Volume Rate (1 per voice)	P3XR+5	IGIDXR=6, 86	w, r	00h	14-10
SVSI	Synth. Volume Start (1 per voice)	P3XR+5	IGIDXR=7, 87	w, r	00h	14-9
SVSR	Synth. Voice Select	P3XR+2	—	r/w	00h	14-1
U2X6R	Sound Blaster 2X6	P2XR+6	—	write	—	12-3
U2XCR	Sound Blaster 2XC (no IRQ)	P2XR+0Dh	—	write	00h	12-6
U2XER	Sound Blaster 2XE	P2XR+0Eh	—	r/w	00h	12-6
UACRR	AdLib Command Read	P2XR+0Ah	—	read	00h	12-3
UACWR	AdLib Command Write	P2XR+8, P388AR+0	—	write	00h	12-3
UADR	AdLib Data	P2XR+9, P389AR+1	—	r/w	00h	12-4
UASBCI	AdLib–Sound Blaster Control	P3XR+5	IGIDXR=45	r/w	00h	12-12
UASRR	AdLib Status Read	P2XR+8, P388AR+0	—	read	00h	12-3
UASWR	AdLib Status Write	P2XR+0Ah	—	write	00h	12-3
UAT1I	AdLib Timer 1	P3XR+5	IGIDXR=46	r/w	00h	12-13
UAT2I	AdLib Timer 2	P3XR+5	IGIDXR=47	r/w	00h	12-13
UCLRII	Clear Interrupt	P2XR+0Bh	URCR[2:0]=5	write	—	12-11
UDCI	DMA Channel Control	P2XR+0Bh	UMCR[6]=0, URCR[2:0]=0	r/w	00h	12-8
UGP1I	General Purpose Register 1 (Emulation Address)	UGPA1I		r/w	00h	12-10
UGP1I	General Purpose Register 1 (Back Door)	P2XR+0Bh	URCR[2:0]=1	r/w	00h	12-10
UGP2I	General Purpose Register 2 (Emulation Address)	UGPA2I		r/w	00h	12-10
UGP2I	General Purpose Register 2 (Back Door)	P2XR+0Bh	URCR[2:0]=2	r/w	00h	12-10
UGPA1I	General Purpose Register 1 Address	P2XR+0Bh	URCR[2:0]=3	r/w	00h	12-11
UGPA2I	General Purpose Register 2 Address	P2XR+0Bh	URCR[2:0]=4	r/w	00h	12-11
UHRDP	GUS Hidden Register Data Port	P2XR+0Bh	—	r/w	—	12-5
UI2XCR	Sound Blaster Interrupt 2XC	P2XR+0Ch	—	r/w	00h	12-5

Table 11-3 InterWave Registers and Ports by Mnemonic (Continued)

Mnem.	Description	I/O Address	Index	Read/ Write	Default Value	Page
UICI	Interrupt Control	P2XR+0Bh	UMCR[6]=1, URCR[2:0]=0	write	07h	12-9
UISR	IRQ Status	P2XR+6	—	read	00h	12-2
UJMPI	Jumper	P2XR+0Bh	URCR[2:0]=6	r/w	06h	12-11
UMCR	Mix Control	P2XR+0	—	r/w	03h	12-1
URCR	Register Control	P2XR+0Fh	—	r/w	00h	12-6
URSTI	GUS Reset	P3XR+5	IGIDXR=4C	write	xxxx x000	12-14
USCI	ADC Sample Control	P3XR+5	IGIDXR=49	r/w	00h	12-13
USRR	Status Read	P2XR+0Fh	—	read	01h	12-7

CHAPTER

12 SYSTEM CONTROL REGISTERS

P2XR Direct Registers

UMCR—Mix Control

Address: P2XR+0 read, write

Default: 03h

Note: For a description of how this register controls access to the hidden registers, see "IVERI—Version Number" on page 12-17.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Control Register Select	MIDI Loopback	Channel Synthesizer Interrupts	IRQ and DMA Enable	Enable Stereo Microphone Input	Enable Line Out	Enable Line In

Bit 7: Reserved.

Bit 6: **Control Register Select**. If the *Register Selector* field of the Register Control register (URCR[2:0]) is Low, this bit selects between indexing the Interrupt Control register (UICI) and the DMA Channel Control register (UDCI).
1:UICI
0:UDCI

Bit 5: **MIDI Loopback**. If set High, causes data from the MIDITX pin to be looped back to the MIDIRX pin. This setting does not block the transfer of data out of the MIDITX line; it does, however, block data reception through MIDIRX.

Bit 4: **Channel Synthesizer Interrupts**. If set High, causes the ORing of all the synthesizer and codec interrupts to the selected Channel 2 IRQ pin and the masking of synthesizer and codec interrupts to the selected Channel 1 IRQ pin.

Bit 3: **IRQ and DMA Enable**. If set High, enables the IRQ and DRQ pins (for audio functions only; does not affect the selected IRQ and DRQ lines for other logical device numbers). If set Low, forces all IRQ and DRQ pins into the high-impedance mode (for audio functions only).

Bit 2: **Enable Stereo Microphone Input**. If set Low, disables the stereo microphone inputs (no sound).

Bit 1: **Enable Line Out**. If set High, disables the stereo line-out outputs (no sound). This switch comes after all enables and attenuators in the codec module.

Bit 0: **Enable Line In**. If set High, disables the stereo line-in inputs (no sound). This switch comes before all enables and attenuators in the codec module.

UISR—IRQ Status

Address: P2XR+6 read

Default: 00h (after initialization)

This register specifies the cause of various interrupts.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DMA Terminal Count IRQ	Volume Loop IRQ	Address Loop IRQ	AdLib Register IRQ	AdLib Timer 2	AdLib Timer 1	MIDI Receive IRQ	MIDI Transmit IRQ

Bit 7: **DMA Terminal Count IRQ**. If set High, indicates that the ISA-bus terminal count signal, TC, has become active as a result of DMA activity between system and local memory. The flip-flop that drives this bit is cleared by a read of the LMC DMA Control register (LDMACI). This bit is ORed into the interrupt associated with the synthesizer. If the TC interrupt is not enabled (LDMACI[5]=0), then this bit is read as inactive even if the interrupt's flip-flop has been set.

Bit 6: **Volume Loop IRQ**. If set High, indicates that the volume ramp for one of the voices reached an end point. This bit is cleared after writing 8Fh to the General Index register (IGIDXR)—the value used to access the Synthesizer Voices IRQ register (SVII). This bit is enabled (but not cleared) by the *Synthesizer Interrupt Enable* field of the GUS Reset register (URSTI[2]).

Bit 5: **Address Loop IRQ**. If set High, indicates that the local memory address of one of the voices has reached an end point. This bit is cleared after writing 8Fh to the General Index register (IGIDXR)—the value used to access the Synthesizer Voices IRQ register (SVII). This bit is enabled (but not cleared) by the *Synthesizer Interrupt Enable* field of the GUS Reset register (URSTI[2]).

Bit 4: **AdLib–Sound Blaster Register IRQ**. The OR value of the following bits from the AdLib Status Read register: UASRR[0], UASRR[3], and UASRR[4]. The flip-flop that drives the AdLib Data register (UADR) interrupt is enabled when the *Enable Data Interrupt* field of the AdLib–Sound Blaster Control register (UASBCI[1]) is High, and is asynchronously cleared when UASBCI[1] is Low. The other two bits are enabled when the *Sound Blaster Interrupts Enable* field of the AdLib–Sound Blaster Control register (UASBCI[5]) is High, and are asynchronously cleared when UASBCI[5] is Low. This bit is ORed into the IRQ associated with AdLib–Sound Blaster emulation.

Bit 3: **AdLib Timer 2**. Is set High when AdLib Timer 2 rolls from FF to the preload value set in the AdLib Timer 2 register (UAT2I). This bit is cleared and disabled by the *Enable Interrupt for Timer 2* field of the AdLib–Sound Blaster Control register (UASBCI[3]). The flip-flop that drives this bit is ORed into the interrupt associated with AdLib–Sound Blaster and is also readable in the *Timer 2, Non-Maskable* field of the AdLib Status Read register (UASRR[1]).

Bit 2: **AdLib Timer 1**. Is set High when AdLib Timer 1 rolls from FF to the preload value set in the AdLib Timer 1 register (UAT1I). This bit is cleared and disabled when the *Enable Interrupt for Timer 1* field of the AdLib–Sound Blaster Control register (UASBCI[2]) is Low. The flip-flop that drives this bit is ORed into the interrupt associated with AdLib–Sound Blaster and is also readable in the *Timer 1, Non-Maskable* field of the AdLib Status Read register (UASRR[2]).

Bit 1: **MIDI Receive IRQ**. If set High, indicates the MIDI Receive Data register (GMRDR) contains data. This bit is cleared by reading GMRDR.

Bit 0: **MIDI Transmit IRQ**. If set High, indicates the MIDI Transmit Data register (GMTDR) is empty. This bit is cleared by writing to GMTDR.

U2X6R—Sound Blaster 2X6

Address: P2XR+6 write

A write to this address sets the *Write to 2X6 Interrupt* bit in the AdLib Status register (UASRR[3]) to High. No data is transferred or latched at this address.

UACRR, UACWR—AdLib Command Read/Write

Address: P2XR+0Ah read (UACRR); P2XR+08h and 388h write (UACWR)

Default: 00h

These registers emulate AdLib operation. UACRR is read and UACWR is written by AdLib application software to program the internal synthesizer to duplicate the AdLib sound.

UASRR, UASWR—AdLib Status Read/Write

Address: P2XR+08h and 388h read (UASRR); P2XR+0Ah write (UASWR)

Default: 00h

When the *Disable Auto-Timer Mode* field of the AdLib–Sound Blaster Control register (UASBCI[0]) is High (auto-timer mode disabled), this is a read-write register with different locations for the read and write addresses. When the *Disable Auto-Timer Mode* field of the AdLib–Sound Blaster Control register (UASBCI[0]) is Low (auto-timer mode enabled), writes to this register are latched but not readable. Reads of this register provide the following status information:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OR of Bits 5 and 6	Timer 1, Maskable	Timer 2, Maskable	Write to 2XC Interrupt	Write to 2X6 Interrupt	Timer 1, Non-maskable	Timer 2, Non-maskable	Data IRQ

Bit 7: **OR of Bits 5 and 6**. This bit represents the logical OR of bits 5 and 6 of this register.

Bit 6: **Timer 1, Maskable**. This bit is set High when AdLib Timer 1 rolls from FF to the preload value set in the AdLib Timer 1 register (UAT1I). This bit is cleared by setting the *AdLib IRQ Reset* bit of the AdLib Data register (UADR[7]). This bit does not become active if the *Mask Timer 1* bit of the AdLib Data register (UADR[6]) is High.

Bit 5: **Timer 2, Maskable**. This bit is set High when AdLib Timer 2 rolls from FF to the preload value set in the AdLib Timer 2 register (UAT2I). This bit is cleared by setting the *AdLib IRQ Reset* bit of the AdLib Data register (UADR[7]). This bit does not become active if the *Mask Timer 2* bit of the AdLib Data register (UADR[5]) is set.

Bit 4: **Write to 2XC Interrupt**. This bit is set High by a write to UI2XCR. The flip-flop driving this bit is enabled when the *Sound Blaster Interrupts Enable* bit of the AdLib–Sound Blaster Control register (UASBCI[5]) is High and asynchronously cleared when UASBCI[5] is Low.

Bit 3: **Write to 2X6 Interrupt**. This bit is set High by a write to U2X6R. The flip-flop driving this bit is enabled when the *Sound Blaster Interrupts Enable* bit of the AdLib–Sound Blaster Control register (UASBCI[5]) is High and asynchronously cleared when UASBCI[5] is Low.

Bit 2: **Timer 1, Non-maskable**. This bit is set High when AdLib Timer 1 rolls from FF to the preload value set in the AdLib Timer 1 register (UAT1I). It is cleared and disabled when the *Enable Interrupt for Timer 1* bit of the AdLib–Sound Blaster Control register (UASBCI[2]) is set Low. The flip-flop that drives this bit is ORed into the interrupt associated with AdLib–Sound Blaster emulation and is also readable in UISR[2].

Bit 1: **Timer 2, Non-maskable**. This bit is set High when AdLib Timer 2 rolls from FF to the preload value set in the AdLib Timer 2 register (UAT2I). It is cleared and disabled when the *Enable Interrupt for Timer 2* bit of the AdLib–Sound Blaster Control register (UASBCI[3]) is set Low. The flip-flop that drives this bit is ORed into the interrupt associated with AdLib–Sound Blaster emulation and is also readable in UISR[3].

Bit 0: **Data IRQ**. This bit is set High by a write to the AdLib Data register (UADR). The flip-flop that drives this bit is enabled when the *Enable Data Interrupt* bit of the AdLib–Sound Blaster Control register (UASBCI[1]) is High, and is asynchronously cleared when UASBCI[1] is Low. It is ORed into the interrupt associated with AdLib–Sound Blaster and is also readable in the *AdLib Register IRQ* bit of the IRQ Status register (UISR[4]).

UADR—AdLib Data

Address: P2XR+9 and 389h read, write

Default: 00h

This register performs AdLib functions based on the state of various bits as follows:

Table 12-1 AdLib Data (UADR) Function

Case	Condition	Result
1	$\overline{((UASBCI[0]=0) \bullet (UACWR=04h))}$	UADR behaves like a simple read/write register that is accessible through two different I/O addresses. Writes to UADR cause interrupts (see bit 4 of the IRQ Status (UISR) register).
2	$(UASBCI[0]=0) \bullet (UACWR=04h)$	Writes to UADR are disabled and no interrupt is generated; AdLib timer emulation functions are written instead of UADR. Reads provide the data that was last latched in case 1.

For case 2, the following AdLib timer emulation bits are written. All of these bits also default to Low after reset. Note that when the most significant bit is set High, the other bits do not change. Also, when IVERI[3] is active, the following bits are readable from this address, regardless of the state of the *Disable Auto-Timer Mode* bit of the AdLib–Sound Blaster Control register (UASBCI[0]) or the AdLib Command Write register (UACWR).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
AdLib IRQ Reset	Mask Timer 1	Mask Timer 2	Reserved	Reserved	Reserved	Start Timer 2	Start Timer 1

Bit 7: **AdLib IRQ Reset**. If set High, the flip-flops driving the *Timer 1, Maskable* and *Timer 2, Maskable* bits of the AdLib Status Read register (UASRR[6] and UASRR[5]) are cleared; this bit is automatically cleared after UASRR[6] and UASRR[5] are cleared. Also, when this bit is set High, the other four bits of this register are not altered. When this bit is set Low, the other bits of this register are latched.

Bit 6: **Mask Timer 1**. When High, the flip-flop that drives the *Timer 1, Maskable* bit of the AdLib Status Read register (UASRR[6]) is disabled from becoming active.

Bit 5: **Mask Timer 2**. When High, the flip-flop that drives the *Timer 2, Maskable* bit of the AdLib Status Read register (UASRR[5]) is disabled from becoming active.

Bits 4–2: Reserved.

Bit 1: **Start Timer 2**. When Low, the value found in the AdLib Timer 2 register (UAT2I) is loaded into AdLib Timer 2 with every 320-µs rising clock edge. When High, the timer increments with every 320-µs rising clock edge; on the next clock edge after the timer reaches FFh, UAT2I is again loaded into the timer.

Bit 0: **Start Timer 1**. When Low, the value found in the AdLib Timer 1 register (UAT1I) is loaded into AdLib Timer 1 with every 80-µs rising clock edge. When High, the timer increments with every 80-µs rising clock edge; on the next clock edge after the timer reaches FFh, UAT1I is again loaded into the timer.

UHRDP—GUS Hidden Register Data Port

Address: P2XR+0Bh read write

This port allows access to the hidden registers. For more details, see “UGP1I—General Purpose Register 1” on page 12-10. For information about the access protocol for registers based at this port, see the description of bit 0 in “IVERI—Version Number” on page 12-17.

UI2XCR—Sound Blaster IRQ 2XC

Address: P2XR+0Ch read, write

Default: 00h

Writes to this simple read/write register cause an interrupt. This register can also be written through the Sound Blaster 2XC register (U2XCR), which does not generate an interrupt. The interrupt is cleared by clearing the *Sound Blaster Interrupts Enable* bit of the AdLib–Sound Blaster Control register (UASBCI[5]).

U2XCR—Sound Blaster 2XC

Address: P2XR+0Dh write

Default: 00h (after initialization)

This register provides access to the Sound Blaster IRQ 2XC register (UI2XCR) without generating an interrupt.

U2XER—Sound Blaster 2XE

Address: P2XR+0Eh read, write

Default: 00h

This is a simple read-write register used for Sound Blaster emulation. If the *Enable U2XER Read Interrupts* bit of the Register Control register (URCR[7]) is High, then reads of this register cause interrupts.

URCR—Register Control

Address: P2XR+0Fh write, read (if IVERI[3] is active)

Default: 00h

Note: When the* Register Read Mode *bit of the Version Number register (IVERI[3]) is High, this register becomes readable; if IVERI[3] is Low, then reads from this address provide the data in the Status Read register (USRR).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Enable U2XER Read Interrupts	Enable General Purpose Register Access	Toggle UI2XCR[7]	General Purpose 2 Interrupt	General Purpose 1 Interrupt	Register Selector		

Bit 7: **Enable U2XER Read Interrupts**. If set High, enables an interrupt to be generated by reading the Sound Blaster 2XE register (U2XER). This interrupt is ORed with the interrupts associated with AdLib–Sound Blaster.

Bit 6: **Enable General Purpose Register Access**. If set High, enables access to the general purpose registers through the addresses specified in the *General Purpose Register 1 Address* and *General Purpose Register 2 Address* fields of the Compatibility register (ICMPTI[1:0] and ICMPTI[3:2]), and the General Purpose Register 1 Address and General Purpose Register 2 Address registers (UGPA1I and UGPA2I).

Bit 5: **Toggle UI2XCR[7]**. If set High, causes bit 7 of the Sound Blaster IRQ 2XC register (UI2XCR) to toggle with each I/O read of that register.

Bit 4: **General Purpose 2 Interrupt**. If set High, enables the interrupt caused by either a read or a write to General Purpose Register 2 through the address specified in the *General Purpose Register 2 Address* field of the Compatibility register (ICMPTI[3:2]) and the General Purpose Register 2 Address register (UGPA2I). The interrupt is logically ORed with the AdLib–Sound Blaster interrupt. Accesses to this register through the GUS Hidden Register Data Port (UHRDP) back door do not cause an interrupt.

Bit 3: **General Purpose 1 Interrupt**. If set High, enables the interrupt caused by either a read or a write to General Purpose Register 1 through the address specified in the *General Purpose Register 1 Address* field of the Compatibility register (ICMPTI[1:0]) and the General Purpose Register 1 Address register (UGPA1I). The interrupt is logically ORed with the AdLib–Sound Blaster interrupt. Accesses to this register through the GUS Hidden Register Data Port (UHRDP) back door do not cause an interrupt.

Bits 2–0: **Register Selector**. This field selects which register is accessed by writes to the GUS Hidden Register Data Port (UHRDP).
0:DMA Channel Control and Interrupt Control registers (UDCI and UICI)
1:General Purpose Register 1 Back Door register (UGP1I)
2:General Purpose Register 2 Back Door register (UGP2I)
3:General Purpose Register 1 Address register (UGPA1I)
4:General Purpose Register 2 Address register (UGPA2I)
5:Clear Interrupts register (UCLRII)
6:Jumper register (UJMPI)

USRR—Status Read

Address: P2XR+0Fh read

Default: 01h

This register provides the state of various interrupts. Clear all of these interrupts by a write to the Clear Interrupts register (UCLRII).

Note: When the* Register Read Mode *bit of the Version Number register (IVERI[3]) is High, the data in this register is not accessible.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
2XE Interrupt	General Purpose 2 Read Interrupt	General Purpose 2 Write Interrupt	General Purpose 1 Read Interrupt	General Purpose 1 Write Interrupt	Always Low	IRQ/DMA Enable Status	Always High

Bit 7: **2XE Interrupt**. If High, indicates that a read of the Sound Blaster 2XE register (U2XER) caused an interrupt.

Bit 6: **General Purpose 2 Read Interrupt**. If High, indicates that a read of the General Purpose 2 register through the address specified in the *General Purpose Register 2 Address* field of the Compatibility register (ICMPTI[3:2] and the General Purpose Register 2 Address register (UGPA2I) caused an interrupt.

Bit 5: **General Purpose 2 Write Interrupt**. If High, indicates that a write of the General Purpose 2 register through the address specified in ICMPTI[3:2] and UGPA2I caused an interrupt.

Bit 4: **General Purpose 1 Read Interrupt**. If High, indicates that a read of the General Purpose 1 register through the address specified in the *General Purpose Register 1 Address* field of the Compatibility register (ICMPTI[1:0] and the General Purpose Register 1 Address register (UGPA1I) caused an interrupt.

Bit 3: **General Purpose 1 Write Interrupt**. If High, indicates that a write of General Purpose 1 register through the address specified in ICMPTI[1:0] and UGPA1I caused an interrupt.

Bit 2: Always reads as Low. This bit cannot be written.

Bit 1: **IRQ/DMA Enable Status**. Provides the state of the *IRQ and DMA Enable* bit of the Mix Control register (UMCR[3]).

Bit 0: Always reads as High. This bit cannot be written.

URCR[2:0], UHRDP Indexed Registers

UDCI—DMA Channel Control

Address: P2XR+0Bh read, write; indexes UMCR[6]=0 and URCR[2:0]=0

Default: 00h

Writing to the PNP Audio DMA Channel 1 Select register (PUD1SI) modifies the *DMA Select Channel 1* field (UDCI[2:0]), and writing to the PNP Audio DMA Channel 2 Select register (PUD2SI) modifies the *DMA Select Channel 2* field (UDCI[5:3]). If the value written to PUD1SI or PUD2SI is not supported by UDCI, then the corresponding UDCI field is set to 0. The ability to alter these fields can be disabled by clearing the *Compatibility Enable* bit of the Compatibility register (ICMPTI[4]).

For information about restricting access to this register, see “IVERI—Version Number” on page 12-17.

Note: It is not legal to write to this register while the IC has any DMA activity enabled. Do not write to this register if any of the following bits are High:
– Enable GUS-Compatible DMA bit of the DMA Control register (LDMACI[0])
– Interleaved DMA Enable bit of the LMC DMA Interleave Control register (LDICI[9])
– Record Enable bit of the Configuration Register 1 (CFIG1I[1])
– Playback Enable bit of the Configuration Register 1 (CFIG1I[0])

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Extra Interrupt	Combine DMA Channels	DMA Select Channel 2			DMA Select Channel 1		

Bit 7: **Extra Interrupt**. When both interrupt sources are combined through the *Combine Interrupt Channels* field of the Interrupt Control register (UICI[6]), setting this bit High drives the IRQ line selected by the *Channel 2 IRQ Selection* field of the Interrupt Control register (UICI[5:3]).

Bit 6: **Combine DMA Channels**. If set High, combines both DMA channels using the channel selected in the *DMA Select Channel 1* field of this register (UDCI[2:0]).

Bits 5–3: **DMA Select Channel 2** (codec play)
0:no DMA
1:DRQ/DAK1 (8-bit)
2:DRQ/DAK3 (8-bit)
3:DRQ/DAK5 (16-bit)
4:DRQ/DAK6 (16-bit)
5:DRQ/DAK7 (16-bit)
6:DRQ/DAK0 (8-bit)

Bits 2–0: **DMA Select Channel 1** (system memory to local memory and codec record)
0:no DMA
1:DRQ/DAK1 (8-bit)
2:DRQ/DAK3 (8-bit)
3:DRQ/DAK5 (16-bit)
4:DRQ/DAK6 (16-bit)
5:DRQ/DAK7 (16-bit)
6:DRQ/DAK0 (8-bit)

UICI—Interrupt Control

Address: P2XR+0Bh read, write; indexes UMCR[6]=1 and URCR[2:0]=0

Default: 07h

Writing to the PNP Audio IRQ Channel 1 Select register (PUI1SI) modifies the *Channel 1 IRQ Selection* field (UICI[2:0]), and writing to the PNP Audio IRQ Channel 2 Select register (PUI2SI) modifies the *Channel 2 IRQ Selection* field (UICI[2:0]). If the value written to PUI1SI is not supported by UICI[2:0], then UICI[2:0] is set to 7. If the value written to PUI2SI is not supported by UICI[5:3], then UICI[5:3] is set to 0 (no IRQ). The ability to alter these fields can be disabled by clearing the *Compatibility Enable* bit of the Compatibility register (ICMPTI[4]).

For information about restricting access to this register, see “IVERI—Version Number” on page 12-17.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
AdLib–Sound Blaster to NMI	Combine Interrupt Channels	Channel 2 IRQ Selection			Channel 1 IRQ Selection		

Bit 7: **AdLib–Sound Blaster to NMI**. If set High, causes $\overline{\text{IOCHK}}$ (NMI) to be selected for interrupts associated with AdLib–Sound Blaster emulation and disables such interrupts from going to the IRQ selected in the *Channel 1 IRQ Selection* field (UICI[2:0]).

Bit 6: **Combine Interrupt Channels**. If set High, combines both interrupt sources to the IRQ selected in the *Channel 1 IRQ Selection* field (UICI[2:0]).

Bits 5–3: **Channel 2 IRQ Selection** (MIDI)
0:No Interrupt
1:IRQ9 (aka IRQ2)
2:IRQ5
3:IRQ3
4:IRQ7
5:IRQ11
6:IRQ12
7:IRQ15

Bits 2–0: **Channel 1 IRQ Selection** (codec, synthesizer, Sound Blaster, and AdLib)
0:$\overline{\text{IOCHK}}$
1:IRQ9 (aka IRQ2)
2:IRQ5
3:IRQ3
4:IRQ7
5:IRQ11
6:IRQ12
7:IRQ15

Note: The reference to IRQ2 in the GUS is changed to IRQ9. This change reflects the preference expressed in the PNP ISA specification; however, this IRQ is still physically connected to pin B04 of the ISA connector.

UGP1I—General Purpose Register 1

Address: P2XR+0Bh read, write; index URCR[2:0]=1

Default: 00h

The General Purpose 1 register consists of two 8-bit registers—UGP1I IN and UGP1I OUT—used for compatibility with other sound cards and with MPU-401. Each of these registers is accessed through both the Hidden Register Data Port (UHRDP) and the emulation address, which is specified in the *General Purpose Register 1 Address* field of the Compatibility register (ICMPTI[1:0]) and the General Purpose Register 1 Address register (UGPA1I). UGP1I IN is written through the emulation address and read through UHRDP. UGP1I OUT is read through the emulation address and written through UHRDP. Accessing these registers through the emulation address results in interrupts (if the interrupts are enabled).

For information about restricting access to this register, see “IVERI—Version Number” on page 12-17.

UGP2I—General Purpose Register 2

Address: P2XR+0Bh read, write; index URCR[2:0]=2

Default: 00h

The General Purpose 2 register consists of two 8-bit registers, UGP2I IN and UGP2I OUT, used for compatibility with other sound cards and with MPU-401. Each of these registers is accessed through both the Hidden Register Data Port (UHRDP) and the emulation address, which is specified in the *General Purpose Register 2 Address* field of the Compatibility register (ICMPTI[3:2]) and the General Purpose Register 2 Address register (UGPA2I). UGP2I IN is written through the emulation address and read through UHRDP. UGP2I OUT is read through the emulation address and written through UHRDP. Accessing these registers through the emulation address results in interrupts (if enabled).

For information about restricting access to this register, see “IVERI—Version Number” on page 12-17.

UGPA1I—General Purpose Register 1 Address

Address: P2XR+0Bh read, write; index URCR[2:0]=3

Default: 00h

This register controls the emulation address through which the General Purpose Register 1 (UGP1I) is accessed. The 8 bits written become bits 7–0 of the emulation address for UGP1I; emulation address bits 9–8 are specified in the *General Purpose Register 1 Address* field of the Compatibility register (ICMPTI[1:0]).

For information about restricting access to this register, see “IVERI—Version Number” on page 12-17.

UGPA2I—General Purpose Register 2 Address

Address: P2XR+0Bh read, write; index URCR[2:0]=4

Default: 00h

This register controls the emulation address through which the General Purpose Register 2 (UGP2I) is accessed. The 8 bits written become bits 7–0 of the emulation address for UGP2I; emulation address bits 9–8 are specified in the *General Purpose Register 2 Address* field of the Compatibility register (ICMPTI[3:2]).

For information about restricting access to this register, see “IVERI—Version Number” on page 12-17.

UCLRII—Clear Interrupt

Address: P2XR+0Bh write; index URCR[2:0]=5

Writing to this register causes all the interrupts described in the Status Read register (USRR) to be cleared.

For information about restricting access to this register, see “IVERI—Version Number” on page 12-17.

UJMPI—Jumper

Address: P2XR+0Bh read, write; index URCR[2:0]=6

Default: 06h

For information about restricting access to this register, see “IVERI—Version Number” on page 12-17.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Reserved	Reserved	Reserved	Enable Joystick	Enable MIDI	Reserved

Bits 7–3: Reserved.

Bit 2: **Enable Joystick**. If set High, enables reading and writing of the Game Control register (GGCR) (located at P201AR).

Bit 1: **Enable MIDI**. If set High, enables reading and writing of the following MIDI registers (located at P3XR+0 and P3XR+1):
–MIDI Control (GMCR)
–MIDI Status (GMSR)
–MIDI Transmit Data (GMTDR)
–MIDI Receive Data (GMRDR)

Bit 0: Reserved.

P3XR Direct Registers

IGIDXR—General Index

Address: P3XR+3, read, write

Default: 00h

This register specifies the indexed address to a variety of registers within the InterWave IC. The data ports associated with this index are I8DP and I16DP. When the *Auto Increment* field of the Synthesizer Voice Select register (SVSR[7]) is set High, the value in this register is incremented by one after every I/O write to either I8DP or I16DP (but not after 8-bit writes to the low byte of I16DP, P3XR+4).

I8DP, I16DP—General 8-Bit/16-Bit Data Port

Address: P3XR+5 for I8DP, P3XR+4-5h for I16DP, read, write

These data ports are used to access a variety of registers within the InterWave IC. The 8-bit I/O accesses to P3XR+5 are used to transfer 8-bit data. The 16-bit I/O accesses to P3XR+4 are used to transfer 16-bit data. It is also possible to transfer 16-bit data by using an 8-bit I/O access to P3XR+4 followed by an 8-bit access to P3XR+5 (in that order). The index associated with these ports is IGIDXR. When the *Auto Increment* field of the Synthesizer Voice Select register (SVSR[7]) is set High, the value in IGIDXR is incremented by one after every I/O write to either I8DP or I16DP (but not after 8-bit writes to the low byte of I16DP, P3XR+4).

IGIDXR, I8DP, and I16DP Indexed Registers

UASBCI—AdLib–Sound Blaster Control

Address: P3XR+5 read, write; index IGIDXR=45h

Default: 00h

This register is used to control the AdLib and Sound Blaster emulation hardware.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Sound Blaster Interrupts Enable	Enable Timer Test	Enable Interrupt for Timer 2	Enable Interrupt for Timer 1	Enable Data Interrupt	Disable Auto-Timer Mode

Bits 7–6: Reserved.

Bit 5: **Sound Blaster Interrupts Enable**. If set High, enables interrupts for writes to the Sound Blaster 2X6 (U2X6R) and Sound Blaster IRQ 2XC (UI2XCR) registers. If set Low, disables and asynchronously clears both of the interrupts.

Bit 4: **Enable Timer Test**. If set High, enables a high-speed clock to operate AdLib Timer 1 and AdLib Timer 2. If set Low, allows normal clocks to operate these timers. The frequency of the high-speed clock is 0.99614 MHz.

Bit 3: **Enable Interrupt For AdLib Timer 2**. If set High, enables the interrupt associated with AdLib Timer 2. If set Low, disables and asynchronously clears the interrupt.

Bit 2: **Enable Interrupt For AdLib Timer 1**. If set High, enables the interrupt associated with AdLib Timer 1. If set Low, disables and asynchronously clears the interrupt.

Bit 1: **Enable Data Interrupt**. If set High, enables the interrupt that results from a write to the AdLib Data register (UADR). If set Low, disables and asynchronously clears the interrupt.

Bit 0: **Disable Auto-Timer Mode**. If set Low, places the IC into auto-timer mode. If set High, disables auto-timer mode. For an explanation of auto-timer mode, see "UASRR, UASWR—AdLib Status Read/Write" on page 12-3 and "UADR—AdLib Data" on page 12-4.

UAT1I—AdLib Timer 1

Address: P3XR+5 read, write; index IGIDXR=46h

Default: 00h

This value is loaded into AdLib Timer 1 when one of the following conditions occurs:

- The *Start Timer 1* bit of the AdLib Data register (UADR[0]) is High and this timer increments past 0FFh
- UADR[0] is Low and there is a rising clock edge of this timer's 80-µs clock (16.9344 MHz divided by 1344)

Reading this register returns the preload value, not the actual state of the timer.

UAT2I—AdLib Timer 2

Address: P3XR+5 read, write; index IGIDXR=47h

Default: 00h

This value is loaded into AdLib Timer 2 when one of the following conditions occurs:

- The *Start Timer 2* bit of the AdLib Data register (UADR[1]) is High and this timer increments past 0FFh
- UADR[1] is Low and there is a rising clock edge of this timer's 320-µs clock (Timer 1's clock divided by 4)

Reading this register returns the preload values, not the actual state of the timer.

USCI—ADC Sample Control

Address: P3XR+5 read, write; index IGIDXR=49h

Default: 00h

This register exists for compatibility with the GUS hardware. Some software writes to and reads this index and expects bit 6 to be Low. The bits in this register are reset by the process described in "URSTI—GUS Reset" on page 12-14.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Read/Write Bit	Always Reads Low	Read/Write Bits					

Bits 7, 5–0: **Read/Write Bits**. Each of these bits returns what was most recently written to it and does not control anything.

Bit 6: **Always Reads Low**. This bit always reads back as Low. It cannot be set High.

URSTI—GUS Reset

Address: P3XR+5 read, write; index IGIDXR = 4Ch

Default: XXXX X000

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Reserved	Reserved	Reserved	Synthesizer Interrupt Enable	DAC Enable	Reset GUS

Bits 7–3: Reserved.

Bit 2: **Synthesizer Interrupt Enable**. If set High, enables the synthesizer's loop and volume interrupts, which are read in the IRQ Status register (UISR[6:5]). Disabling these interrupts with this bit does not clear the interrupts.

Bit 1: **DAC Enable**. If set High, enables the synthesizer digital-to-analog converter. If set Low, mutes the output of the synthesizer DAC.

Bit 0: **Reset GUS**. This bit allows software to reset the InterWave IC to GUS-compatible mode. To perform a reset, set this bit Low, wait for at least 22 µs, then set the bit High. The following items are reset by this process:

- Interrupt associated with a write to the Sound Blaster 2X6 register (U2X6R)
- Interrupt associated with a write to the Sound Blaster IRQ 2XC register (UI2XCR)
- Any DMA or I/O activity to local memory (including IOCHRDY)
- LMC DMA Control register (LDMACI)
- *DRAM/ROM Select* and *Auto Increment* bits of the LMC Control register (LMCI[1] and LMCI[0])
- LMC FIFO Size register (LMFSI)
- LMC DMA Interleave Control register (LDICI)
- Synthesizer Global Mode register (SGMI)
- Synthesizer LFO Base Address register (SLFOBI)
- Synthesizer Voices IRQ register (SVII)
- Synthesizer Voices IRQ Read register (SVIRI)

- Flip-flop that drives the *DMA Terminal Count IRQ* bit of the IRQ Status register (UISR[7])
- *Synthesizer Interrupt Enable* bit and *DAC Enable* bit of the GUS Reset register (URSTI[2:1])
- AdLib–Sound Blaster Control register (UASBCI)
- Interrupt associated with a write to the AdLib Data register (UADR)
- AdLib Data register (UADR)
- Flip-flops that drive the AdLib Status Read register (UASRR)
- ADC Sample Control register (USCI)

Also, while this bit is Low, the following conditions exist:

- It is not possible to write to the synthesizer's register array
- The synthesizer IRQs are all cleared
- All synthesizer state machines are prevented from operating; they stay frozen and no sound is generated.

This bit is fully controlled by software.

Note: This bit must remain Low for at least 22 µs after hardware and software resets have completed for the synthesizer register array to be properly initialized.

ICMPTI—Compatibility

Address: P3XR+5 read, write; index IGIDXR=59h

Default: 0001 1111 binary (1Fh)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Serial Transfer Mode			Compatibility Enable	General Purpose 2 Address Bits 9–8		General Purpose 1 Address Bits 9–8	

Bits 7–5: **Serial Transfer Mode**. These bits specify the mode of the serial transfer block of the codec module. This block is fully specified in the codec module. When ICMPTI[7] is High, the four external device pins ($\overline{\text{EX\_CS}}$, EX_IRQ, EX_DRQ, and $\overline{\text{EX\_DAK}}$) are switched to become the external serial port pins. Table 12-2 shows the possible modes.

Bit 4: **Compatibility Enable**. If set High, specifies that writes to the *DMA Select Channel* fields of the DMA Channel Control register (UDCI[5:0]) and the *Channel IRQ Selection* fields of the Interrupt Control register (UICI[5:0]) are allowed. If set Low, such writes are not allowed. These bits can also be altered by writes to the PNP Audio DMA Channel 1 Select (PUD1SI), PNP Audio DMA Channel 2 Select (PUD2SI), PNP Audio IRQ Channel 1 Select (PUI1SI), and PNP Audio IRQ Channel 1 Select (PUI2SI) registers, regardless of the state of this bit.

Bits 3–2: **General Purpose 2 Address Bits 9–8**. These bits specify ISA-address bits 9–8 to go with address bits 7–0 found in the relocatable General Purpose Register 2 Address register (UGPA2I).

Bits 1–0: **General Purpose 1 Address Bits 9–8**. These bits specify ISA-address bits 9–8 to go with address bits 7–0 found in the relocatable General Purpose Register 1 Address register (UGPA1I).

Table 12-2 Serial Transfer Mode Selection

Bits 2, 1, 0	Description
0 0 0	Disabled
0 0 1	Synthesizer DSP data to codec record FIFO input
0 1 0	Synthesizer DSP data to codec play FIFO input
0 1 1	Codec record FIFO output to codec play FIFO input
1 0 0	Synthesizer DSP data to external serial port pins
1 0 1	Codec record FIFO to external serial port output and external serial port input to codec playback FIFO
1 1 0	Not valid
1 1 1	Not valid

Note: It is not possible to use the serial transfer mode when the local memory record and playback FIFOs are in use. For details, see "LMFSI—LMC FIFO Size" on page 15-6. Also, to use the serial transfer mode, the application or driver software must always surround the enabling of codec playback and record and synthesizer play with the control of these bits. For example, these bits would be changed to 011 binary first, then the codec record and playback paths would be enabled; the codec record and playback paths would be disabled before changing these bits back to 000.

IDECI—Decode Control

Address: P3XR+5 read, write; index IGIDXR = 5Ah

Default: 7Fh

This register enables and disables the reading and writing of various address spaces.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Interrupt Associated w/Codec to Channel 2	Enable Interrupts on Channel 1	Enable Interrupts on Channel 2	Enable NMI Interrupts	Enable Decode of Codec	Enable Decodes of 388h and 389h	Enable Decodes of 2XE, 2XD, and 2XC	Enable Decodes of 2XA, 2X9, and 2X8

Bit 7: **Interrupt Associated With Codec To Channel 2**. If set High, the interrupt associated with the codec comes out on the Channel 2 IRQ pin and not on the Channel 1 IRQ pin. If set Low, this interrupt comes out on Channel 1.

Bit 6: **Enable Interrupts on Channel 1**. If set High, Channel 1 interrupts are enabled. If set Low, the selected Channel 1 IRQ output becomes high-impedance. If the IRQ selected by the *Channel 1 IRQ Selection* field of the Interrupt Control register (UICI[2:0]) is $\overline{\text{IOCHK}}$ (NMI), then this bit has no effect; however, IDECI[4] can be used to gate the NMI.

Bit 5: **Enable Interrupts on Channel 2**. If set High, Channel 2 interrupts are enabled. If set Low, the selected Channel 2 IRQ output becomes high-impedance.

Bit 4: **Enable NMI Interrupts**. If set High, $\overline{\text{IOCHK}}$ interrupts are enabled. If set Low, $\overline{\text{IOCHK}}$ becomes high-impedance.

Bit 3: **Enable Decode of Codec**. If set High, enables reading and writing of the codec registers—accessed through the PCODAR address space. If set Low, disables reading and writing of these registers.

Bit 2: **Enable Decodes of 388h and 389h**. If set High, enables reading and writing of the following addresses:
P388AR+0 write:AdLib Command Write register (UACWR)
P388AR+0 read:AdLib Status Read register (UASRR)
P388AR+ 1 r/w:AdLib Data register (UADR)

If set Low, disables reading and writing of these addresses.

Bit 1: **Enable Decodes of 2XE, 2XD, and 2XC**. If set High, enables reading and writing of the following addresses:
P2XR+Ch:Sound Blaster IRQ 2XC register (UI2XCR)
P2XR+Dh:Sound Blaster 2XC register (no IRQ) (U2XCR)
P2XR+Eh:Sound Blaster 2XE register (U2XER)

If set Low, disables reading and writing of these addresses.

Bit 0: **Enable Decodes of 2XA, 2X9, and 2X8**. If set High, enables reading and writing of the following addresses:
P2XR+8h write:AdLib Command Write register (UACWR)
P2XR+8h read:AdLib Status Read register (UASRR)
P2XR+9h:AdLib Data register (UADR)
P2XR+Ah read:AdLib Command Read register (UACRR)
P2XR+Ah write:AdLib Status Write register (UASWR)

If set Low, disables reading and writing of these addresses.

IVERI—Version Number

Address: P3XR+5 read, write; index IGIDXR = 5Bh

Default: 0001 0X00; see the description of bit 2 for its default value.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Version Number				Register Read Mode	Pull-Up Power	MPU-401 Emulation Mode	Hidden Register Lock Enable

Bits 7–4: **Version Number**. This field contains the version number of the IC die. Here are the possibilities:

- 0h = rev A silicon (A0, A1, and A2)
- 1h = rev B silicon

This field is read-only.

Bit 3: **Register Read Mode**. If set High, specifies that reads of three of the InterWave IC's normally unreadable registers return the data written to those registers:

- The AdLib Data register (UADR) returns bits 7–0 (with bits 4–2 all Low) regardless of the state of the *Disable Auto-Timer Mode* bit of the AdLib–Sound Blaster Compatibility register (UASBCI[0]) or the AdLib Command Write register (UACWR).
- The Register Control register (URCR) returns the data last written to it instead of the data in the Status Read register (USRR).
- The MIDI Control register (GMCR) returns the data last written to it instead of the data in the MIDI Status register (GMSR).

Bit 2: **Pull-Up Power**. If set Low, disables the power to the internal pull-up resistors on the signals IRQ15, $\overline{\text{IRQ12}}$, IRQ11, IRQ7, IRQ5, SA11–SA6, DRQ7–DRQ5, DRQ3, $\overline{\text{DAK7}}$–DAK5, and DAK3 so that these signals do not drive voltages onto the ISA bus during suspend mode, or add current load. If set High, enables the pull-up resistors on those signals. Normally, this bit should be left High for the 144-pin package and set Low for the 160-pin package. The default state of this bit is latched at the trailing edge of RESET by the state of the $\overline{\text{EX\_DAK}}$ pin. This bit is not reset by the software reset (PCCCI).

Bit 1: **MPU-401 Emulation Mode**. If set High, moves the MIDI Transmit Data register (GMTDR) and the MIDI Receive Data register (GMRDR) from P3XR+1 to P3XR+0, and the MIDI Control register (GMCR) and MIDI Status register (GMSR) from P3XR+0 to P3XR+1.

Bit 0: **Hidden Register Lock Enable**. If set High (inactive), unconditionally enables accesses to registers through the GUS Hidden Register Data Port (UHRDP). If set Low (active), access to registers through UHRDP must conform to a protocol. The protocol is initiated by a write to the Mix Control register (UMCR) which enables the next subsequent I/O access to the hidden registers through UHRDP. An I/O read or write (while AEN is Low) to any address except P2XR+0 (UMCR) or P2XR+0Bh (UHRDP) locks out further I/O accesses to the hidden registers.

IEMUAI—MPU-401 Emulation Control A

Address: P3XR+5 read, write; index IGIDXR=5Ch

Default: 00h

The *emulation addresses* described in the following bit definitions are the addresses specified in the *General Purpose Register 1 Address* and *General Purpose Register 1 Address* fields of the Compatibility register (ICMPTI[1:0] and ICMPTI[3:2]), the General Purpose Register 1 Address register (UGPA1I), and the General Purpose Register 2 Address register (UGPA2I).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
UART Receive Buffer Read Enable	UART Status Read Enable	Emulation Address 2 Read Enable	Emulation Address 1 Read Enable	UART Transmit Buffer Write Enable	UART Command Buffer Write Enable	Emulation Address 2 Write Enable	Emulation Address 1 Write Enable

Bit 7: **UART Receive Buffer Read Enable**. If set Low, allows reading of the MIDI Receive Data register (GMRDR). If set High, reads of that register are ignored internally (although the ISA data bus is still driven).

Bit 6: **UART Status Read Enable**. If set Low, allows reading of the MIDI Status register (GMSR). If set High, reads of that register are ignored internally (although the ISA data bus is still driven).

Bit 5: **Emulation Address 2 Read Enable**. If set Low, allows reading of the General Purpose Register 2 (UGP2I) through the emulation address. If set High, reads of UGP2I through the emulation address are ignored internally (although the ISA data bus is still driven).

Bit 4: **Emulation Address 1 Read Enable**. If set Low, allows reading of the General Purpose Register 1 (UGP1I) through the emulation address. If set High, reads of UGP1I through the emulation address are ignored internally (although the ISA data bus is still driven).

Bit 3: **UART Transmit Buffer Write Enable**. If set Low, allows writing to the MIDI Transmit Data register (GMTDR). If set High, writes to that register are ignored by the UART.

Bit 2: **UART Command Buffer Write Enable**. If set Low, allows writing to the MIDI Control register (GMCR). When set High, writes to that register are ignored by the UART.

Bit 1: **Emulation Address 2 Write Enable**. If set Low, allows writing to the General Purpose 2 back door register (UGP2I) through the emulation address. If set High, UGP2I does not change during writes to the emulation address.

Bit 0: **Emulation Address 1 Write Enable**. If set Low, allows writing to the General Purpose 1 back door register (UGP1I) through the emulation address. If set High, UGP1I does not change during writes to the emulation address.

IEMUBI—MPU-401 Emulation Control B

Address: P3XR+5 read, write; index IGIDXR=5Dh

Default: 30h

The *emulation addresses* described in the following bit definitions are the addresses specified in the *General Purpose Register 1 Address* and *General Purpose Register 1 Address* fields of the Compatibility register (ICMPTI[1:0] and ICMPTI[3:2]), the General Purpose Register 1 Address register (UGPA1I), and the General Purpose Register 2 Address
register (UGPA2I).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
MIDI Receive Data Enable	MIDI Transmit Data Enable	Select Status Emulation Register 1 Bit 7 for I/O Reads	Select Status Emulation Register 1 Bit 6 for I/O Reads	Emulation Register 2 Write Interrupt Enable	Emulation Register 1 Write Interrupt Enable	Emulation Register 2 Read Interrupt Enable	Emulation Register 1 Read Interrupt Enable

Bit 7: **MIDI Receive Data Enable**. If set Low, allows MIDI receive data from the MIDIRX pin to pass into the UART. If set High, disables the data path from MIDIRX to the UART.

Bit 6: **MIDI Transmit Data Enable**. If set Low, allows MIDI transmit data from the UART to pass to the MIDITX pin. If set High, disables the data path from MIDITX to the UART.

Bit 5: **Select Status Emulation Register 1 Bit 7 for I/O Reads**. If set High, enables bit 7 of the General Purpose Register 1 (UGP1I OUT[7]) onto the data bus during reads of UGP1I OUT through the emulation address. Setting this bit Low enables the Data Send Ready signal ($\overline{\text{DSR}}$) onto bit 7 of the data bus during those reads. $\overline{\text{DSR}}$ is set High (inactive) by the hardware when the General Purpose Register 2 register (UGP2I OUT) is read through the emulation address, if the *Emulation Address 2 Read Enable* bit of the MPU-401 Emulation Control A register (IEMUAI[5]) is Low. This flag is also controlled by writes to UGP1I OUT[7] through the GUS Hidden Register Data Port (UHRDP).

Bit 4: **Select Status Emulation Register 1 Bit 6 for I/O Reads**. If set High, enables bit 6 of the General Purpose Register 1 (UGP1I OUT[6]) onto the data bus during reads of UGP1I OUT through the emulation address. Setting this bit Low enables Data Receive Ready ($\overline{\text{DRR}}$) onto bit 6 of the data bus during those reads. $\overline{\text{DRR}}$ is set High (inactive) by the hardware when there is a write to either the General Purpose Register 1 (UGP1I IN) or General Purpose Register 2 (UGP2I IN) through the emulation address, if the *Emulation Address 1 Write Enable* bit (for UGP1I) or the *Emulation Address 2 Write Enable* bit (for UGP2I) of the MPU-401 Emulation Control A register (IEMUAI[0] or IEMUAI[1]) is Low. $\overline{\text{DRR}}$ is also controlled by writes to UGP1I OUT[6] through the GUS Hidden Register Data Port (UHRDP).

Bit 3: **Emulation Register 2 Write Interrupt Enable**. If set Low, writing to UGP2I through the emulation address causes an interrupt. If set High, writing to UGP2I does not cause an interrupt.

Bit 2: **Emulation Register 1 Write Interrupt Enable**. If set Low, writing to UGP1I through the emulation address causes an interrupt. If set High, writing to UGP1I does not cause an interrupt.

Bit 1: **Emulation Register 2 Read Interrupt Enable**. If set Low, reading UGP2I through the emulation address causes an interrupt. If set High, reading UGP2I does not cause an interrupt.

Bit 0: **Emulation Register 1 Read Interrupt Enable**. If set Low, reading UGP1I through the emulation address causes an interrupt. If set High, reading UGP1I does not cause an interrupt.

IEIRQI—Emulation IRQ

Address: P3XR+5 read, write; index IGIDXR = 60h

Default: 0?XX XX00; see the description of bit 6 for its default value

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Select GPOUT1–GP OUT2 Codec Flags	16-Bit I/O Decoding Selected	Reserved				IRQ MPU-401	IRQ Sound Blaster

Bit 7: **Select GPOUT1–GPOUT0 Codec Flags**. If set High, the pins GPOUT1–GPOUT0 are selected. If Low, the pins are IRQ10 and IRQ4.

Bit 6: **16-Bit I/O Decoding Selected**. This read-only bit reflects the state of the internal pin IP16BITIO, which is latched off of $\overline{\text{EX\_CS}}$ at the trailing edge of RESET. When Low, system address bit 11 and bit 10 (SA11–SA10) are ignored for most address blocks. When High, SA11–SA10 must be Low to decode most address blocks.

Bits 5–2: Reserved.

Bit 1: **IRQ MPU-401**. This bit controls the state of the IRQ line selected by the PNP MPU-401 Emulation IRQ Select register (PMISI). If set High, the IRQ line becomes High; if set Low, the IRQ line becomes Low.

Bit 0: **IRQ Sound Blaster**. This bit controls the state of the IRQ line selected by the PNP AdLib–Sound Blaster Emulation IRQ Select register (PSBISI). If set High, the IRQ line becomes High; if set Low, the IRQ line becomes Low.

PNP Direct Registers

PCSNBR—PNP Card Select Number Back Door

Address: 0201h write

Default: 00h

It is possible to write a card select number (CSN) to the InterWave IC through this I/O port if all of the following conditions exist:

- The InterWave IC is in PNP system mode (latched by the state of the PNPCS pin at the end of reset).
- The audio logical device has not been activated (bit 0 of the PNP Audio Activate register (PUACTI[0]) is Low).
- The IC is in the PNP isolation state.

PIDXR—PNP Index Address

Address: 0279h write

Default: 00h

This 8-bit index address register points to standard Plug and Play registers.

PNPWRP—PNP Write Data Port

Address: 0A79h write

Write to the Plug and Play ISA registers through this port, indexed by PIDXR.

PNPRDP—PNP Read Data Port

Address: Address is relocatable between 003h and 3FFh, read only. The address is set by writing 00h (the index value for the PNP Set Read Data Port Address register (PSRPAI)) to the PNP Index Address register (PIDXR), and then writing a byte that represents bits 9–2 of the address to the PNP Data Write Port (PNPWRP). Bits 1–0 are always assumed to be High.

Read Plug and Play ISA registers through this port, indexed by PIDXR.

PIDXR, PNPWRP, and PNPRDP PNP Indexed Registers

These PNP registers are indexed with PIDXR and accessed through PNPRDP and PNWRP. Many of the registers—PIDXR=30h and greater—are further indexed by the Logical Device Number register (PLDNI). All such registers can be accessed only when the IC is in the PNP configuration state.

PSRPAI—PNP Set Read Data Port Address

Address: 0A79h write; index PIDXR=0

Default: 80h

Writing to this register sets up bits 9–2 of the address of the PNP Read Data Port (PNPRDP). Bits 1–0 of the address are both assumed to be High. Writing to this register is allowed only when the IC is in the PNP isolation state.

PISOCI—PNP Isolate Command

Address: PNPRDP read; index PIDXR=1

Reading this register causes a specific value—based on data read out of the PNP serial EEPROM—to be driven onto the ISA bus and then to be read back to check for a difference. This process can result in a *lose-isolation* condition and cause the IC to enter the PNP sleep state. If the IC is in PNP system mode, then it is assumed that there is no serial EEPROM and no data will ever be driven on the bus for reads from this register. In PNP system mode, reads of PISOCI always cause the InterWave IC to lose isolation and go into the sleep state. Reading this register is allowed only when the IC is in the PNP isolation state.

PCCCI—PNP Configuration Control Command

Address: 0A79h write; index PIDXR=2

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Reserved	Reserved	Reserved	Reset CSN	Wait-for-Key	Reset

Bits 7–3: Reserved.

Bit 2: **Reset CSN**. If the IC is in the PNP sleep, isolation, or configuration state, setting this bit High causes the CSN to be set to zero. This command is ignored if the IC is in the PNP wait-for-key state.

Bit 1: **Wait for Key**. If set High, the IC enters the PNP wait-for-key state. This command is ignored if the IC is already in the PNP wait-for-key state, but is valid for the other three states.

Bit 0: **Reset**. If set High, the InterWave IC resets. This results in a 3-ms to 10-ms pulse over the general reset line to the whole IC. The only areas not reset by this command are the PNP Set Read Data Port Address register (PSRPAI), the PNP Card Select Number register (PCSNI), and the PNP state. This command is ignored if the IC is in the PNP wait-for-key state, but is valid for the other three states.

PWAKEI—PNP Wake[CSN] Command

Address: 0A79h write; index PIDXR=3

Writes to this register affect the PNP state based on the state of the PNP Card Select Number register (PCSNI) and the data written. If the data is 00h and the CSN is 00h, then the IC enters the PNP isolation state. If the data is not 00h and the CSN matches the data, then the IC enters the PNP configuration state. If the data does not match the CSN, then the IC enters the PNP sleep state. This command also resets the serial EEPROM control logic that contains the address to that part. This command is ignored if the IC is in the PNP wait-for-key state, but it is valid for the other three states.

PRESDI—PNP Resource Data

Address: PNPRDP read; index PIDXR=4

Default: 00h

PNP software reads this register when it obtains resources from the PNP serial EEPROM.

Note: If the serial EEPROM has been placed into direct-control mode—the* Serial EEPROM Mode *bit of the PNP Serial EEPROM Enable register (PSEENI[0]) set High—then the wake command must be executed before access through the PNP Resource Data register (PRESDI) is possible. This command is valid only when the IC is in the PNP configuration state.

PRESSI—PNP Resource Data Status

Address: PNPRDP read; index PIDXR=5

Default: 00h

If bit 0 of this register is High, then the next byte of PNP resource data is available to be read; all other bits are reserved. After the PNP Resource Data register (PRESDI) is read, this bit becomes cleared until the next byte is available. This register can be read only when the IC is in the PNP configuration state.

PCSNI—PNP Card Select Number

Address: PNPRDP read, 0A79h write; index PIDXR=6

Default: 00h

While the IC is in the PNP isolation state, writing to this register sets up the CSN for the IC and sends the IC into the PNP configuration state. When the IC is in the configuration state, this register can be read but not written.

PLDNI—PNP Logical Device Number

Address: 0A79h write, PNPRDP read; index PIDXR=7

Default: 00h

This register further indexes the PNP address space into logical devices. The InterWave IC contains the following logical devices with corresponding logical device numbers (LDNs):

audio 00h—all audio functions, synthesizer, codec, and ports

external device
01h—the external device interface (e.g., a CD-ROM interface to the four EX_xxx pins)

game port 02h—game port (P201AR only)

AdLib–Sound Blaster emulation

03h—AdLib–Sound Blaster emulation IRQ and I/O addresses

MPU-401 emulation

04h—MPU-401 emulation IRQ and I/O addresses

This register can be accessed only when the IC is in the PNP configuration state.

PNP Unimplemented Registers

Reads of all PNP indexed addresses from PIDXR=08h through 2Fh return all zeros, per the PNP specification.

PUACTI, PRACTI, PGACTI, PSACTI, PMACTI—PNP Activate Registers

Address: 0A79h write, PNPRDP read; indexes PIDXR = 30h and PLDNI = one of the following:
0:PUACTI (audio)
1:PRACTI (external)
2:PGACTI (game port)
3:PSACTI (AdLib–Sound Blaster)
4:PMACTI (MPU-401)

Default: 00h

Setting bit 0 High activates the specified logical device; all other bits are reserved. If set Low, none of the specified logical device's address spaces are decoded and the interrupt and DMA channels are not enabled.

PURCI, PRRCI, PGRCI, PSRCI, PMRCI—PNP I/O Range Check Registers

Address: 0A79h write, PNPRDP read; indexes PIDXR=31h and PLDNI= one of the following:
0:PURCI (audio)
1:PRRCI (external)
2:PGRCI (game port)
3:PSRCI (AdLib–Sound Blaster)
4:PMRCI (MPU-401)

Default: 00h

Note: These registers are not available when the IC is in external decoding mode. These registers are disabled when bit 0 of any of the corresponding activate registers (PUACTI, PRACTI, PGACTI, PSACTI, PMACTI) is set High.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Range Check Enable	High 55h/ Low AAh

Bits 7–2: Reserved.

Bit 1: **Range Check Enable**. If set High, causes reads of all of the specified logical device address spaces to drive either 55h or AAh based on the state of the *High 55h/Low AAh* field. This bit functions only when the logical device is not activated.

Bit 0: **High 55h/Low AAh**. When the *Range Check Enable* bit is High (PURCI[1]), this bit selects the data value that is driven back onto the ISA data bus during a read. Setting this bit High specifies that 55h be driven; Low specifies AAh.

PNP Address Control Registers

Table 12-3 lists the PNP registers that control the address of blocks of I/O space within the InterWave IC.

Table 12-3 PNP Address Control Registers

Mnemonic	Index	LDN	Default	Description
P2X0HI	60h	0	00h	P2X0HI[1:0] specifies P2XR[9:8]
P2X0LI	61h	0	00h	P2X0LI[7:4] specifies P2XR[7:4]
P3X0HI	62h	0	00h	P3X0HI[1:0] specifies P3XR[9:8]
P3X0LI	63h	0	00h	P3X0LI[7:3] specifies P3XR[7:3]
PHCAI	64h	0	00h	PHCAI[1:0] specifies PCODAR[9:8]
PLCAI	65h	0	00h	PLCAI[7:2] specifies PCODAR[7:2]
PRAHI	60h	1	00h	PRAHI[1:0] specifies PCDRAR[9:8]
PRALI	61h	1	00h	PRALI[7:4] specifies PCDRAR[7:4]
PATAHI	62h	1	00h	PATAHI[1:0] specifies PATAAR[9:8]
PATALI	63h	1	00h	PATALI[7:1] specifies PATAAR[7:1]
P201HI	60h	2	00h	P201HI[1:0] specifies P201AR[9:8]
P201LI	61h	2	01h*	P201LI[7:6] specifies P201AR[7:6]
P388HI	60h	3	00h	P388HI[1:0] specifies P388AR[9:8]
P388LI	61h	3	08h*	P388LI[7:6] specifies P388AR[7:6]
P401HI	60h	4	00h	P401HI[1:0] specifies P401AR[9:8]
P401LI	61h	4	00h	P401LI[7:1] specifies P401AR[7:1]

Notes:

1. If P201AR[9:6] = 0h, then the value read back from P201LI is 00h (not 01h as shown).

2. If P388AR[9:6] = 0h, then the value read back from P388LI is 00h (not 08h as shown).

All unused bits in the above PNP address control registers are reserved. All of the PNP address control registers shown in Table 12-3 are written through 0A79h (PNPWRP) and read through the PNP Read Data Port (PNPRDP). The value read back in the unspecified bits of all of the above registers is shown in the default column. For a description of the functions controlled by the various address blocks, see “I/O Address Spaces” on page 4-2.

All of the groups of registers in Table 12-3 are used to describe an I/O address. When the programmable bits of any of these registers are zero (which is the power-up default), the addresses they describe are not decoded. Leaving the bits at zero is the PNP method of specifying that the range is not valid.

PUI1SI, PUI2SI, PRISI, PSBISI, PMISI—PNP IRQ Select Registers

Address: 0A79h write, PNPRDP read; see Table 12-4 for indexes

Default: 00h

Bits 3–0 select the IRQ number for the specified logical device interrupts as shown in Table 12-5. The mapping of IRQ number to interrupt event for each register is shown in Table 12-6.

IRQ10 and IRQ4 are not available unless selected. For details, see the description of bit 7 in “IEIRQI—Emulation IRQ” on page 12-20.

Bits 7–4 of each of these registers are reserved.

Writes to PUI1SI appropriately affect the *Channel 1 IRQ Selection* field of the Interrupt Control register (UICI[2:0]). If a value is written to PUI1SI that is not supported by UICI[2:0], then UICI[2:0] is set to 7 (IRQ15). Writes to PUI2SI appropriately affect the *Channel 2 IRQ Selection* field of the Interrupt Control register (UICI[5:3]). If a value is written to PUI2SI that is not supported by UICI[5:3], then UICI[5:3] is set to 0 (no IRQ).

Table 12-4 Indexes for PNP IRQ Select Registers

Mnemonic	PIDXR	PLDNI	Register Name
PUI1SI	70h	00h	PNP Audio IRQ Channel 1 Select
PUI2SI	72h	00h	PNP Audio IRQ Channel 2 Select
PRISI	70h	01h	PNP CD-ROM (External Function) IRQ Select
PSBISI	70h	03h	PNP AdLib–Sound Blaster Emulation IRQ Select
PMISI	70h	04h	PNP MPU-401 Emulation IRQ Select

Table 12-5 IRQ Number Selection

[3:0]	Description	[3:0]	Description	[3:0]	Description	[3:0]	Description
0h	No IRQ	4h	IRQ4	8h	No IRQ	0Ch	IRQ12
1h	No IRQ	5h	IRQ5	9h	IRQ2/9	0Dh	No IRQ
2h	IRQ2/9	6h	No IRQ	0Ah	IRQ10	0Eh	No IRQ
3h	IRQ3	7h	IRQ7	0Bh	IRQ11	0Fh	IRQ15

Table 12-6 IRQ Number to Interrupt Event Mapping for IRQ Select Registers

Mnemonic	Selects the IRQ number for the following interrupts
PUI1SI	Audio Channel 1 IRQs; normally passes along synthesizer, codec, and various miscellaneous IRQs
PUI2SI	Audio Channel 2 IRQs; normally passes along the MIDI IRQs; not used in many configurations
PRISI	External function (e.g., CD-ROM) IRQ. This IRQ is sourced from the EX_IRQ pin.
PSBISI	AdLib–Sound Blaster emulation IRQ. This IRQ is fully controlled by the *IRQ Sound Blaster* bit of the Emulation IRQ register (IEIRQI[0]).
PMISI	MPU-401 emulation IRQ. This IRQ is fully controlled by the *IRQ MPU-401* bit of the Emulation IRQ register (IEIRQI[1]).

PUI1TI, PUI2TI, PRITI, PSBITI, PMITI—PNP IRQ Type Registers

Address: PNPRDP read; see Table 12-4 for indexes

Default: 02h

These registers provide data back to standard PNP software concerning the type of interrupts supported by the InterWave IC. Each register always reads as 02h to indicate edge-triggered, active-high interrupts.

Table 12-7 Indexes for PNP IRQ Type Registers

Mnemonic	PIDXR	PLDNI	Register Name
PUI1TI	71h	00h	PNP Audio IRQ Channel 1 Type
PUI2TI	73h	00h	PNP Audio IRQ Channel 2 Type
PRITI	71h	01h	PNP CD-ROM (External Function) IRQ Type
PSBITI	71h	03h	PNP AdLib–Sound Blaster Emulation IRQ Type
PMITI	71h	04h	PNP MPU-401 Emulation IRQ Type

PUD1SI, PUD2SI, PRDSI—PNP DMA Select Registers

Address: 0A79h write, PNPRDP read; see Table 12-4 for indexes

Default: 04h

Bits 2–0 of these registers select the DMA request-acknowledge number as shown in Table 12-9.

The DMA functions for which the DMA request-acknowledge number is selected by each register are:

PUD1SI: DMA Channel 1—local–system memory transfers or codec record

PUD2SI: DMA Channel 2—codec playback

PRDSI: External Device—passes EX_DRQ to the selected DRQ pin and selects a $\overline{\text{DAK}}$ pin to control $\overline{\text{EX\_DAK}}$

Bits 7–3 are reserved.

Writes to PUD1SI appropriately affect the *DMA Select Channel 1* field of the DMA Channel Control register (UDCI[2:0]). Writes to PUD2SI appropriately affect the *DMA Select Channel 2* field of the DMA Channel Control register (UDCI[5:3]). If a value is written to either of these registers that is not supported by UDCI, then the corresponding UDCI field is set to 0 (no DMA).

Note: It is not legal to write to this register while the IC has any DMA activity enabled. Do not write to this register if any of the following bits are High:
– Enable GUS-Compatible DMA *bit of the DMA Control register (LDMACI[0])*
– Interleaved DMA Enable *bit of the LMC DMA Interleave Control register (LDICI[9])*
– Record Enable *bit of the Configuration Register 1 (CFIG1I[1])*
– Playback Enable *bit of the Configuration Register 1 (CFIG1I[0])*

Table 12-8 Indexes for PNP DMA Select Registers

Mnemonic	PIDXR	PLDNI	Register Name
PUD1SI	74h	00h	PNP Audio DMA Channel 1 Select
PUD2SI	75h	00h	PNP Audio DMA Channel 2 Select
PRDSI	74h	01h	PNP CD-ROM (External Function) DMA Select

Table 12-9 DMA Request Number Selection

[2:0]	Description	[2:0]	Description
0h	DRQ/AK0	4h	No DMA
1h	DRQ/AK1	5h	DRQ/AK5
2h	No DMA	6h	DRQ/AK6
3h	DRQ/AK3	7h	DRQ/AK7

PSEENI—PNP Serial EEPROM Enable

Address: 0A79h write, PNPRDP read; index PIDXR=F0h and PLDNI=0

Default: 00h

This register is accessible only when the PNP state machine is in the configuration state.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	ISA Data Bus Drive	Serial EEPROM Mode

Bits 7–2: Reserved.

Bit 1: **ISA Data Bus Drive**. This bit specifies the output-low drive capability, Iol, of the SD15–SD0, IOCHRDY, $\overline{\text{IOCS16}}$, and $\overline{\text{IOCHK}}$ pins. At 5 V: 00=24 mA, 01=12mA, 10=3mA, 11=reserved. At 3.3 V, the drive is at least 3 mA for bits 2–1 = 00, 01, and 10.

Bit 0: **Serial EEPROM Mode**. If set Low, specifies that the serial EEPROM interface circuitry is in initialization mode, whereby the data transfer is controlled by the PNP state machine. If set High, the serial EEPROM is controlled directly by the PNP Serial EEPROM Control register (PSECI).

PSECI—PNP Serial EEPROM Control

Address: 0A79h write, PNPRDP read; index PIDXR=F1h and PLDNI=0

Default: XXXX 000X

If the InterWave IC is in control mode—the *Serial EEPROM Mode* bit of the PNP Serial EEPROM Enable register (PSEENI[0]) set High—and the PNP Audio Activate register (PUACTI[0]) is inactive, then bits 2–0 of this register are used to directly control the serial EEPROM. Bits 7–4 are read-only status bits that show the state of various control signals that are latched at the trailing edge of RESET. For details, see Appendix A, “Packaging and Pin Designations.” This register is accessible only when the PNP state machine is in the configuration state.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SUSPEND-C 32KHz Select	External Decode Select	PNP System Board Select	VCC is 5 V	Serial EEPROM Chip Select	Serial EEPROM Serial Clock	Serial EEPROM Data In	Serial EEPROM Data Out

Bit 7: **SUSPEND-C32KHZ Select**. Provides the state of the internal signal IPSUS32, which is latched off the RA[21] pin at the trailing edge of RESET.

Bit 6: **External Decode Select**. Provides the state of the internal signal IPEXDEC which is latched off the RA[20] pin at the trailing edge of RESET.

Bit 5: **PNP System Board Select**. Provides the state of the internal signal IPPNPSYS, which is latched off the PNPCS pin at the trailing edge of RESET.

Bit 4: **VCC is 5 V**. Provides the state of the internal 5-V/3.3-V detect circuitry. It is High for 5 V and Low for 3.3 V.

Bit 3: **Serial EEPROM Chip Select**. If the *Serial EEPROM Mode* bit of the PNP Serial EEPROM Enable register (PSEENI[0]) is set High, then the data latched in this bit is reflected on the PNPCS pin. Reads provide the latched value.

Bit 2: **Serial EEPROM Serial Clock**. Writes to this bit are reflected on the MD[2] pin. Reads provide the latched value.

Bit 1: **Serial EEPROM Data In**. Writes to this bit are reflected on the MD[1] pin. Reads provide the latched value.

Bit 0: **Serial EEPROM Data Out**. Writes to this bit are ignored. Reads provide the state of the MD[0] pin.

PPWRI—PNP Power Mode

Address: 0A79h write, PNPRDP read; index PIDXR=F2h and PLDNI=0

Default: X111 1111

Use this register to enable and disable clocks and low-power modes for major sections of the InterWave IC. Writing to this register is accomplished differently from most. Bit 7 specifies the value to be written (1 or 0); for bits 6–0, a High indicates that the value in bit 7 is to be written into the bit and a Low indicates that the bit is to be left unmodified. Thus, to modify a subset of bits 6–0, it is not necessary to read the register ahead of time to determine the state of bits that should not change. Examples: To set bit 0 High, write 81h; to clear bit 4, write 10h.

If a single command comes to set bits 6–1 Low (I/O write of 0111 111X, binary), then the IC enters shut-down mode and the 16.9-MHz oscillator becomes disabled. When, subsequently, one or more of bits 6–1 are set High, the 16.9-MHz oscillator is re-enabled. After being re-enabled, the 16.9-MHz clock requires 4 ms to 8 ms to become stable.

This register is accessible only when the IC is in the PNP configuration state.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Enable	24.576-MHz Oscillator Enable	Local Memory Control Enable	Synthesizer Enable	Game–MIDI Ports Enable	Codec Playback Path Enable	Codec Record Path Enable	Codec Analog Circuitry Enable

Bit 7: **Enable**. Specifies the value that is to be written to bits 6–0 of the register. In all seven cases, a High specifies that the block is functional and a Low indicates that it is in low-power mode.

Bit 6: **24.576-MHz Oscillator Enable**. If set Low, causes the 24.576-MHz oscillator to stop. It is not recommended that this oscillator be disabled if either the *Playback Crystal Select* bit of the Playback Data Format register (CPDFI[0]) or the *Record Crystal Select* bit of the Record Data Format register (CRDFI[0]) are Low. However, it is okay to set this bit Low as part of the shut-down command, regardless of the state of CPDFI[0] and CRDFI[0].

Bit 5: **Local Memory Control Enable**. If set Low, disables the 16.9-MHz clock to the local memory control module and allows slow refresh cycles to local DRAM using C32KHZ.

Bit 4: **Synthesizer Enable**. If set Low, disables the 16.9-MHz clock to the synthesizer module and the clocks to the synthesizer DAC input to the codec mixer.

Bit 3: **Game–MIDI Ports Enable**. If set Low, disables all clocks to the ports module and disables internal and external resistors from consuming current.

Bit 2: **Codec Playback Path Enable**. If set Low, disables clocks to the codec playback path including the playback FIFO, format conversion, filtering, and DAC.

Bit 1: **Codec Record Path Enable**. If set Low, disables clocks to the codec record path including the record FIFO, format conversion, filtering, and ADC.

Bit 0: **Codec Analog Circuitry Enable**. If set Low, disables all the codec analog circuitry and places it in a low-power mode. If set Low, the following analog pins are placed into the high-impedance state: MIC[L,R], AUX1[L,R], AUX2[L,R], LINEIN[L,R], MONOIN, LINEOUT[L,R], MONOOUT, CFILT, IREF.

PSRSTI—PNP Software Reset

Address: 0A79h write; index PIDXR=F3h and PLDNI=0

This register is accessible only when the IC is in the PNP configuration state.

Note: This register is similar to PCCCI; however, where writes to PCCCI affect all PNP-compliant cards in the system, writes to this register affect only the InterWave IC that is in the PNP configuration state.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Reserved	Reserved	PNP Register Reset	Reset CSN	Wait For Key	General Reset

Bits 7–4: Reserved.

Bit 3: **PNP Register Reset**. Setting this bit High resets the following registers:

- PNP Activate registers (PUACTI, PRACTI, PGACTI, PSACTI, PMACTI)
- PNP I/O Range Check registers (PURCI, PRRCI, PGRCI, PSRCI, PMRCI)
- PNP P2XR (P2X0HI, P2X0LI)
- PNP P3XR (P3X0HI, P3X0LI)

- PNP PCODAR (PHCAI, PLCAI)
- PNP Audio IRQ Channel Select (PUI1SI, PUI2SI)
- PNP DMA Channel Select (PUD1SI, PUD2SI)
- PNP Serial EEPROM Enable (PSEENI)
- PNP Serial EEPROM Control (PSECI)
- PNP Power Mode (PPWRI)
- PNP PCDRAR (PRAHI, PRALI)
- PNP PATAAR (PATAHI, PATALI)
- PNP CD-ROM IRQ Type (PRISI)
- PNP CD-ROM DMA Select (PRDSI)
- PNP P201AR (P201HI, P201LI)
- PNP P388AR (P388HI, P388LI)
- PNP AdLib–Sound Blaster Emulation IRQ Select (PSBISI)
- PNP P401AR (P401HI, P401LI)
- PNP MPU-401 Emulation IRQ Select (PMISI)
- Interrupt Control (UICI)
- DMA Channel Control (UDCI)

Bit 2: **Reset CSN**. Setting this bit High sets the Card Select Number (CSN) to zero.

Bit 1: **Wait For Key**. Setting this bit High causes the IC to enter the PNP wait-for-key state.

Bit 0: **General Reset**. Setting this bit High resets most of the IC. The only parts of the IC not reset by this command are the PNP Set Read Data Port (PSRPAI), the PNP Card Select Number register (PCSNI), the PNP state, and the registers reset by bit 3 of this register.

CHAPTER

13 CODEC/MIXER REGISTERS

The InterWave codec is fully register-compatible with the CS4231 (modes 1 and 2) and the AD1848 devices. The InterWave IC uses an indirect addressing mechanism for accessing most of the codec registers. In mode 1, there are 16 indirect registers; in mode 2, there are 28 indirect registers; and in mode 3, there are 32 indirect registers.

To better understand the purpose and action of the registers described in this chapter, refer to Figure 6-2, “Left Half of the InterWave Mixer,” on page 6-17.

Codec Direct Registers

CIDXR—Codec Index Address

Address: PCODAR+0 read, write

Default: 40h

Modes: bits 7–5, 3–0 in modes 1, 2, and 3; bit 4 in modes 2 and 3

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Initialization	Mode Change Enable	DMA Transfer Disable	Indirect Address Pointer				

Bit 7: **Initialization**. This read-only bit is High if the codec is in an initialization phase and unable to respond to I/O activity. This bit is set by software and hardware resets.

Bit 6: **Mode Change Enable**. This bit protects the Playback Data Format register (CPDFI), Record Data Format register (CRDFI), and Configuration Register 1 (CFIG1I) from being written (except CFIG1I[1:0]; these bits can be changed at any time). If set High, the protected registers can be modified, and the left and right DAC outputs are muted. If set Low, the protected registers cannot be modified. Moving this bit from High to Low resets the *Record FIFO Overrun* bit and *Playback FIFO Underrun* bit in the Codec Status Register 2 (CSR2I[7:0]) and the *Record FIFO Underrun*, *Record FIFO Overrun*, *Playback FIFO Overrun*, and *Playback FIFO Underrun* bits of the Codec Status Register 3 (CSR3I[3:0]).

Bit 5: **DMA Transfer Disable**. If set High, causes DMA transfers to be suspended when the sample counter interrupts of the Codec Status Register 3 (CSR3I) become active.

Mode 1 DMA is suspended (whether it be playback or record) when the sample counter stops after the sample counter causes an interrupt. Also, the active FIFO is disabled from transferring more data to the codec. DMA transfers, FIFO transfers, and the sample counter resume when the *Global Interrupt Status* bit of the Codec Status Register 1 (CSR1R[0]) is cleared or when this bit (CIDXR[5]) is cleared.

Modes 2 and 3

Record DMA, the record FIFO, and the record sample counter stop when the record sample counter causes an interrupt, read in the *Record FIFO Interrupt Request* bit of Codec Status Register 3 (CSR3I[5]=1); playback DMA, the playback FIFO, and the playback sample counter all stop when the playback sample counter causes an interrupt. That interrupt is read in the *Playback FIFO Interrupt Request* bit of Codec Status Register 3 ((CSR3I[4]=1). The pertinent DMA transfers and sample counter resume when the appropriate interrupt bit in CSR3I is cleared or when this bit (CIDXR[5]) is cleared.

In mode 3, this bit also works to discontinue the transfer of data between the codec FIFOs and the Local Memory Record and Playback FIFOs.

Bits 4–0: **Indirect Address Pointer**. These bits point to registers in the indirect address space. In mode 1, a 16-register space is defined and bit 4 of this address is reserved. In modes 2 and 3, a 32-register space is defined.

CDATAP—Codec Indexed Data Port

Address: PCODAR+1 read, write

Modes: 1, 2, and 3

Read and write to all of the codec indexed registers—pointed to by the *Indirect Address Pointer* field of the Codec Index Address register (CIDXR[4:0])—through this port.

CSR1R—Codec Status Register 1

Address: PCODAR+2 read, write

Default: CCh

Modes: 1, 2, and 3

This register reports the interrupt status and various playback and record FIFO conditions. Reading this register also clears the overrun and underrun bits of the Codec Status Register 2 (CSR2I[7:6]) and Codec Status Register 3 (CSR3I[3:0]), if any are set. Writing to this register clears all codec interrupts and the *Global Interrupt Status* bit (CSR1R[0]).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Record Channel Upper/Lower Byte Indication	Record Channel Left/Right Sample Indication	Record Channel Data Available	Sample Error	Playback Channel Upper/Lower Byte Indication	Playback Channel Left/Right Sample Indication	Playback Channel Buffer Available	Global Interrupt Status

Bit 7: **Record Channel Upper/Lower Byte Indication**. When High, indicates that a read of the record FIFO returns the upper byte of a 16-bit sample (bits 15–8) or that the record data is eight or fewer bits wide. When Low, indicates that a read of the record FIFO returns the lower byte of a 16-bit sample (bits 7–0). After the last byte of the last received sample has been read from the record FIFO, this bit does not change from its state during that byte until the next sample is received.

Bit 6: **Record Channel Left/Right Sample Indication**. When High, indicates that a read of the record FIFO returns the left sample or that the record path is in either mono or ADPCM mode (or both). When Low, a read returns the right sample. After the last byte of the last received sample has been read from the record FIFO, this bit does not change from its state during that byte until the next sample is received.

Bit 5: **Record Channel Data Available**. When High, there is valid data to be read from the record FIFO. When Low, the FIFO is empty.

Bit 4: **Sample Error**. This bit is High whenever data has been lost because of either a record FIFO overrun or a playback FIFO underrun—it is a logical OR of the *Record FIFO Overrun* and *Playback FIFO Underrun* bits of the Codec Status Register 2 (CSR2I[7:6]) and the Codec Status Register 3 (CSR3I[3:0]). If both record and playback channels are enabled, the specific channel that set this bit can be determined by reading the other codec status registers (CSR2I or CSR3I). However, the overrun and underrun status bits in CSR2I and CSR3I are cleared when this register is read, so any overrun or underrun detection process should involve reading CSR2I or CSR3I before CSR1R is read.

Bit 3: **Playback Channel Upper/Lower Byte Indication**. When High, indicates that the next write to the playback FIFO should be the upper byte of a 16-bit sample (bits 15–8) or that playback data is eight or fewer bits wide. When Low, indicates the next write to the playback FIFO should be the lower byte (bits 7–0) of a 16-bit sample. After the playback FIFO becomes full, this bit stays in the state of the last byte written until a space becomes available in the FIFO.

Bit 2: **Playback Channel Left/Right Sample Indication**. When High, indicates that the next write to the playback FIFO should be the left sample or that the playback path is in either mono or ADPCM mode. When Low, indicates the right sample is expected. After the playback FIFO becomes full, this bit stays in the state of the last byte written until a space becomes available in the FIFO.

Bit 1: **Playback Channel Buffer Available**. When High, there is room in the playback FIFO for additional data. When Low, the FIFO is full.

Bit 0: **Global Interrupt Status**. This bit is High whenever there is an active condition that can request an interrupt. It is implemented by ORing together the three sources of interrupts in the codec, found in the Codec Status Register 3 (CSR3I[6:4]).

To clear this bit, write any value to the Codec Status Register 1 (CSR1R) or write a 0 to the status bit in the Codec Status Register 3 (CSR3I) causing the interrupt. If the *Playback Enable* bit (CFIG1I[0]) or the *Record Enable* bit (CFIG1I[1]) in the Configuration Register 1 change from High to Low while this bit is High, the active interrupt still has to be cleared.

CPDR, CRDR—Playback and Record Data

Address: PCODAR+3 read (record FIFO), write (playback FIFO)

Modes: 1, 2, and 3

Data written to this address is loaded into the playback FIFO. Data read from this address is removed from the record FIFO. Bits in the Codec Status Register 1 (CSR1R) indicate whether the data is for the left or right channel, and for 16-bit samples, the upper or lower portion of the sample. Writes to this address are ignored when either the playback FIFO is in DMA mode or the playback path is not enabled—the *Playback Enable* bit of the Configuration Register 1 (CFIG1I[0]) is Low. Reads from this address are ignored when either the record FIFO is in DMA mode or the record path is not enabled—the *Record Enable* bit of the Configuration Register 1 (CFIG1I[1]) is Low.

Codec CIDXR, CDATAP Indexed Registers

CLICI, CRICI—Left/Right ADC Input Control

Address: PCODAR+1 read, write; left index CIDXR[4:0]=0, right index CIDXR[4:0]=1

Default: 000X 0000 (for both)

Modes: 1, 2, and 3

These registers are used to select the input source to the analog-to-digital converter (ADC) and to specify the amount of gain to be applied to the signal path. The registers are identical: One controls the left channel and the other controls the right channel.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Left/Right ADC Source Select		Read/Write Bit	Reserved	Left/Right ADC Input Gain Select			

Bits 7–6: **Left/Right ADC Source Select**. These bits select which input source will be fed to the analog-to-digital converter.
00:Line
01:Auxiliary 1
10:Stereo Microphone
11:Mixer Output

Bit 4: Reserved.

Bit 5: **Read/Write Bit**. This bit does not control anything. Reads return what has been written to it.

Bits 3–0: **Left/Right ADC Input Gain Select**. The selected input source is fed to the ADC through a gain stage. These four bits specify the amount of gain applied to the signal. The values vary from 0h = 0 dB to 0Fh = +22.5 dB with 1.5 dB per step.

CLAX1I, CRAX1I—Left/Right Auxiliary 1/Synthesizer Input Control

Address: PCODAR+1 read, write; left index CIDXR[4:0]=2, right index CIDXR[4:0]=3

Default: 1XX0 1000 (for both)

Modes: 1, 2, and 3

This register pair controls the left and right Auxiliary 1 or Synthesizer inputs—multiplexed by the *Aux 1/Synth Signal Select* field of the Configuration Register 3 (CFIG3I[1])—to the mixer. The registers are identical: One controls the left channel and the other controls the right channel.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Left/Right Aux 1/Synth Mute Enable	Reserved	Reserved	Left/Right Aux 1/Synth Gain Select				

Bit 7: **Left/Right Aux 1/Synth Mute Enable**. If set High, the selected input is muted. If set Low, the input operates normally.

Bits 6–5: Reserved.

Bits 4–0: **Left/Right Aux 1/Synth Gain Select**. These bits specify the amount of gain applied to the selected Auxiliary 1 or Synthesizer input signal. The values vary from 00h = +12 dB to 1Fh = –34.5 dB with 1.5 dB per step.

CLAX2I, CRAX2I—Left/Right Auxiliary 2 Input Control

Address: PCODAR+1 read, write; left index CIDXR[4:0]=4, right index CIDXR[4:0]=5

Default: 1XX0 1000 (for both)

Modes: 1, 2, and 3

This register pair controls the left and right Auxiliary 2 inputs to the mixer. The registers are identical: One controls the left channel and the other controls the right channel.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Left/Right Aux 2 Mute Enable	Reserved	Reserved	Left/Right Aux 2 Gain Select				

Bit 7: **Left/Right Aux 2 Mute Enable**. If set High, the Auxiliary 2 input is muted. If set Low, the input operates normally.

Bits 6–5: Reserved.

Bits 4–0: **Left/Right Aux 2 Gain Select**. These bits specify the amount of gain applied to the Auxiliary 2 input signal. The values vary from 00h = +12 dB to 1Fh = –34.5 dB with 1.5 dB per step.

CLDACI, CRDACI—Left/Right Playback DAC Control

Address: PCODAR+1 read, write; left index CIDXR[4:0]=6, right index CIDXR[4:0]=7

Default: 1X00 0000 (for both)

Modes: 1, 2, and 3

Each of these registers sums the DAC output and the loopback signal from the input to the ADC as they are input to the mixer. The registers are identical: One controls the left channel and the other controls the right channel.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Left/Right DAC Mute Enable	Reserved	Left/Right DAC Attenuation Select					

Bit 7: **Left/Right DAC Mute Enable**. If set High, the DAC output and the loopback signal from the input to ADC are muted. If set Low, the input to the mixer operates normally.

Bit 6: Reserved.

Bits 5–0: **Left/Right DAC Attenuation Select**. These bits specify the amount of attenuation applied to the DAC output and the loopback signal from the input to the ADC. The values vary from 00h = 0 dB to 3Fh = –94.5 dB with 1.5 dB per step.

CPDFI—Playback Data Format

Address: PCODAR+1 read, write; index CIDXR[4:0]=8

Default: 00h

Modes: The definition of this register varies depending on the mode.

This register specifies the sample rate (selects the oscillator to be used and the divide factor for that oscillator), stereo or mono operation, linear or compressed data, and 8-bit or 16-bit data. It can be changed only when the *Mode Change Enable* bit of the Codec Index Address register (CIDXR[6]) is High (active).

Mode 1 This register controls both the playback and record paths.

Mode 2 Bits 3–0 of this register control both the record and playback sample rate (i.e., both rates must be the same), and bits 7–4 specify the state of the playback-path data format.

Mode 3 This register controls only the playback path; the record sample rate is controlled by the Record Data Format register (CRDFI).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Playback Data Format Select			Playback Stereo/Mono Select	Playback Clock Divider Select			Playback Crystal Select

Bits 7–5: **Playback Data Format Select**. These 3 bits specify the playback data format for the codec.
000:8-bit unsigned
001:µ-law
010:16-bit signed, little endian
011:A-law
100:Reserved, default to 8-bit unsigned*
101:IMA-compliant ADPCM*
110:16-bit signed, big endian*
111:Reserved, default to 8-bit unsigned*
*Modes 2 and 3 only. In mode 1, bit 7 is treated as Low regardless of the value written to it.

Bit 4: **Playback Stereo/Mono Select**. If set High, stereo operation is selected; samples alternate left then right. If set Low, mono mode is selected; playback samples are fed to both left and right FIFOs. Record samples (in mode 1) come only from the left ADC.

Bits 3–1: **Playback Clock Divider Select**. These three bits specify the playback clock rate in mode 3 and the record and playback rate in modes 1 and 2. The possible values are listed in Table 13-1.

Table 13-1 Playback Clock Divider Selections

	Sampling Rate in kHz	
Bits 3 2 1	**24.576-MHz crystal (XTAL1)**	**16.9344-MHz crystal (XTAL2)**
000	8.0	5.51
001	16.0	11.025
010	27.42	18.9
011	32.0	22.05
100	÷ 448*	37.8
101	÷ 384*	44.1
110	48.0	33.075
111	9.6	6.62

Note:

**These divide-downs are provided but are not guaranteed to function unless XTAL1 is less than 18.5 MHz.*

Bit 0: **Playback Crystal Select**. If set High, the 16.9344-MHz crystal oscillator (XTAL2) is used for the playback sample frequency. If set Low, the 24.576-MHz crystal oscillator (XTAL1) is used.

CFIG1I—Configuration Register 1

Address: PCODAR+1 read, write; index CIDXR[4:0]=9

Default: 00XX 1000

Modes: 1, 2, and 3

This register specifies whether to use I/O cycles or DMA to service the codec FIFOs, selects one-channel or two-channel DMA operation, and enables or disables the record and playback paths. Bits 7–2 are protected; to write to protected bits, the *Mode Change Enable* bit of the Codec Index Address register (CIDXR[6]) must be set High (active).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Record FIFO I/O Select	Playback FIFO I/O Select	Reserved	Reserved	Calibration Emulation	1 or 2 Channel DMA Operation Select	Record Enable	Playback Enable

Bit 7: **Record FIFO I/O Select**. If set High, the record FIFO can be serviced only through I/O cycles. If set Low, DMA operation is supported.

Bit 6: **Playback FIFO I/O Select**. If set High, the playback FIFO can be serviced only through I/O cycles. If set Low, DMA operation is supported.

Bits 5–4: Reserved.

Bit 3: **Calibration Emulation**. This bit is present for compatibility with the CS4231and has no actual effect on the operation of the InterWave IC. For more details, see the *Calibration Active Emulation* bit (CSR2I[5]) description in “CSR2I—Codec Status Register 2” on page 13-9.

Bit 2: **1 or 2 Channel DMA Operation Select**. If set High, single-channel DMA operation is selected. Record or playback operation is allowed, but not both. When both record and playback DMA are enabled while this bit is set, only the playback transfers are serviced. If set Low, two-channel DMA operation is allowed.

Bit 1: **Record Enable**. If set High, the record codec path is enabled. If set Low, the record path is turned off and the *Record Channel Data Available* bit of the Codec Status Register 1 (CSR1R[5]) is held Low (inactive).

Bit 0: **Playback Enable**. If set High, the playback codec path is enabled. If set Low, the playback path is turned off and the *Playback Channel Buffer Available* bit of the Codec Status Register 1 (CSR1R[1]) is held Low (inactive).

CEXTI—External Control

Address: PCODAR+1 read, write; index CIDXR[4:0]=Ah

Default: 00XX 0X0X

Modes: 1, 2, and 3

This register contains the global interrupt enable control as well as control bits for the two general purpose external output pins.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
General Purpose Output Flags		Reserved	Reserved	Read/Write Bit	Reserved	Global Interrupt Enable	Reserved

Bits 7–6: **General Purpose Output Flags**. The state of these bits are reflected on the GPOUT1 and GPOUT0 pins. These two pins default to IRQ10 and IRQ4. They become the general purpose output flags when the *Select GPOUT Codec Flags* bit of the Emulation IRQ register (IEIRQI[7]) is High.

Bits 5–4: Reserved.

Bit 3: **Read/Write Bit**. This bit can be read and written but it does not control anything within the InterWave IC.

Bit 2: Reserved.

Bit 1: **Global Interrupt Enable.** If set High, enables codec interrupts. If set Low, codec interrupts are not passed on to the selected IRQ pin. The status bits are not affected by the state of this bit.

Bit 0: Reserved.

CSR2I—Codec Status Register 2

Address: PCODAR+1 read; index CIDXR[4:0]=Bh

Default: 00h

Modes: 1, 2, and 3

This register reports certain FIFO errors and the state of the record and playback data request bits, and allows testing of the analog-to-digital paths for clipping.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Record FIFO Overrun	Playback FIFO Underrun	Calibration Active Emulation	DMA Request Pin Status	Right Overrange Detect		Left Overrange Detect	

Bit 7: **Record FIFO Overrun**. When High, the record FIFO is full and the codec needs to load another sample (the sample is discarded). This bit is cleared to Low when the Codec Status Register 1 (CSR1R) is read or when the *Mode Change Enable* bit of the Codec Index Address register (CIDXR[6]) goes from High to Low.

Bit 6: **Playback FIFO Underrun**. When High, the playback FIFO is empty and the codec needs another sample. This bit is cleared to Low when the Codec Status Register 1 (CSR1R) is read or when the *Mode Change Enable* bit of the Codec Index Address register (CIDXR[6]) goes from High to Low. (In mode 1, the previous sample is reused. In modes 2 and 3, either the previous sample is reused or the data is forced to all zeros depending on the state of the *DAC Output Force Enable* bit of the Configuration Register 2 (CFIG2I[0]). For details, see “CFIG2I—Configuration Register 2” on page 13-11.

Bit 5: **Calibration Active Emulation**. If the *Calibration Emulation* bit of Configuration Register 1 (CFIG1I[3]) is High, this bit goes High when the *Mode Change Enable* bit of the Codec Index Address register (CIDXR[6]) goes from High to Low; it goes back to Low after the trailing edge of the first subsequent read of the Codec Status Register 2 (CSR2I). This action has no actual effect on InterWave IC operation and is present only for compatibility with the Crystal 4231.

Bit 4: **DMA Request Pin Status**. This bit is High anytime that either the record or playback DMA request pins are active.

Bits 3–2: **Right Overrange Detect**. See *Left Overrange Detect*.

Bits 1–0: **Left Overrange Detect**. These two pairs of bits are updated on a sample by sample basis to reflect whether the signal into the ADC is causing clipping.
00:Less than 1.5 dB underrange
01:Between 1.5 dB and 0 dB underrange
10:Between 0 dB and 1.5 dB overrange
11:More than 1.5 dB overrange

CMODEI—Mode Select, ID

Address: PCODAR+1 read, write; index CIDXR[4:0]=Ch

Default: 100X 1010

Modes: 1, 2, and 3

This register specifies the operating mode of the codec and reports the revision number of the InterWave IC.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Revision ID Number Bit 4	Mode Select		Reserved	Revision ID Number Bits 3–0			

Bits 7, 3–0: **Revision ID Number**. These five bits specify the revision number of the IC, which is 1,1010 for the first AMD part. These bits are read-only; they cannot be changed.

Bits 6–5: **Mode Select**. To enter mode 3, write 6Ch to this port; bit 5 is forced Low for writes of any other value.
00:mode 1
10:mode 2
01:reserved
11:mode 3

Bit 4: Reserved.

CLCI—Loopback Control

Address: PCODAR+1 read, write; index CIDXR[4:0]=Dh

Default: 0000 00X0

Modes: 1, 2, and 3

This register enables and specifies the attenuation of the analog path between the output of the ADC path gain stage (at the input to the ADC) and the input of the DAC-loopback sum. This register affects both the left and right channels.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Loopback Attenuation						Reserved	Loopback Enable

Bits 7–2: **Loopback Attenuation**. These bits specify the amount of attenuation applied to the loopback signals before being summed with the DAC outputs. The values vary from 00h = 0 dB to 3Fh = –94.5 dB with 1.5 dB per step.

Bit 1: Reserved.

Bit 0: **Loopback Enable**. If set High, the loopback path is enabled for mixing with the DAC outputs. When cleared, the path is disabled and the signal is muted.

CUPCTI, CLPCTI—Upper/Lower Playback Count

Address: PCODAR+1 read, write; upper index CIDXR[4:0]=Eh, lower index CIDXR[4:0]=Fh

Default: 00h (for both)

Modes: The definition of these registers varies depending on the mode.

These registers collectively provide the 16-bit preload value used by the playback sample counters. CUPCTI provides the upper preload bits 15–8 and CLPCTI provides the lower preload bits 7–0. All 16 bits are loaded into the counter during the write of the upper byte; therefore, the lower byte should be written first. However, if only the low byte is written and the counter underflows, the new value is placed into the counter. The preload value is loaded into the counter on the cycle after the counter decrements to 0. Reads of these registers return the value written into them, not the current state of the counter. In mode 1, these registers are used for both playback and record; in modes 2 and 3 they are used for playback only.

CFIG2I—Configuration Register 2

Address: PCODAR+1 read, write; index CIDXR[4:0]=10h

Default: 0000 XXX0

Modes: 2 and 3

This register selects the full-scale voltage output range, enables the codec timer, enables the record and playback sample counters, and enables DAC output forcing.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Output Full-Scale Voltage Select	Timer Enable	Record Sample Counter Disable	Playback Sample Counter Disable	Reserved	Reserved	Reserved	DAC Output Force Enable

Bit 7: **Output Full-Scale Voltage Select**. If set High, the full-scale output is 2.9 V for V_{cc} = 5 V and 1.34 for V_{cc} = 3.3 V. If set Low, the full-scale output is 2.0 V for V_{cc} = 5 V and 1.00 for V_{cc} = 3.3 V. This bit affects the left and right signals that exit the mixers, prior to entering the Left and Right Output Attenuation registers (CLOAI and CROAI); therefore, it also affects the input to the record multiplexer.

Bit 6: **Timer Enable**. If set High, the timer and its associated interrupt are enabled. If set Low, the timer is disabled. The timer count is specified in the Lower Timer and Upper Timer registers (CLTIMI and CUTIMI).

Bit 5: **Record Sample Counter Disable**. If set High, this bit disables the record sample counter from counting. This bit can be accessed in mode 3 only and affects only the sample counter in mode 3.

Bit 4: **Playback Sample Counter Disable**. If set High, this bit disables the playback sample counter from counting. This bit can be accessed in mode 3 only and affects only the sample counter in mode 3.

Bits 3–1: Reserved.

Bit 0: **DAC Output Force Enable**. If set High, the output of the DACs are forced to the center of the scale whenever a playback FIFO underrun error occurs. When cleared, the last valid sample is output in the event of an underrun.

CFIG3I—Configuration Register 3

Address: PCODAR+1 read, write; index CIDXR[4:0]=11h

Default: 0000 X000

Modes: bits 7–1 in mode 3; bit 0 in modes 2 and 3

In mode 3, this register provides for the programming of FIFO thresholds and the generation of I/O-mode FIFO service interrupts.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Record FIFO Service Request Interrupt Enable	Playback FIFO Service Request Interrupt Enable	FIFO Threshold Select		ADPCM Record Suspend	Variable Frequency Playback Mode	Aux 1/Synth Signal Select	Read/Write Bit

Bit 7: **Record FIFO Service Request Interrupt Enable**. When the record path is enabled—the *Record Enable* bit of the Configuration Register 1 (CFIG1I[1]) is set High—setting this bit High enables the generation of an interrupt request when the *Record FIFO Interrupt Request* bit in the Codec Status Register 3 (CSR3I) goes High. This bit can be accessed in mode 3 only.

Bit 6: **Playback FIFO Service Request Interrupt Enable**. When the playback path is enabled—the *Playback Enable* bit of the Configuration Register 1 (CFIG1I[0]) is set High—setting this bit High enables the generation of an interrupt request when the *Playback FIFO Interrupt Request* bit in Codec Status Register 3 (CSR3I) goes High. This bit can be accessed in mode 3 only.

Bits 5–4: **FIFO Threshold Select**. These two bits specify the record and playback FIFO thresholds, as listed in Table 13-2.

When the appropriate bit of the Configuration Register 1 is Low—*Record FIFO I/O Select* (CFIG1I[7]) or *Playback FIFO I/O Select* (CFIG1I[6])—a DMA request occurs when the FIFO threshold is reached. The DMA transfers occur until the playback FIFO is full or the record FIFO is empty.

The appropriate interrupt bit in the Codec Status Register 3 (CSR3I[5] for record, CSR3I[4] for playback) is set High when the FIFO threshold is reached and the following conditions exist:

- The FIFO I/O select bit is High
- The InterWave IC is programmed for mode 3 operation
- The appropriate FIFO service request interrupt enable bit is High (CFIG3I[7] for record, CFIG3I[6] for playback).

Table 13-2 FIFO Threshold Selections

FT 1 0	Point at Which Request Becomes Active
0 0	Minimum: Record FIFO not empty; playback FIFO not full
0 1	Middle: Record FIFO half full; playback FIFO half empty
1 0	Maximum: Record FIFO full; playback FIFO empty
1 1	Reserved

These bits are active in mode 3 only and have no effect in modes 1 and 2.

For a complete understanding of the FIFO threshold interrupt structure, see "Codec Interrupt Structure" on page 6-6.

Bit 3: **ADPCM Record Suspend**. When the record path is in ADPCM mode, setting this bit High causes recording to suspend. Setting this bit Low allows recording to resume without resetting the ADPCM index that is used when calculating ADPCM values. The index should be reset only at the beginning of a file of record data. The index is reset when the *Record Enable* bit of the Configuration Register 1 (CFIG1I[1]) is inactive (Low).

Bit 2: **Variable Frequency Playback Mode**. If set High, selects variable-frequency-playback mode. In this mode, the sample rate is selected by a combination of the *Playback Crystal Select* bit of the Playback Data Format register (CPDFI[0]) and the Playback Variable Frequency register (CPVFI) to allow variable frequencies between 3.5 kHz and 32 kHz. This bit can be accessed in mode 3 only.

Bit 1: **Aux 1/Synth Signal Select**. This bit selects the source of the signals that enter the Left/Right Auxiliary 1/Synthesizer Input Control register (CLAX1I and CRAX1I) attenuators before entering the left and right mixers. This bit Low selects the AUX1[L,R] input pins. If set High, selects the output of the synthesizer DACs. This bit can be accessed in mode 3 only.

Bit 0: **Read/Write Bit**. This bit can be read and written but it does not control anything within the InterWave IC. This bit can be accessed in mode 2 and mode 3.

CLLICI, CRLICI—Left/Right Line Input Control

Address: PCODAR+1 read, write; left index CIDXR[4:0]=12h, right index CIDXR[4:0]=13h

Default: 1XX0 1000 (for both)

Modes: 2 and 3

This register pair controls the gain or attenuation applied to the line inputs to the mixer. The registers are identical; one controls the left channel and the other controls the right channel.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Left/Right Line Input Mute Enable	Reserved	Reserved	Left/Right Line Input Gain Select				

Bit 7: **Left/Right Line Input Mute Enable**. If set High, the LINE IN input is muted. If set Low, the input operates normally.

Bits 6–5: Reserved.

Bits 4–0: **Left/Right Line Input Gain Select**. These bits specify the amount of gain applied to the LINE IN[L,R] input signals. The values vary from 0 = +12 dB to 1Fh = –34.5 dB with 1.5 dB per step.

CUTIMI, CLTIMI—Upper Timer, Lower Timer

Address: PCODAR+1 read, write; low index CIDXR[4:0]=14h, upper index CIDXR[4:0]=15h

Default: 00h (for both)

Modes: 2 and 3

These registers collectively provide the 16-bit preload value used by the general purpose timer. Each count represents 10 µs (total of 650 ms). CUTIMI provides the upper preload bits 15–8 and CLTIMI provides the lower preload bits 7–0. Writing to CLTIMI causes all 16 bits to be loaded into the general purpose timer, so write to CUTIMI first. Reads of these registers return the value written into them, not the current state of the counter.

When the timer decrements to 0, the *Timer Interrupt Request* bit in the Codec Status Register 3 (CSR3I[6]) is set. The values in CUTIMI and CLTIMI are loaded into the counter on the next clock cycle, independent of the state of the interrupt bit.

CLMICI, CRMICI—Left/Right Microphone Input Control

Address: PCODAR+1 read, write; left index CIDXR[4:0]=16h, right index CIDXR[4:0]=17h

Default: 1XX0 1000 (for both)

Modes: 3

This register pair controls the left and right MIC inputs to the mixer. The registers are identical; one controls the left channel and the other controls the right channel.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Left/Right MIC Mute Enable	Reserved	Reserved	Left/Right MIC Gain Select				

Bit 7: **Left/Right MIC Mute Enable**. If set High, the MIC input is muted. If set Low, the input operates normally.

Bits 6–5: Reserved.

Bits 4–0: **Left/Right MIC Gain Select**. These bits specify the amount of gain applied to the MIC[L,R] input signals. The values vary from 0 = +12 dB to 1Fh = –34.5 dB with 1.5 dB per step.

CSR3I—Codec Status Register 3

Address: PCODAR+1 read, write (to clear specific bits); index CIDXR[4:0]=18h

Default: X000 0000

Modes: 2 and 3; definition of bits 5–4 vary based on the mode

This register provides additional status information on the FIFOs and reports the cause of various interrupt requests. Each of the bits 6–4 are cleared by writing a 0 to the active bit; writing a 1 to a bit is ignored; these bits can also be cleared by a write of any value to the Codec Status Register 1 (CSR1R). Bits 3–0, the underrun-overrun bits, are cleared to a Low by reading CSR1R; these bits are also cleared when the *Mode Change Enable* bit in the Codec Index Address register (CIDXR[6]) goes from High to Low.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Timer Interrupt Request	Record FIFO Interrupt Request	Playback FIFO Interrupt Request	Record FIFO Underrun	Record FIFO Overrun	Playback FIFO Overrun	Playback FIFO Underrun

Bit 7: Reserved.

Bit 6: **Timer Interrupt Request**. When High, indicates an interrupt request from the timer. It is cleared by writing a 0 to this bit or by writing any value to the Codec Status Register 1 (CSR1R).

Bit 5: **Record FIFO Interrupt Request**. When High, indicates a record path interrupt. It is cleared by writing a 0to this bit or by writing any value to CSR1R.

Mode 2 This bit indicates an interrupt request from the record sample counter.

Mode 3 and CFIG1I[7] = 0 (DMA)
This bit indicates an interrupt request from the record sample counter. The setting in the *FIFO Threshold Select* field of Configuration Register 3 (CFIG3I[5:4]) determines at which point DMA requests become active.

Mode 3 and CFIG1I[7] = 1 (I/O)
This bit indicates that the record FIFO threshold (specified in the *FIFO Threshold Select* field of Configuration Register 3 (CFIG3I[5:4])) has been reached.

Bit 4: **Playback FIFO Interrupt Request**. If set High, indicates a playback path interrupt. It is cleared by writing a 0 to this bit or by writing any value to the Codec Status Register 1 (CSR1R).

Mode 2 This bit indicates an interrupt request from the playback sample counter.

Mode 3 and CFIG1I[6] = 0 (DMA)
This bit indicates an interrupt request from the playback sample counter.

Mode 3 and CFIG1I[6] = 1 (I/O)

This bit indicates that the playback FIFO threshold (specified in the *FIFO Threshold Select* field of Configuration Register 3 (CFIG3I[5:4])) has been reached.

Bit 3: **Record FIFO Underrun** (Modes 2, 3). This bit is set High if there is an attempt to read from an empty record FIFO.

Bit 2: **Record FIFO Overrun** (Modes 2, 3). This bit is set High if the ADC needs to load a sample into a full record FIFO. The record sample is lost and the FIFO contents are not changed. CSR3I[2] is identical to the *Record FIFO Overrun* bit of the Codec Status Register 2 (CSR2I[7]).

Bit 1: **Playback FIFO Overrun** (Modes 2, 3). This bit is set High if there is an attempt to write to a full playback FIFO.

Bit 0: **Playback FIFO Underrun** (Modes 2, 3). This bit is set High if the DAC needs a sample from an empty playback FIFO. The previous sample is used, of if the *DAC Output Force Enable* bit of the Configuration Register 2 (CFIG2[0]) is High, a zero value sample is used. CSR3I[0] is identical to the *Playback FIFO Underrun* bit of the Codec Status Register 2 (CSR2I[6]).

CLOAI, CROAI—Left/Right Output Attenuation

Address: PCODAR+1 read, write; left index CIDXR[4:0]=19h, right index CIDXR[4:0]=1Bh

Default: 1XX0 0000 (for both);

Modes: 3 only. In mode 2, CLOAI is a read-only register that drives an 80h when read.

This register pair controls the left and right MONO and LINE output levels. The LINE output mute control bit is also located in this register pair.

Note: These registers can be written to only in mode 3, but any value written in mode 3 is still in effect if software changes operation to mode 1 or mode 2.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Line Output Mute Enable	Reserved	Reserved	Line Output Attenuation Select				

Bit 7: **Line Output Mute Enable**. If set High, the LINE output is muted. If set Low, the output operates normally.

Bits 6–5: Reserved.

Bits 4–0: **Line Output Attenuation Select**. These bits specify the amount of attenuation applied to both the MONO and LINE output signals. The values vary from 00h = 0 dB to 1Fh = –46.5 dB with 1.5 dB per step.

CMONOI—Mono Input and Output Control

Address: PCODAR+1 read, write; index CIDXR[4:0]=1Ah

Default: 000X 0000

Modes: bits 7–6, 4–0 modes 2 and 3; bit 5 mode 3

This register specifies the amount of attenuation applied to the MONO input path. The mute controls for the MONO input and output are also located here.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Mono Input Mute Enable	Mono Output Mute Enable	AREF to High Impedance	Reserved	Mono Input Attenuation			

Bit 7: **Mono Input Mute Enable**. If set High, the MONO input is muted. If set Low, the input is active.

Bit 6: **Mono Output Mute Enable**. If set High, the MONO output is muted. If set Low, the output operates normally.

Bit 5: **AREF to High Impedance**. If set High, the AREF pin is placed into high-impedance mode. If Low, AREF operates normally. This bit is accessible only in mode 3.

Bit 4: Reserved.

Bits 3–0: **Mono Input Attenuation**. This specifies the amount of attenuation to be applied to the MONO input path. The values vary from 0 = 0 dB to 0Fh = –45 dB with 3.0 dB per step.

CRDFI—Record Data Format

Address: PCODAR+1 read, write; index CIDXR[4:0]=1Ch

Default: 00h

Modes: 2 and 3; definition of register varies based on the mode

This register specifies the sample rate (selects which of the two oscillators is to be used and the divide factor for that oscillator), stereo or mono operation, linear or companded data, and 8-bit or 16-bit data. It can only be changed when the *Mode Change Enable* bit of the Codec Index Address register (CIDXR[6]) is High (active).

Mode 2 Bits 3–0 are not used (the record-path sample rate is specified in the Playback Data Format register (CPDFI)) and bits 7–4 specify the record-path data format.

Mode 3 All of this register controls record path attributes; the playback attributes are controlled by the Playback Data Format register (CPDFI).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Record Data Format Selection			Record Stereo/Mono Select	Record Clock Divider Select			Record Crystal Select

Bits 7–5: **Record Data Format Selection**. These 3 bits specify the playback data format for the codec. These bits can be accessed in modes 2 and 3 only.
000:8-bit unsigned
001:µ-law
010:16-bit signed, little endian
011:A-law
100:Reserved, default to 8-bit unsigned
101:IMA-compliant ADPCM
110:16-bit signed, big endian
111:Reserved, default to 8-bit unsigned

Bit 4: **Record Stereo/Mono Select**. If set High, stereo operation is selected; samples alternate left then right. If set Low, mono mode is selected; record samples come only from the left ADC. This bit can be accessed in modes 2 and 3 only.

Bits 3–1: **Record Clock Divider Select**. These three bits specify the record clock rate. These bits can be accessed in mode 3 only; in mode 2, they are reserved.

Table 13-3 Record Clock Divider Selections

Bits 3,2,1	Sampling Rate in kHz	
	24.576-MHz crystal (XTAL1)	16.9344-MHz crystal (XTAL2)
000	8.0	5.51
001	16.0	11.025
010	27.42	18.9
011	32.0	22.05
100	÷ 448*	37.8
101	÷ 384*	44.1
110	48.0	33.075
111	9.6	6.62

Note:

**These divide-downs are provided but are not guaranteed to function unless XTAL1 is less than 18.5 MHz.*

Bit 0: **Record Crystal Select**. If set High, the 16.9344-MHz crystal oscillator is used. If set Low, the 24.576-MHz crystal oscillator is used. This bit can be accessed from mode 3 only; in mode 2, this bit is reserved.

CPVFI—Playback Variable Frequency

Address: PCODAR+1 read, write; index CIDXR[4:0]=1Dh

Default: 00h

Modes: 3 only

This 8-bit register specifies the playback frequency when the *Variable Frequency Playback* bit of the Configuration Register 3 (CFIG3I[2]) is set High. The playback frequency is PCS/(16•(48+CPVFI)), where PCS is the frequency of the oscillator selected in the *Playback Crystal Select* bit of the Playback Data Format register (CPDFI[0]). The 16.9-MHz oscillator provides a range from about 3.5 kHz to 22.05 kHz; the 24.5-MHz oscillator provides a range

from about 5.0 kHz to 32 kHz. It is not necessary to set the *Mode Change Enable* bit of the Codec Index Address register (CIDXR[6]) High when altering the value of this register.

Note: It is not recommended that the* Playback Crystal Select *field of the Playback Data Format register (CPDFI[0]) be changed while in variable frequency playback mode. Doing so causes glitches in internal circuitry and can produce unpredictable results.

CURCTI, CLRCTI—Upper/Lower Record Count

Address: PCODAR+1 read, write; upper index CIDXR[4:0]=1Eh, lower index CIDXR[4:0]=1Fh

Default: 00h (for both)

Modes: 2 and 3. In mode 1, function is moved to CUPCTI and CLPCTI.

These registers collectively provide the 16-bit preload value used by the record sample counters. CURCTI provides the upper preload bits 15–8 and CLRCTI provides the lower preload bits 7–0. All 16 bits are loaded into the counter during the write of the upper byte; therefore, the lower byte should be written first. However, if only the low byte is written and the counter underflows, the new value is paced into the counter. The preload value is loaded into the counter on the cycle after the counter decrements to 0. Reads of these registers return the value written into them, not the current state of the counter.

CHAPTER

14 SYNTHESIZER REGISTERS

Direct Register

SVSR—Synthesizer Voice Select

Address: P3XR+2h read/write

Default: 00h

Use this register to select voice-specific indirect registers for reading or writing and to enable auto-increment mode. The Synthesizer Voice Select register can be written with 0 through 31 (0h to 1Fh) to select one of 32 voices to program. Auto-increment mode allows the General Index register (IGIDXR) to automatically increment with every write to either of the general I/O data ports (I8DP and I16DP).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Auto Increment	Reserved	Reserved	Voice Select				

Bit 7: **Auto Increment**. If set High, the value in the General Index register (IGIDXR) automatically increments with every write to the General 8-Bit I/O Data Port (I8DP) or the General 16-Bit I/O Data Port (I16DP). This bit is held Low when the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is Low.

Bits 6–5: Reserved.

Bits 4–0: **Voice Select**. Write a value from 0 through 31 (0h to 1Fh) to select one of 32 voices to program.

Indirect Registers

The synthesizer module has two types of indirect registers: global and voice-specific. Global registers affect the operation of all voices, and voice-specific registers affect the operation of only one voice. For information about programming these two types of indirect registers, see “Programming Voice-Specific Registers” on page 7-30.

Global Registers

SAVI—Synthesizer Active Voices

Address: P3XR+5h read/write; index IGIDXR=0Eh write or IGIDXR=8Eh read

Default: CDh

This register is needed only to remain compatible with the GUS. When the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is set High, this register does not affect operation. When SGMI[0] is Low, this register controls which voices produce an output and affect the output sample rate. The number of active voices can range from 14 to 32. With 14 active voices, the output sample rate is 44.1 kHz or a sample period of

approximately 22.7 µs. Each additional voice above 14 adds approximately 1.6 µs to the sample period. When SGMI[0] is Low, the frequency control values must be adjusted to compensate for the slower output sample rates when more than 14 voices are active. The programmed value equals the number of active voices minus 1. The programmed values of this register can range from CDh to DFh.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Reserved	Active Voices				

Bits 7–5: Reserved.

Bits 4–0: **Active Voices**. These bits indicate the number of active voices.

SVII—Synthesizer Voices IRQ

Address: P3XR+5h read; index IGIDXR=8Fh read

Default: E0h

This register indicates which voice needs interrupt service and what type of interrupt service is needed. Indexing this register with a value of 8Fh in the General Index register (IGIDXR) clears the IRQ bits in the voice-specific Synthesizer Volume Control or Synthesizer Address Control registers that caused the interrupt and also clears the *Volume Loop IRQ* and *Address Loop IRQ* bits in the IRQ Status register (UISR[6:5]).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Wavetable IRQ	Volume IRQ	Reserved	Voice Number				

Bit 7: **Wavetable IRQ**. When Low, the voice indicated in the *Voice Number* field has crossed an address boundary and has caused an interrupt.

Bit 6: **Volume IRQ**. When Low, the voice indicated in the *Voice Number* field has crossed a volume boundary and has caused an interrupt.

Bit 5: Reserved.

Bits 4–0: **Voice Number**. These bits indicate which voice needs interrupt service.

Note: All bits in this register except reserved bits are self-modifying.

SVIRI—Synthesizer Voices IRQ Read

Address: P3XR+5h read; index IGIDXR=9Fh read

Default: E0h

This register contains the same bits as the SVII register but can be read without clearing any internally stored interrupt conditions.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Wavetable IRQ	Volume IRQ	Reserved	Voice Number				

Bit 7: **Wavetable IRQ**. When Low, the voice indicated in the *Voice Number* field has crossed an address boundary and has caused an interrupt.

Bit 6: **Volume IRQ**. When Low, the voice indicated in the *Voice Number* field has crossed a volume boundary and has caused an interrupt.

Bit 5: Reserved.

Bits 4–0: **Voice Number**. These bits indicate which voice needs interrupt service.

Note: All bits in this register except reserved bits are self-modifying.

SGMI—Synthesizer Global Mode

Address: P3XR+5h read/write; index IGIDXR=19h write or IGIDXR=99h read

Default: 00h

This register controls modes of operation that affect all voices.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Global LFO Enable	Enhanced Mode

Bits 7–4: Reserved.

Bit 3: Reserved. This bit must be written to 0.

Bit 2: Reserved. This bit must be written to 0.

Bit 1: **Global LFO Enable**. If set High, enables operation of all LFOs.

Bit 0: **Enhanced Mode**. If set High, enables enhanced features added to the GUS capabilities.

SLFOBI—Synthesizer LFO Base Address

This register holds the base address for the locations of voice LFO parameters.

Address: P3XR+(4-5) read, write; index IGIDXR=1Ah write or IGIDXR=9Ah read

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved		LFO Base Address bits 23–10													

Bits 15–14: Reserved.

Bit 13–0: **LFO Base Address**. Bits 23–10 of the base address in local memory for the locations of voice LFO parameters.

Voice-Specific Registers

SUAI—Synthesizer Upper Address

Address: P3XR+5h read/write; index IGIDXR=10h write or IGIDXR=90h read; voice index SVSR=(00h through 1Fh)

Default: 00h

This register contains the upper bits of the local memory address for a voice's wavetable data. The upper bits of the address are added to the starting, ending, and current addresses for each voice. The upper address bits fix a voice in one of four 4-Mbyte memory spaces. With the upper address bits, the synthesizer can address a total of 16 Mbytes of memory for wavetable data. When the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is Low, SUAI is held to the default value.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Upper Address bits 23–22	

Bits 7–2: Reserved.

Bits 1–0: **Upper Address**. Bits 23–22 of the local memory addresses for a voice's wavetable data.

Synthesizer Starting Address Registers

The integer portion of this pair of registers specifies a real local memory boundary address when a voice is moving through wavetable data. These registers' value is less than the Synthesizer Address End registers' value. Bits 21–20 have been added to the GUS address to allow a voice to access 4 Mbytes of local memory instead of just 1 Mbyte. When the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is Low, address bits 21–20 are held Low.

SASHI—Synthesizer Address Start High

Address: P3XR+(4-5)h read/write; index IGIDXR=02h write or IGIDXR=82h read; voice index SVSR=(00h through 1Fh)

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Res.	Start Address Bits 21–7														

Bit 15: Reserved.

Bits 14–13: **Start Address Bits 21–20**. Extended integer portion of Start Address.

Bits 12–0: **Start Address Bits 19–7**. Part of integer portion of Start Address.

SASLI—Synthesizer Address Start Low

Address: P3XR+(4-5) read/write; index IGIDXR=03h write or IGIDXR=83h read; voice index SVSR=(00h through 1Fh)

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Start Address Bits 6–0							Start Address (Fraction) Bits 3–0				Reserved				

Bits 15–9: **Start Address Bits 6–0**. Part of integer portion of Start Address.

Bits 8–5: **Start Address (Fraction) Bits 3–0.** These four bits represent the upper bits of a 10-bit fractional portion that is fully represented in the Synthesizer Frequency Control register (SFCI).

Bits 4–0: Reserved.

Synthesizer Ending Address Registers

The integer portion of this pair of registers specifies a real local memory boundary address when a voice is moving through wavetable data. These registers' value is greater than the Synthesizer Address Start registers' value. Bits 21–20 have been added to the GUS address to allow a voice to access 4 Mbytes of local memory instead of just 1 Mbyte. When the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is Low, address bits 21–20 are held Low.

SAEHI—Synthesizer Address End High

Address: P3XR+(4-5) read/write; index IGIDXR=04h write or IGIDXR=84h read; voice index SVSR=(00h through 1Fh)

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Res.	End Address Bits 21–7														

Bit 15: Reserved.

Bits 14–13: **End Address Bits 21–20**. Extended integer portion of End Address.

Bits 12–0: **End Address Bits 19–7**. Part of integer portion of End Address.

SAELI—Synthesizer Address End Low

Address: P3XR+(4-5) read/write; index IGIDXR=05h write or IGIDXR=85h read; voice index SVSR=(00h through 1Fh)

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
End Address Bits 6–0							End Address (Fraction) Bits 3–0				Reserved				

Bits 15–9: **End Address Bits 6–0**. Part of integer portion of End Address.

Bits 8–5: **End Address (Fraction) Bits 3–0**. These four bits represent the upper bits of a 10-bit fractional portion that is fully represented in the Synthesizer Frequency Control register.

Bits 4–0: Reserved.

Synthesizer Current Address Registers

The integer portion of this pair of registers is the current location in local memory where a voice is fetching sample data. The fractional portion is used to interpolate between the sample in the location addressed by the integer portion of the address (address bits 21–0) and the sample in the location addressed by the value in bits 21–0 plus 1. This register pair is self-modifying and changes values as a voice moves through wavetable data in local memory. Bits 21–20 have been added to the GUS address to allow a voice to access 4 Mbytes of local memory instead of just 1 Mbyte. When the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is Low, address bits 21–20 are held Low. Bit 0 of the fractional portion of the address is used in interpolation but is not normally accessible for programming. Resetting and writing to the Synthesizer Address Low register (SALI) clears bit 0 of the fractional portion of the address.

SAHI—Synthesizer Address High

Address: P3XR+(4-5)h read/write; index IGIDXR=Ah write or IGIDXR=8Ah read; voice index SVSR=(00h through 1Fh)

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Res.	Address Bits 21–7														

Bit 15: Reserved.

Bits 14–13: **Address Bits 21–20**. Extended integer portion of Address.

Bits 12–0: **Address Bits 19–7**. Part of integer portion of Address.

Note: All bits in this register except reserved bits are self-modifying.

SALI—Synthesizer Address Low

Address: P3XR+(4-5)h read/write; index IGIDXR=Bh write or IGIDXR=8Bh read; voice index SVSR=(00h through 1Fh)

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Address Bits 6–0							Address (Fraction) Bits 9–1								

Bits 15–9: **Address Bits 6–0**. Part of integer portion of the Address.

Bits 8–0: **Address (Fraction) Bits 9–1**. Fractional bits used during interpolation.

Note: All bits in this register are self-modifying.

Synthesizer Effects Address Registers

When a voice operates as an effects processor, this pair of registers indicates the current address where data is being written in local memory. The data is from the effects accumulator linked to the effects processor. The effects address is integer only, because the data is being written. The integer portion of all wavetable addresses represents a local memory location.

SEAHI—Synthesizer Effects Address High

Address: P3XR+(4-5)h read/write; index IGIDXR=11h write or IGIDXR=91h read; voice index SVSR=(00h through 1Fh)

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Res.	Effects Address Bits 21–7														

Bit 15: Reserved.

Bits 14–0: **Effects Address Bits 21–7**. Part of integer portion of Address.

SEALI—Synthesizer Effects Address Low

Address: P3XR+(4-5)h read/write; index IGIDXR=12h write or IGIDXR=92h read; voice index SVSR=(00h through 1Fh)

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Effects Address Bits 6–0							Reserved								

Bits 15–9: **Effects Address Bits 6–0**. Part of integer portion of the Address.

Bits 8–0: Reserved.

Note: All bits in this register except reserved bits are self-modifying.

SFCI—Synthesizer Frequency Control

Address: P3XR+(4-5)h read/write; index IGIDXR=01h write or IGIDXR=81h read; voice index SVSR=(00h through 1Fh)

Default: 0400h

This register controls the rate at which a voice moves through local memory addresses. This sets the pitch of the voice. At the default value of decimal 1.0, the synthesizer plays back the wavetable data at the same rate as it was recorded. Bit 0 of the fractional portion has been added to increase the fractional frequency resolution to 10 bits. When the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is Low, fractional portion bit 0 is held Low.

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Integer Bits 5–0						Fraction Bits 9–0									

Bits 15–10: **Integer Bits 5–0**. Integer portion of frequency control value.

Bits 9–1: **Fraction Bits 9–1**. Fractional portion of frequency control value.

Bit 0: **Fraction Bit 0**. Fractional portion of frequency control value.

SFLFOI—Synthesizer Frequency LFO

Address: P3XR+5h read/write; index IGIDXR=17h write or IGIDXR=97h read; voice index SVSR=(00h through 1Fh)

Default: 00h

This register contains a value generated by the LFO generator used to modify the frequency of a voice (vibrato effect). When SGMI[0] is Low, SFLFOI is held to the default value.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
LFO Frequency Value							

Bits 7–0: **LFO Frequency Value**.

Note: All bits in this register are self-modifying.

SACI—Synthesizer Address Control

Address: P3XR+5h read/write; index IGIDXR=00h write or IGIDXR=80h read; voice index SVSR=(00h through 1Fh)

Default: 01h

This register controls how the synthesizer module addresses local memory and the local memory data width.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Wavetable IRQ*	Direction*	Wavetable IRQ Enable	Bidirectional Loop Enable	Loop Enable	Data Width	Stop 1	Stop 0

* Self-modifying bits

Bit 7: **Wavetable IRQ**. When High, the *Wavetable IRQ Enable* bit has been set High and the synthesizer address has crossed a boundary set by the start or end addresses. This bit is set Low when the voice's interrupt condition has been loaded into the Synthesizer Voices IRQ register (SVII) and a value of 8Fh has been written to the General Index register (IGIDXR). This bit can also be set Low to clear an interrupt or set High to cause an interrupt.

Bit 6: **Direction**. This bit sets the direction in which the wavetable data in local memory is addressed. If set Low, the address increases towards the boundary set by the Address End registers. If set High, the address decreases towards the boundary set by the Address Start registers. This bit is modified by internal logic when the *Bidirectional Loop Enable* bit (SACI[4]) is set High.

Bit 5: **Wavetable IRQ Enable**. If set High, the *Wavetable IRQ* bit (SACI[7]) is set High when an address boundary is crossed. If set Low, SACI[7] is cleared and cannot be set.

Bit 4: **Bidirectional Loop Enable**. If set High, the synthesizer current address changes direction at both the start and the end address boundaries. If set Low, the current address continues to loop in the same direction when end points are crossed. This bit has no effect if the *Loop Enable* bit (SACI[3]) is set Low.

Bit 3: **Loop Enable**. If set High, the address loops between address boundaries controlled by the *Bidirectional Loop Enable* (SACI[4]) and *Direction* (SACI[6]) bits. If set Low, the address moves to the boundary indicated by the start or end addresses or beyond if the *Enable PCM Operation* bit in the Synthesizer Volume Control register (SVCI[2]) is set High.

Bit 2: **Data Width**. This bit determines whether local memory is addressed as 16-bit or 8-bit data. If set High, local memory is addressed as 16-bit data. If set Low, local memory is addressed as 8-bit data.

Bit 1: **Stop 1**. If set High, stops voice activity. Both this bit and the *Stop 0* bit (SACI[0]) must be Low for a voice to operate.

Bit 0: **Stop 0**. This bit is modified by the address control logic. If a voice is set to stop at a boundary, this bit is set High when the boundary is crossed. Set it High to stop a voice. When read, it shows the status of a voice. Both the *Stop 1* bit (SACI[1]) and this bit must be Low for a voice to operate.

SVSI—Synthesizer Volume Start

Address: P3XR+5h read/write; index IGIDXR=07h write or IGIDXR=87h read; voice index SVSR=(00h through 1Fh)

Default: 00h

This register contains the low point of a volume ramp.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Volume Start							

Bits 7–0: **Volume Start**. An 8-bit value indicating the beginning volume of a volume ramp.

SVEI—Synthesizer Volume End

Address: P3XR+5h read/write; index IGIDXR=08h write or IGIDXR=88h read; voice index SVSR=(00h through 1Fh)

Default: 00h

This register contains the high point of a volume ramp.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Volume End							

Bits 7–0: **Volume End**. An 8-bit value indicating the ending volume of a volume ramp.

SVLI—Synthesizer Volume Level

Address: P3XR+(4-5)h read/write; index IGIDXR=09h write or IGIDXR=89h read; voice index SVSR=(00h through 1Fh)

Default: 0000h

This register contains the current value of the looping component of volume. Volume has three fractional bits that are needed for more resolution when choosing a slow rate of increment. These three bits do not affect the volume multiplication until an increment causes them to roll over into the least significant bit (LSB) of the *Looping Volume* field.

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Looping Volume Integer Bits 11–0												Looping Volume Fraction Bits 2–0			Res

Bits 15–4: **Looping Volume Integer Bits 11–0**. Current looping volume value.

Bits 3–1: **Looping Volume Fraction Bits 2–0**. Fractional volume value.

Bit 0: Reserved.

Note: All bits in this register except reserved bits are self-modifying.

SVRI—Synthesizer Volume Rate

Address: P3XR+5h read/write; index IGIDXR=06h write or IGIDXR=86h read; voice index SVSR=(00h through 1Fh)

Default: 00h

This register controls the rate at which the looping volume for a voice is incremented and the amount of the increment.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Volume Rate		Volume Increment					

Bits 7–6: **Volume Rate**. The rate at which the increment adds to the volume and the division of the increment value.
0:add *volume increment* every frame
1:add (*volume increment*)/8 every frame
2:add (*volume increment*)/8 every 8th frame
3:add (*volume increment*)/8 every 64th frame

Bits 5–0: **Volume Increment**. The amount of increment.

SVCI—Synthesizer Volume Control

Address: P3XR+5h read/write; index IGIDXR=0Dh write or IGIDXR=8Dh read; voice index SVSR=(00h through 1Fh)

Default: 01h

This register controls how the looping component of a voice's volume multiplier moves from volume start to volume end. This register also contains the *Enable PCM Operation* bit that controls local memory addressing to allow a voice to continuously play blocks of PCM data.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Volume IRQ*	Direction*	Volume IRQ Enable	Bidirectional Loop Enable	Loop Enable	Enable PCM Operation	Stop 1	Stop 0*

* Self-modifying bits

Bit 7: **Volume IRQ**. When High, the *Volume IRQ Enable* bit (SVCI[5]) has been set and the volume has crossed a boundary point set by the start or the end volume. This bit is cleared when the voice's interrupt condition has been loaded into the Synthesizer Voices IRQ register (SVII) and a value of 8Fh has been written to the General Index register (IGIDXR). This bit can also be set Low to clear an interrupt, or set High to cause an interrupt.

Bit 6: **Direction**. If set High, volume decreases. If set Low, volume increases. This bit is modified by internal logic when the *Bidirectional Loop Enable* bit (SVCI[4]) is set High.

Bit 5: **Volume IRQ Enable**. If set High, the *Volume IRQ* bit (SVCI[7]) is set when a volume boundary is crossed. If set Low, SVCI[7] is cleared and cannot be set.

Bit 4: **Bidirectional Loop Enable**. If set High, the volume changes directions at both the start and end volumes. If set Low, the volume continues to loop in the same direction when end points are crossed. This bit has no effect if the *Loop Enable* bit (SVCI[3]) is set Low.

Bit 3: **Loop Enable**. If set High, the volume loops between end points controlled by the *Bidirectional Loop Enable* (SVCI[4]) and *Direction* (SVCI[6]) bits. If set Low, the volume moves to a volume boundary and then the volume is held constant.

Bit 2: **Enable PCM Operation**. If set High, the address continues past a wavetable address boundary, which allows for continuous play of PCM data. For information about how this bit affects the interpolation process, see “Address Control” on page 7-9.

Bit 1: **Stop 1**. If set High, stops the change in the looping component of volume. Both this bit and the *Stop 0* (SVCI[0]) bit must be set Low to allow the looping component of volume to change.

Bit 0: **Stop 0**. This bit is modified by the volume looping logic. If volume is set to stop at a boundary, this bit is set High when the boundary is crossed. It can also be set High to stop volume looping. When read, it shows the status of volume looping. Both the *Stop 1* bit (SVCI[1]) and this bit must be set Low to allow the looping component of volume to change.

SVLFOI—Synthesizer Volume LFO

Address: P3XR+5h read/write; index IGIDXR=18h write or IGIDXR=98h read; voice index SVSR=(00h through 1Fh)

Default: 00h

This register contains a value generated by the LFO generator used to modify the volume of a voice. When the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is Low, SVLFOI is held to the default value.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Volume LFO							

Bits 7–0: **Volume LFO**.

Note: All bits in this register are self-modifying.

Synthesizer Offset Registers

These two register pairs control the placement of a voice in the stereo field. These registers have two modes of operation depending on the setting of the *Offset Enable* bit in the Synthesizer Mode Select register (SMSI[5]). If SMSI[5] is Low, bits 11–8 of the Synthesizer Right Offset register (SROI) are used to control both right and left offsets. In this mode, 16 positions of pan are available. A decimal value of 0 places the voice full left and a value of 15 places the voice full right. This mode is compatible with the GUS. If SMSI[5] is High, the Synthesizer Right Offset (SROI[15:4]) and Synthesizer Left Offset (SLOI[15:4]) registers contain the current right and left offset values that separately affect the right and left channel outputs of a voice. The final values for right and left offsets are contained in the Synthesizer Right Offset Final Value (SROFI[15:4]) and Synthesizer Left Offset Final Value (SLOFI[15:4]) registers. During the processing of a voice, the values in SROI and SLOI are incremented or decremented by 1 to move their value closer to the values in SROFI and SLOFI. SLOI affects operation only when SMSI[5] is High.

SROI—Synthesizer Right Offset

Address: P3XR+(4-5)h read/write; index IGIDXR=0Ch write or IGIDXR=8Ch read; voice index SVSR=(00h through 1Fh)

Default: 0700h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Right Offset												Reserved			

Bits 15–4: **Right Offset**. The voice's current right offset value.

Bits 3–0: Reserved.

Note: All bits in this register except reserved bits are self-modifying.

SROFI—Synthesizer Right Offset Final Value

Address: P3XR+(4-5)h read/write; index IGIDXR=1Bh write or IGIDXR=9Bh read; voice index SVSR=(00h through 1Fh)

Default: 0700h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Right Offset Final												Reserved			

Bits 15–4: **Right Offset Final**. The voice's final right offset value.

Bits 3–0: Reserved.

SLOI—Synthesizer Left Offset

Address: P3XR+(4-5)h read/write; index IGIDXR=13h write or IGIDXR=93h read; voice index SVSR=(00h through 1Fh)

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Left Offset												Reserved			

Bits 15–4: **Left Offset**. The voice's current left offset value.

Bits 3–0: Reserved.

Note: All bits in this register except reserved bits are self-modifying.

SLOFI—Synthesizer Left Offset Final Value

Address: P3XR+(4-5)h read/write; index IGIDXR=1Ch write or IGIDXR=9Ch read; voice index SVSR=(00h through 1Fh)

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Left Offset Final												Reserved			

Bits 15–4: **Left Offset Final**. The voice's final left offset value.

Bits 3–0: Reserved.

SEVI—Synthesizer Effects Volume

Address: P3XR+(4-5)h read/write; index IGIDXR=16h write or IGIDXR=96h read; voice index SVSR=(00h through 1Fh)

Default: 0000h

This register contains the current value of volume that controls the effects output of a voice. During the processing of a voice, the value in SEVI is incremented or decremented by 1 to move its value closer to the value in the Synthesizer Effects Volume Final Value register (SEVFI).

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Special Effects Volume												Reserved			

Bits 15–4: **Special Effects Volume**. The current special effects volume value.

Bits 3–0: Reserved.

Note: All bits in this register except reserved bits are self-modifying.

SEVFI—Synthesizer Effects Volume Final Value

Address: P3XR+(4-5)h read/write; index IGIDXR=1Dh write or IGIDXR=9Dh read; voice index SVSR=(00h through 1Fh)

Default: 0000h

This register contains the final value of volume that controls the effects output of a voice. During the processing of a voice, the value in the Synthesizer Effects Volume register (SEVI) is incremented or decremented by the least significant bit to move its value closer to the value in SEVFI.

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Special Effects Volume Final												Reserved			

Bits 15–4: **Special Effects Volume Final**. The final special effects volume value.

Bits 3–0: Reserved.

SEASI—Synthesizer Effects Output Accumulator Select

Address: P3XR+5h read/write; index IGIDXR=14h write or IGIDXR=94h read; voice index SVSR=(00h through 1Fh)

Default: 00h

This register controls which of the effects accumulators receive the effects output of a voice. Any, all, or none of the effects accumulators can be chosen. There are 8 effects accumulators, numbered 0 to 7. When the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is Low, SEASI is held to the default value.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Accumulator Selection							

Bits 7–0: **Accumulator Selection**. Any bit set High indicates that the corresponding effects accumulator receives the effects output of the voice.

SMSI—Synthesizer Mode Select

Address: P3XR+5h read/write; index IGIDXR=15h write or IGIDXR=95h read; voice index SVSR=(00h through 1Fh)

Default: 02h

Use this register to enable or disable various features within a voice and to control whether a voice is a signal generator, an effects processor, or off. Turning a voice off causes the voice to not access local memory, allowing more access to local memory for other functions. When the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is Low, SMSI is held to the default value.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ROM	μ-law	Offset Enable	Alternate Effects Path	Reserved	Reserved	Deactivate Voice	Effects Processor Enable

Bit 7: **ROM**. If set High, enables the voice's input data to come from external ROM.

Bit 6: **µ-law**. If set High, enables the voice's input data to be in µ-law format. If this bit is High, the *Data Width* bit of the Synthesizer Address Control register (SACI[2]) must be Low to get 8-bit samples from local memory.

Bit 5: **Offset Enable**. If set High, enables the Synthesizer Offset registers to separately control the left and right volume of the voice.

Bit 4: **Alternate Effects Path**. If set High, enables the alternate effects signal path for the voice.

Bits 3–2: Reserved.

Bit 1: **Deactivate Voice**. If set High, the voice is not processed.

Bit 0: **Effects Processor Enable**. If set High, the voice acts as an effects processor. During effects processing, the *Data Width* bit of the Synthesizer Address Control register (SACI[2]) must be set High to enable 16-bit accesses of local memory.

CHAPTER

15 LOCAL MEMORY CONTROL REGISTERS

LMBDR—LMC Byte Data

Address: P3XR+7 read, write

This register is an 8-bit port into local memory that is indexed by the I/O address counter value in the LMC I/O Address High (LMAHI) and LMC I/O Address Low (LMALI) registers. If the *Auto Increment* bit of the LMC Control register (LMCI[0]) is set High, then the I/O address counter value automatically increments by one with each access through this port.

LDMACI—LMC DMA Control

Index: P3XR+5 read, write; index IGIDXR=41h

Default: 00h

This register controls aspects of both interleaved and GUS-compatible DMA access to local memory.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Invert Most Significant Bit	DMA Terminal Count/Invert Bit 15	DMA IRQ Enable	DMA Rate Divider		DMA Width	Direction	Enable GUS-Compatible DMA

Bit 7: **Invert Most Significant Bit**. If set High, causes the most significant bit of the DMA data from system memory to local memory to be inverted. If set Low, the data passes unchanged. Bit 6 of this register controls whether the most significant bit is bit 7 or bit 15. This bit affects only GUS-compatible DMA, not interleaved DMA.

Bit 6: read: **DMA Terminal Count**. This bit has separate read and write functions. When read as High, it indicates a DMA terminal count (TC) interrupt is active; this read also clears the interrupt bit (the first time this bit is read after the interrupt is set, the value comes back as High; after that it is Low).

write: **Invert Bit 15**. If set High, specifies the data width of the DMA data from system memory to local memory as 16 bits wide. If set Low, specifies 8-bit data. This bit is used only in conjunction with the *Invert Most Significant Bit* field of this register. This bit can also be read from the *Invert Bit 15* bit of the LMC Control register (LMCI[6]).

Bit 5: **DMA IRQ Enable**. If set High, enables the ability for terminal count (TC) to cause an interrupt at the end of a block of system-memory/local-memory DMA. This interrupt becomes active if either the *Enable GUS-Compatible DMA* bit (LDMACI[0]) or the *Interleaved DMA Enable* bit of the LMC DMA Interleave Control register (LDICI[9]) is High, but not for codec DMA. This bit is ANDed with the output of the flip-flop that drives the TC interrupt; the output of this AND gate drives the *DMA Terminal Count IRQ* bit of the IRQ Status register (UISR[7]).

Bits 4–3: **DMA Rate Divider**. These bits control the rate at which transfers between local memory and system memory are allowed.

The times given are measured from the end of DMA acknowledge until the new DMA request is set; however, if the local memory cycle associated with the previous DMA cycle has not yet completed when the time is expired, then the logic waits for that memory cycle to complete before setting the DRQ signal.

The delay values for GUS-compatible DMA (controlled by this register) are
00:0.5 µs–1.0 µs
01:6 µs–7 µs
10:6 µs–7 µs
11:13 µs–14 µs

The delay values for interleaved DMA (controlled by the LMC DMA Interleave Control register (LDICI)) are:
00:the DRQ pin becomes active immediately after the write cycle to local memory is completed from the previous DMA cycle
01:0.5 µs–1.5 µs
10:6 µs–7 µs
11:13 µs–14 µs

Bit 2: **DMA Width**. This read-only bit specifies the data width of the DMA channel for system memory to or from local memory transfers. It is set High (16-bit) when the *DMA Select Channel 1* field of the DMA Channel Control register (UDCI[2:0]) is set to DMA request-acknowledge signals 5, 6, or 7. It is set Low (8-bit) for all others.

Bit 1: **Direction**. If set Low, specifies local memory DMA transfers to be reads of system memory and writes to local memory. If set High, specifies local memory DMA transfers to be reads of local memory and writes to system memory. This bit affects only GUS-compatible DMA, not interleaved DMA.

Bit 0: **Enable GUS-Compatible DMA**. If set High, causes DMA transfers between the system bus and local memory to occur (this does not affect codec DMA). There is a 0.5-µs–1.0-µs delay from the time that this bit is set High until the first DMA request is issued. The hardware resets this bit when the TC line is asserted.

LDSALI—LMC DMA Start Address Low

Index: P3XR+(4-5) read, write; index IGIDXR=42h

Default: 0000h

This 16-bit register specifies bits 19–4 of the GUS-compatible DMA address counter that points to local memory. Writing to this register automatically clears bits 3–0 of the DMA address counter.

LDSAHI—LMC DMA Start Address High

Index: P3XR+5 read, write; index IGIDXR=50h

Default: 00h

This register specifies bits 23–20 and 3–0 of the GUS-compatible DMA address counter that points to local memory. Address bits 3–0 are automatically cleared during writes to the LMC DMA Start Address Low register (LDSALI) for compatibility reasons. It is not legal to start DMA transfers from an odd byte address.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I/O Address Counter Bits 23–20				I/O Address Counter Bits 3–0			

LMALI—LMC I/O Address Low

Index: P3XR+(4-5) read, write; index IGIDXR=43h

Default: 0000h

This register specifies bits 15–0 of the I/O address counter that points to local memory. The rest of the address is in the LMC I/O Address High register (LMAHI). The corresponding data ports are the LMC Byte Data register (LMBDR) for byte accesses and the LMC 16-Bit Access register (LMSBAI) for 16-bit accesses. The least significant bit of this register is ignored for 16-bit accesses; it is not possible to write 16-bit data starting at an odd address. If the *Auto Increment* bit of the LMC Control register (LMCI[0]) is set High, then the I/O address counter automatically increments by one with each access through LMBDR and by two with each access through LMSBAI.

LMAHI—LMC I/O Address High

Index: P3XR+5 read, write; index IGIDXR=44h

Default: 00h

This register specifies bits 23–16 of the I/O address counter that points to local memory. The rest of the address is located in the LMC I/O Address Low register (LMALI). The corresponding data ports are the LMC Byte Data register (LMBDR) for byte accesses and the LMC 16-Bit Access register (LMSBAI) for 16-bit accesses. If the *Auto Increment* bit of the LMC Control register (LMCI[0]) is set High, then the I/O address counter automatically increments by one with each access through LMBDR and by two with each access through LMSBAI. If the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]) is set Low, then bits 23–20 of the address (LMAHI[7:4]) are reserved.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I/O Address Counter Bits 23–20				I/O Address Counter Bits 19–16			

LMSBAI—LMC 16-Bit Access

Index: P3XR+(4-5) read, write; index IGIDXR=51h

This is a 16-bit port into local memory that is indexed by the I/O address counter value in the LMC I/O Address Low (LMALI) and LMC I/O Address High (LMAHI) registers. If the *Auto Increment* bit of the LMC Control register (LMCI[0]) is set High, then the I/O address counter automatically increments by two with each access through this port. The least significant bit of LMALI is always treated as being 0 during accesses through this port to ensure that 16-bit local memory accesses are always aligned to even-byte boundaries.

LMCFI—LMC Configuration

Index: P3XR+(4-5) read, write; index IGIDXR=52h

Default: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				Suspend Mode Refresh Rate		Normal Mode Refresh Rate		ROM Configuration			Res	DRAM Configuration			

Bits 15–12: Reserved.

Bits 11–10: **Suspend Mode Refresh Rate**. See Table 15-1.

Bits 9–8: **Normal Mode Refresh Rate**. See Table 15-1.

Table 15-1 Refresh Rate Selection

Bit Values	Bits 11–10 - Suspend mode	Bits 9–8 - Normal mode
0 0	No refresh	15-µs refresh rate
0 1	62-µs refresh rate	62-µs refresh rate
1 0	125-µs refresh rate	125-µs refresh rate
1 1	Self-timed refresh	No refresh

Bits 7–5: **ROM Configuration**. Specifies the size of the four ROM banks.
0:128Kx16
1:256Kx16
2:512Kx16
3:1Mx16
4:2Mx16
5:reserved
6:reserved
7:reserved

Bit 4: Reserved

Bits 3–0: **DRAM Configuration.** Table 15-2 shows all values for DRAM configuration in byte quantities:

Table 15-2 DRAM Configuration Selection

Bits 3:0	Bank 3	Bank 2	Bank 1	Bank 0	Total
0	–	–	–	256K	256K
1	–	–	256K	256K	512K
2	256K	256K	256K	256K	1M
3	–	–	1M	256K	1.25M
4	1M	1M	1M	256K	3.25M
5	–	1M	256K	256K	1.5M
6	1M	1M	256K	256K	2.5M
7	–	–	–	1M	1M
Bits 3:0	**Bank 3**	**Bank 2**	**Bank 1**	**Bank 0**	**Total**
8	–	–	1M	1M	2M
9	1M	1M	1M	1M	4M
10	–	–	–	4M	4M
11	–	–	4M	4M	8M
12	4M	4M	4M	4M	16M

LMCI—LMC Control

Index: P3XR+5 read, write; index IGIDXR=53h

Default: 00h

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Invert Bit 15	Reserved	Reserved	Invert MSB of I/O Data	16-Bit I/O Data Select	DRAM/ROM Select	Auto-Increment

Bit 7: Reserved.

Bit 6: **Invert Bit 15**. This bit is read only. It provides CPU read access to the *DMA Terminal Count* bit of the LMC DMA Control register (LDMACI[6]). When High, the data width of the DMA data from system memory to local memory is specified to be 16 bits wide; when Low, the data width is 8 bits. This bit is used only in conjunction with LDMACI[6].

Bits 5–4: Reserved.

Bit 3: **Invert MSB of I/O Data**. If set High, causes the most significant bit (MSB) of I/O data written to local memory to be inverted. If set Low, the data passes unchanged. When access takes place through the LMC 16-Bit Access register (LMSBAI), bit 2 of this register controls whether the MSB is bit 7 or bit 15. When access takes place through the LMC Byte Data register (LMBDR), the MSB is assumed to be bit 7 regardless of the state of bit 2.

Bit 2: **16-Bit I/O Data Select**. This bit is used only in conjunction with LMCI[3] and the LMC 16-Bit Access register (LMSBAI). If set High, specifies that I/O data to be written to local memory is 16 bits wide. If set Low, specifies 8-bit data. When access takes place through the LMC Byte Data register (LMBDR), the data is assumed to be 8 bits wide regardless of the state of this bit.

Bit 1: **DRAM/ROM Select**. If set High, I/O memory cycles access ROM. If set Low, I/O memory cycles access DRAM.

Bit 0: **Auto Increment**. If set Low, specifies that I/O reads and writes to local memory through the LMC Byte Data register (LMBDR) and the LMC 16-Bit Access register (LMSBAI) do not automatically increment the I/O address counter. If set High, causes such accesses to increment the I/O address counter by 1 for accesses through LMBDR and by 2 for accesses through LMSBAI.

LMRFAI, LMPFAI—LMC Record/Playback FIFO Base Address

Index: P3XR+(4-5) read, write; record index IGIDXR=54h, play index IGIDXR=55h

Default: 0000h

These registers specify real (byte-oriented) address bits 23–8 of the base address of the local memory record and playback FIFOs. Writing to LMRFAI causes the local memory record FIFO-offset counter to reset to 0. Writing to LMPFAI causes the local memory playback FIFO-offset counter to reset to 0.

LMFSI—LMC FIFO Size

Index: P3XR+(4-5) read, write; index IGIDXR=56h

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Res	Res	Res	LMRF Enable	LMRF Size				Res	Res	Res	LMPF Enable	LMPF Size			

Bits 15–13: Reserved.

Bit 12: **LMRF Enable**. If set High, samples from the codec record FIFO are transferred into the local memory record FIFO (LMRF).

Note: When this bit is High, serial transfer mode is not available. For details, see "ICMPTI—Compatibility" on page 12-15.

Bits 11–8: **LMRF Size**. These bits specify the rollover point of the LMRF offset counter, that is, the size of the FIFO. The FIFO size is 2^(*LMRF Size* + 3) and can range from 8 bytes to 256 Kbytes.

Bits 7–5: Reserved.

Bit 4: **LMPF Enable**. If set High, samples from the local memory playback FIFO (LMPF) are transferred to the codec playback FIFO.

Note: When this bit is High, serial transfer mode is not available. For details, see "ICMPTI—Compatibility" on page 12-15.

Bits 3–0: **LMPF Size**. These bits specify the rollover point of the LMPF offset counter, that is, the size of the FIFO. The FIFO size is 2^(*LMPF Size* + 3) and can range from 8 bytes to 256 Kbytes.

LDICI—LMC DMA Interleave Control

Index: P3XR+(4-5) read, write; index IGIDXR=57h

Default: 0000h

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved					Invert MSB of Interleaved DMA Data	Interleaved DMA Enable	Data Width 16 Bits	Number of Interleaved Tracks					Size of Interleaved Tracks		

Bits 15–11: Reserved.

Bit 10: **Invert MSB of Interleaved DMA Data**. If set High, causes the most significant bit (MSB) of the DMA data from system memory to local memory to be inverted. If set Low, the data passes unchanged. Bit 8 of this register controls whether the MSB is bit 7 or bit 15. This bit affects only interleaved DMA, not GUS-compatible DMA.

Bit 9: **Interleaved DMA Enable**. While this bit is set High, interleaved DMA cycles occur. This bit is cleared by the hardware when the terminal count is reached after the DMA cycle associated with this function in which the TC pin is active.

Bit 8: **Data Width 16 Bits**. If set High, specifies that the interleaved samples are each 16-bits wide. If set Low, specifies 8-bit-wide data.

Bits 7–3: **Number of Interleaved Tracks**. 00h specifies 1 track, 01h specifies 2 tracks, and so on.

Bits 2–0: **Size of Interleaved Tracks**. The size of each track is 2^(9 + *Size of Interleaved Tracks*) samples. The range is from 512 bytes to 64Kbytes (regardless of whether an 8-bit or 16-bit DMA channel is selected).

LDIBI—LMC DMA Interleave Base

Index: P3XR+(4-5) read, write; index IGIDXR=58h

Default: 0000h

This 16-bit register specifies real address bits 23–8, which are ORed with the offset controlled by the LMC DMA Interleave Control register (LDICI). This register specifies real addresses, as described in “Accessing InterWave Registers” on page 4-2, regardless of the width of the DMA channel.

CHAPTER

16 GAME PORT AND MIDI PORT REGISTERS

Game Port Registers

GGCR—Game Control

Address: 201h write

Default: XXXX 0000

A write of any value to this register causes all four of the joystick X/Y position bits to go High and starts the X/Y position capacitor-charging cycle.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Joystick Button States				Joystick X/Y Position Flags			

Bits 7–4: **Joystick Button States**. These bits reflect the state of the four joystick buttons. If the bit is Low, the corresponding button is being pushed.

bit 4:Joystick 1, button 1
bit 5:Joystick 1, button 2
bit 6:Joystick 2, button 1
bit 7:Joystick 2, button 2

Bits 3–0: **Joystick X/Y Position Flags**. These bits are set High by writing to this register. When each of the joystick position capacitors charge to the joystick trim DAC voltage level, the corresponding bit goes Low.

bit 0:Joystick 1, X position
bit 1:Joystick 1, Y position
bit 2:Joystick 2, X position
bit 3:Joystick 2, Y position

GJTDI—Joystick Trim DAC

Address: P3XR+5 read, write; index IGIDXR=4Bh

Default: 1Dh

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Reserved	Joystick Trim DAC Level				

Bits 7–5: Reserved.

Bits 4–0: **Joystick Trim DAC Level**. Sets the level of the joystick trim DAC as shown in Table 16-1.

Table 16-1 Joystick Trim DAC Level Settings

Setting	Output at V_{CC}=5.0 volts	Output at V_{CC}=3.3 volts
00h	0.59 V ±5%	0.389 V ±5%
1Fh	4.52 V ±5%	2.98 V ±5%
Voltage per step	0.127 V	0.0837 V

These values vary linearly with V_{CC}.

MIDI Port Registers

GMCR—MIDI Control

Address: P3XR+0 write; read (if IVERI[3] is High)

Default: 0X0X XXX0

Note: When the* Register Read Mode *bit of the Version Number register (IVERI[3]) is High, this register becomes readable; if IVERI[3] is Low, then reads from this address provide the data in the MIDI Status register (GMSR). If IVERI[3] is High, a read of this register provides one bit each for the* MIDI Reset *field (GMCR[1:0]) and* Transmit Interrupt Enable *field (GMCR[6:5]): Bits 6 and 1 are unknown for these reads; bit 0 is Low if* MIDI Reset *is written with 11 binary (reset MIDI port); bit 5 is High if* Transmit Interrupt Enable *is written with 01 binary (IRQ enabled).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Receive Data Interrupt Enable	Transmit Interrupt Enable		Reserved	Reserved	Reserved	MIDI Reset	

Bit 7: **Receive Data Interrupt Enable**
0:Receive Interrupt disabled
1:Receive Interrupt enabled

Bits 6–5: **Transmit Interrupt Enable**
00:IRQ disabled
10:IRQ disabled
01:IRQ enabled
11:IRQ disabled

Bits 4–2: Reserved.

Bits 1–0: **MIDI Reset**
00:normal operation
10:normal operation
01:normal operation
11:reset MIDI port

The reset MIDI port command resets the MIDI receive logic, the MIDI receive FIFO, the MIDI Transmit Data register (GMTDR), the MIDI transmit logic, and the MIDI transmit-receive UART. It does not reset the MIDI Receive Data register (GMRDR). While this reset is active, the *MIDI Transmit Data Register Available* bit of the MIDI Status register (GMSR[1]) is forced Low. This command stays active until another I/O write changes this field to a value other than 11 binary. This field is implemented with only one flip-flop with combinatorial logic in front to decode the state: The flip-flop defaults at reset to the active state (reset MIDI port).

GMSR—MIDI Status

Address: P3XR+0 read

Default: 0X00 XX10

Note: When the* Register Read Mode *bit of the Version Number register (IVERI[3]) is High, the data in this register is not accessible.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
MIDI IRQ	Reserved	MIDI Overrun Error	MIDI Framing Error	Reserved	Reserved	MIDI Transmit Data Register Available	MIDI Receive Data Register Full

Bit 7: **MIDI IRQ**. This bit becomes High when one of *MIDI Receive Data Register (GMRDR) Full* (bit 0), *MIDI Overrun Error* (bit 5), or MIDI transmit IRQ is active (High). The interrupt equation for this bit is shown in Equation 16-1.

Equation 16-1 MIDI IRQ

$$MIRQ = GMCR[7] \bullet (GMSR[0] + GMSR[5]) + (GMCR[6:5] == (0, 1)) \bullet XMIT\_IRQ$$

The MIDI transmit IRQ becomes active when the MIDI UART is free to accept another byte of data to be transmitted. It is logically ORed with the *MIDI Reset* field of the MIDI Control register (GMCR[1:0]) to drive bit 1 of this register (GMSR[1]). Therefore, when MIDI reset is active, the MIDI transmit IRQ automatically becomes active.

Bit 6: Reserved.

Bit 5: **MIDI Overrun Error**. This bit becomes High when the MIDI receive FIFO fills up and an additional byte of MIDI data has been received. It is cleared by reading the MIDI Receive Data register (GMRDR).

Bit 4: **MIDI Framing Error**. This bit becomes High (active) as a result of reading the stop bit as other than a logic level 1. It is cleared by the receipt of a subsequent properly framed byte of MIDI data.

Bits 3–2: Reserved.

Bit 1: **MIDI Transmit Data Register (GMTDR) Available**. This bit is set High when the MIDI UART is ready to accept another byte of data. It is cleared to Low when a write to GMTDR initiates a data transfer. During a MIDI port reset—the *MIDI Reset* field of the MIDI Control register (GMCR[1:0]) is set

to 3h—this bit goes Low; after the reset, it goes High again. There is a buffer between the register data bus and the UART that can store one byte; therefore, after an initial byte is sent to GMTDR, this flag goes Low for only a short time (2 µs to 4 µs) before going High again. After the second byte is sent, the logic waits for the first byte to complete transmission and for the data in the buffer to be transferred to the UART before setting this flag.

Bit 0: **MIDI Receive Data Register (GMRDR) Full**. This bit is set High when there is a valid byte of data in the MIDI Receive Data register (GMRDR). This bit is cleared when the byte is read out of the GMRDR. If there is data in the MIDI receive FIFO, then this bit goes High again approximately 2 µs after the GMRDR is read.

GMTDR—MIDI Transmit Data

Address: P3XR+1 write

Writing to this register causes the 8-bit value written to be transmitted serially through the UART to the MIDITX pin in MIDI data format. There is a buffer between the register data bus and the UART that can store one byte; therefore, it is possible to send a second byte to this register shortly after an initial byte is sent. For details, see the descriptions of bit 0 in "GMSR—MIDI Status."

GMRDR—MIDI Receive Data

Address: P3XR+1 read

Default: FFh

This register contains the 8-bit value received in MIDI data format from the MIDIRX pin into the UART. If there is no data in the MIDI Receive FIFO, the value does not change after being read. If there is unread data in the MIDI Receive FIFO, then the next byte in the FIFO is transferred to this register after the read cycle.

Note: Writing to this register when MIDI reset is active—the* MIDI Reset *field of the MIDI Control register (GMCR[1:0]) set to 3h (11 binary)—causes the system to hang.

GMRFAI—MIDI Receive FIFO Access

Index: P3XR+5 write; index IGIDXR=5Eh

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Receive Data							

Bits 7–0: **Receive Data**. Writing to this port places data into the MIDI receive FIFO. It is assumed that no data from the UART is being passed into the FIFO while this command is being executed. Placing this data into the FIFO causes the MIDI receive data interrupt and status to be updated as if the data had come from the MIDIRX pin. This command requires between 2 µs and 4 µs to complete and holds the ISA bus while it is in progress.

Part 4

InterWave Game API and Driver Developer's Kit Reference

This section contains reference information about the InterWave Game API and the InterWave Driver Developer’s Kit (DDK) API.

CHAPTER

17 INTERWAVE GAME API REFERENCE

This chapter describes the InterWave Game API, which provides communication between applications and terminate-and-stay-resident programs (TSRs) or background applications and provides a library of game functions.

The Game API functions perform three types of actions:

- InterWave Program Communication—By calling the InterWave Game API, an application can communicate directly with a resident TSR or application running in the background. Also, a third party program can use this API to free and restore resources from currently loaded software. The application becomes part of a network of resident InterWave programs and can be accessed generically as such.
- InterWave Game Communication—Through the Game API, the caller can access resident game support.
- InterWave Broadcast—By listening to the communication between the application and the resident or background application through the Game API, an application can receive broadcast messages reflecting changes in the state of the machine or the sound card.

The interface to these functions is divided over two software interrupts. The bulk of the communication is done over the DOS Multiplex interrupt INT 2Fh. Most DOS programming guides have ample information on using this interrupt vector so that it's use does not conflict with any other programs. The rest of the functions, which include direct sound interfaces, are accessed through a given software interrupt. The interrupt number is determined when the Game API is successfully opened (see "INT 2Fh Function 21h: Game Device Open" on page 17-7). This interrupt vector is called the Game Vector.

Note: All vectors and pointers described in this chapter should be utilized from real mode only.

The implementation of the Game API differs depending on the purpose of the application using it. If the application is a game, it need only make calls to the API and should have no need to hook the INT 2Fh vector or install an InterWave Game API handler. However, if the application is, for example, a TSR for controlling some or all of the InterWave, it needs to make calls to and hook the INT 2Fh chain.

Game API Functions

This section describes each function of the InterWave Game API. Some sample code from a test program is included to show how the API should be used. Be aware that much of the description is from the point of view of both a program supporting the Game API *and* an application using the API.

The INT 2Fh Specification

The InterWave Game API functions use the Microsoft INT 2Fh specification. To use the INT 2Fh vector, load the INT 2Fh ID into the AH CPU register and the function number into the AL CPU register, then call INT 2Fh. The following descriptions provide details on other register inputs and the return values for each INT 2Fh function in the Game API.

INT 2Fh Function 0: INT 2Fh ID Install Check

Use Function 0 to determine if an INT 2Fh ID is currently in use. If a -1 (FFh) is returned in register AL, the ID is in use. Otherwise, any other value means the ID is open. The application can further determine if the program currently using that ID is an InterWave program by examining other registers looking for a particular InterWave stamp. If this stamp is found, then an InterWave program is currently using that INT 2Fh ID number and the calling application *must* use it.

This function is often used in a loop searching a range of possible INT 2Fh IDs until the first InterWave program is found or the end of the range is reached. In no InterWave program is found in the range, the first INT 2Fh ID that is not occupied is used and the new program is the first InterWave application to install itself on the INT 2Fh chain recognizing that ID. Search the range of CDh to ECh. This range provides ample room for an application to find a free ID.

If a resident program supporting this Game API is already installed on the INT 2Fh chain and receives a Function 0: Install Check, it must set register AL to FFH and register pair DX:SI to the string “ETEK” (DX=”ET” and SI=”EK”). Control is then be returned to the calling program and not chained to the previous INT 2Fh handler.

Called With

AH = the INT 2Fh ID
AL = 0
DX = “IW” (any value but "ET")
SI = “VE” (any value but "EK")

Returns

AL = FFh if the ID is occupied
DX = “ET” if the program using the ID is an InterWave application or TSR
SI = “EK” if the program using the ID is an InterWave application or TSR

Sample Code

```
/**
** test_2f_id() returns TRUE if ID is currently an occupied INT 2Fh ID.
**/
#define IW  0x4957    /* "IW" */
#define VE  0x5645    /* "VE" */
int test_2f_id(int id)
{
   union REGS regs;

   regs.h.al = 0; /* function 0 */
   regs.h.ah = id; /* INT 2Fh ID */
   regs.x.dx = IW; /* load dx:si with anything BUT "ETEK" */
   regs.x.si = VE;
   int86(0x2f,&regs,&regs);
   if (regs.h.al == 0xff /** This ID is occupied **/
      return TRUE;
   else
      return FALSE;
}
```

INT 2Fh Function 1: Get Number Of InterWave Programs / Get Installed Program ID Number

Use this function to determine the number of InterWave programs currently resident in memory and supporting the InterWave Game API. To be able to access each program individually and as single entities, each program must have an Installed Program ID number assigned to it which only it responds to and all other InterWave programs ignore. This function returns the next available Installed Program ID. Because the program IDs are numbered starting with 0, this number is also the number of installed InterWave programs.

If a program does not need to install its own handler but only needs to talk to or find a specific program in memory, it calls this function to get the number of resident InterWave programs. It then calls another INT 2Fh function (like Function 2: Get Program Status and Information) with Installed program ID numbers starting at 0 and up to but not equaling the number of resident InterWave programs until it finds the program it wants.

If a program needs to install its own Game API handler, it uses the new Installed Program ID number as its own. For example, the first InterWave program that loads realizes that it is the first when it does not find any other InterWave programs on the INT 2Fh chain through the use of INT 2Fh Function 0. It then assigns itself Installed Program ID number 0. If a second program then searches the INT 2Fh chain, it finds the first program already loaded and stores the INT 2Fh ID. It then calls INT 2Fh Function 1 with the INT 2Fh ID to determine the number of InterWave programs already loaded and using this Game API. Function 1 returns 1. The new program then stores 1 as its Installed Program ID number. Any program calling the Game API from this point on with the correct INT 2Fh ID number and Installed Program ID number 1, talks directly to this newly installed program.

If the new program receives a Function 1 call through its new Game API handler, it simply increments register BX and chains the INT 2Fh call.

This function is different from others in that control is chained to the previously installed INT 2Fh handler and not immediately returned to the calling handler.

Called With

AH = INT 2Fh ID
AL = 1
BX = 0

Returns

BX = number of loaded InterWave programs using the Game API

Sample Code

```
/**
** Calls INT 2Fh function 1 - Get Number of InterWave Programs
** This function returns the number of programs loaded that
** respond to the InterWave Game API
**/
int get_num_tsrs(int id)
{
   union REGS regs;
   regs.h.al = 1;
   regs.h.ah = id;
   regs.x.bx = 0;
   int86(0x2f,&regs,&regs);
   return regs.x.bx;
}
```

INT 2Fh Function 2: Get Program Status and Information

Use this function to get specific information about an InterWave program currently resident in memory. Only one program in memory responds to this function on each call. This function can return information such as a pointer to an identification string and a status bit field showing which resources the program uses and if it supports game functions.

If the calling program needs to locate a specific program or a specific type of program in memory, it can call this function with all valid Installed Program ID numbers (see “INT 2Fh Function 1: Get Number Of InterWave Programs / Get Installed Program ID Number”) and test the results of each call. For example, if the application is a DAC driver and wants to support the InterWave codec in enhanced mode, it would first poll each resident InterWave driver with this function testing for the codec bit to be on. If the codec bit is on for any driver or resident application, the calling program must call INT 2Fh Function 3: Suspend Program to free the codec hardware. After the calling program is finished with the codec and if it still has control of the codec, it must call INT 2Fh Function 4: Wake Program to restore the connection of the original program to the codec.

You must exercise caution when setting the hardware allocation bits for this function call. If any program specifies that it is using either section of the InterWave hardware, it *must* be prepared at any time to relinquish control of that hardware to another application requesting control through INT 2Fh Function 3: Suspend Program. The program must also be prepared to specify a usable base port, IRQ, and DMA channel (where applicable) to the calling application. Be aware that the InterWave synthesizer and codec can be programmed to use the same IRQ. Because it is not required that any chaining of handlers or IRQ callback be issued, the IRQ lines must be different when the control of a hardware section is given away. In some cases, that may mean that before the suspend call is granted, you may have to separate the IRQ channels through some hardware programming.

This function also determines if a specific program supports any game devices. If bit 2 of the CX CPU register is on, game device support is present. The version number must be supplied for backward compatibility of future versions of the game functions and is only valid if game devices are supported.

If bit 2 is on, the program supports at least one game device. A game device is a library of routines used to produce sound. Each device is different in capability and function. See “INT 2Fh Function 21h: Game Device Open” on page 17-7 for more details on each device.

Called With

AH = INT 2Fh ID
AL = 2
BX = Installed Program ID
BX = Game Functions Version (BCD BH.BL)
CX = Program Status
ES:DI = Pointer to ASCII ID string

Returns

Bits in the CS register are defined as follows (a 1 in the bit equals TRUE):

0 = Using synthesizer section of the InterWave card
1 = Using codec section of the InterWave card
2 = Supports at least one Game Device (see functions 21h and 22h)

Sample Code

```
/**
** Calls INT 2Fh function 2 - Get Program Status and Information
** It prints out the id string and whether or not it is using the
** CODEC and synth.
**/
int get_status(int id, installed_prog_id)
{
   union REGS regs;

   regs.h.al   =  2;
   regs.h.ah  =  id;
   regs.x.bx  =  installed_prog_id;
   int86x(0x2f, &regs, &regs, &regs);

   printf(" ID String ---> \"%Fs\"\n",MK_FP(sregs.es,regs.x.di));
   printf(" STATUS  ---> ");

   if (regs.x.cs & 0x01)
      printf ("Using SYNTH - ");
   else
      printf ("NOT Using SYNTH - ");

   if (regs.x.cx & 0x02)
      printf ("Using CODEC\n");
   else
      printf ("NOT Using CODEC\n");

   if (regs.x.cx & 0x04)
      printf("  Supports GAME Devices\n");
   else
      printf("  No Support For GAME Devices\b");

   return regs.x.cx;
}
```

INT 2Fh Function 3: Suspend Program

Use this function to suspend a resident program which is currently using the InterWave Game API and has allocated sections of the InterWave hardware. The calling program is responsible for waking up the suspended program after the caller is done with the resources. This wake up call can be performed with INT 2Fh Function 4.

The calling program can request the release of only one section of the InterWave card at a time. If control of either the codec or synthesizer sections of the InterWave card is requested, the base port, IRQ, and DMA channel (when applicable) are returned. This should and will be the standard approach for third party drivers to free resources. This will not, however, be the method by which a game allocates hardware. Under no circumstances should a game or any application not providing any InterWave specific services (i.e. emulation, hardware function library, etc.) use hardware directly without allocating a game device and only then if the game device is specified to be used in that manner. See “INT 2Fh Function 21h: Game Device Open” on page 17-7.

Called With

AH = INT 2Fh ID
AL = 3
BX = Installed Program ID
CX = Requested Device

Returns

AL = 0 if Suspended; otherwise the request failed
BX = Base Port
CL = IRQ
CH = DMA channel (for codec only)

Values for CX are:

01 = Just the synthesizer
02 = Just the codec

Sample Code

```
void suspend_prog(void)
{
   char status;
   union REGS regs;

   status = get status (int2f_id */
   if (status & BIT1    /* If bit 1 is on, CODEC is busy */
   {
      regs.h.al = 3;
      regs.h.ah = id;   /* int 2Fh id */
      regs.h.ah = id;   /* choose installed program id 0 */
      regs.x.bx = 0;    /* request SYNTH */
      int86x(0x2f, &regs, &regs, &regs);
      if (regs.h.al = = 0)
      {
         syn_base = regs.x.bx;
         syn_irq = (unsigned int) regs.x.cl;
      }
   }
}
```

Int 2Fh Function 4: Wake Program

Use this function to restore a program to its initial state. It should be used only after using INT 2Fh Function 3: Suspend Program to suspend the program. The suspended program is responsible for restoring all that was suspended from the initial suspend call.

If, for some reason, the wake call cannot pass, it is then up to the calling program to notify the user through some standard method that the InterWave card is now in an unstable state and appropriate action must be taken. For example, if the suspended program was SBOS and the calling program inadvertently broke SBOS so it could not somehow refresh its internal status, SBOS would return a fail status on an attempt to wake. The calling program could then call INT 2Fh function 2: Get Program Status and Information with SBOS's Installed Program ID number and format an error string for the user using the ID string pointed to by the ES:DI registers. It must be assumed that the state of the machine (including the InterWave card, PIC mask, etc.) after this call is in a significantly different and unpredictable state.

Called With

AH = INT 2Fh ID
AL = 4
BX = Installed Program ID

Returns

AL = 0 if the program is successfully awakened; otherwise, the wake call failed

INT 2Fh Function 5: Free Resident Device Driver

Use this function only to try to free a copy of the calling program itself. This function should never be called to free another program. To free another program of its resources, use INT 2Fh Function 3: Suspend Program. Support for this procedure is not required, but if the function is not required, a fail return code must still be returned.

Called With

AH = INT 2Fh ID
AL = 5
BX = Installed Program ID

Returns

AL = 0 if program is successfully freed; otherwise, the free attempt failed

INT 2Fh Function 21h: Game Device Open

Use this function to establish a connection to a program which has been found to support game functions. See "INT 2Fh Function 2: Get Program Status and Information" on page 17-4 for information on how to determine if a program supports the game functions. If the calling program needs a MIDI device, it should first try to open the MIDISIMPLE device. To open this device, the calling program loads a pointer to a null-terminated string containing "MIDISIMPLE" just as it would for a disk file. If the open attempt fails, the device is already open or is not implemented in the current version of the game function interface. If a previous application opened a device and did not close it, the device can not be reopened until it is closed using the correct handle number.

The real mode interrupt number passed back in register BX is the communication vector for the remainder of the game functions if a software interface is utilized. Each function

differs based on which device is opened. Therefore, each call to the Game Vector varies based on the device it is communicating with.

There are currently three devices which applications can use to generate sound with the Game API. These devices can be thought of as disk files. To use a device, you must open it and get a handle to it. When you are finished with it, you must close it. While the application uses the device, it is referred to through a handle number. As with a file, the handle is meaningless before the device is opened and after it is closed. The device cannot be reopened without first issuing a close. The supporting application must recognize which device each handle number represents and parse the incoming data appropriately.

MIDISIMPLE

The MIDISIMPLE device is a simple byte-oriented MIDI engine with two functions. Both functions process raw MIDI data. The data is passed to the processing application through the use of the Game Vector returned from INT 2Fh Function 21: Open Game Device. The functions for MIDISIMPLE are described in “MIDISIMPLE Functions” on page 17-9.

Called With

AH = INT 2Fh ID
AL = 21h
BX = Installed Program ID
ES:DI = pointer to “MIDISIMPLE”

Returns

AL = 0 if successfully allocated; otherwise, the allocation attempt failed
BX = Real Mode Interrupt Number
DX = Device Handle

MIDICOMPLEX

The MIDICOMPLEX device supports more complex MIDI processing including such things as dynamic patch loading. The MIDICOMPLEX specification is still in development.

Called With

AH = INT 2Fh ID
AL = 21h
BX = Installed Program ID”
ES:DI = pointer to “MIDICOMPLEX”

Returns

AL = 0 if successfully allocated; otherwise, the allocation attempt failed
BX = Real Mode Interrupt Number
DX = Device Handle

DIRECTCODEC

The DIRECTCODEC device allocates the codec and allows the application to do direct hardware writes to control digital output. The hardware specification for the codec can be found in the InterWave specification.

If the returned DMA channel is four, no DMA services will be used. This may be the case for PCMCIA implementations. In this case, the CODEC should be programmed using its FIFO capability. The size of the FIFO, in bytes, is returned in the SI register. If DMA is to be used, SI returns 0.

Called With

AH = INT 2Fh ID
AL = 21h
BX = Installed Program ID
ES:DI = pointer to “DIRECTCODEC”

Returns

AL = 0 if successfully allocated; otherwise, the allocation attempt failed
BX = Base Port
CL =
CH = DMA channel; - 4 for FIFO control
SI =
DX = Device Handle

INT 2Fh Function 22h: Game Device Close

Use this function when an application has finished using the previously opened game device from INT 2Fh Function 21h. This function fails if the device is not open or if the handle is invalid.

Called With

AH = INT 2Fh ID
AL = 22h
BX = Installed Program ID
DX = Device Handle

Returns

AL = 0 if the device is successfully closed; -1 if the device is not open its handle is bad

INT 2Fh Function 80h: Mixer Settings Changed Broadcast Message

This function should not be called by any application program unless that program controls the InterWave mixer. This function is intended as a broadcast message only for those applications that care if the mixer has changed values. This function is useful if an application is displaying the values in the mixer and wants to keep them up to date. If this message is received the new mixer values must be read from the InterWave card.

Called With

AH = INT 2Fh ID
AL = 80h

Sample Code

```
void interrupt int2f_handler(void)
{
   if (_AH == INT2f_ID && (_AL == 0x80)
   update_mixer-display();
}
```

MIDISIMPLE Functions

Both of these functions are valid only if the following is true: The handle used reflects a successfully opened MIDISIMPLE device and the Game Vector called is valid as described in “INT 2Fh Function 21h: Game Device Open” on page 17-7.

Game Vector Function 1: MIDI Byte Out

Use this function to send a single MIDI byte out to the game device to be parsed and immediately acted upon.

Called With

EAX = 1
BL = MIDI Byte
DX = Handle

Returns

EAX = 0 if the MIDI byte is successfully passed; -1 if the handle is bad

Game Vector Function 2: MIDI String Out

Use this function to send a string of MIDI bytes out to the game device to be parsed and immediately acted upon. The string should include no timing information. Put the number of bytes to be parsed in the ECX register.

Called With

EAX = 2
ES:EDI = pointer to MIDI string
ECX = byte count for the MIDI string
DX = handle

Returns

EAX = 0 if MIDI string is successfully passed; -1 if the handle is bad

CHAPTER

18 PROGRAMMING WITH THE DRIVER DEVELOPER'S KIT

The InterWave Driver Developer's Kit (DDK) is a set of low-level functions written in C and intended for the development of software for InterWave IC-based sound hardware. The DDK allows applications to be developed with little knowledge of the InterWave IC. The two primary goals of the DDK are:

- To provide an easy to use interface (API-like layer) to any InterWave IC-based hardware. This set of drivers could be used to write complete sound applications or it could be used to write small sound utilities. The DDK is particularly well suited as a test-development tool. Diagnostics software can been written with it to help debug prototype boards (OEMs).
- To serve as a tutorial guide to the InterWave IC. The DDK source files are written in C and some in-line assembly language and are well commented to make it easy for a developer to become familiar with the internal operation of the IC. The InterWave DDK includes sample code that illustrates how to program the InterWave IC to perform certain functions.

This chapter covers the following topics:

- Supported compilers
- DDK source files
- DDK include files
- DDK data types
- Basic structure of a DDK program
- Creating DDK libraries for specific C compilers
- The Plug and Play interface
- Accessing InterWave registers with the DDK

Supported Compilers

The InterWave DDK's set of low-level drivers have been compiled and tested using the following popular C compilers for the IBM PC and compatibles:

- Borland C++, Version 4.0
- Watcom C/C++, Version 9.5
- Microsoft Visual C++, Version 1.0
- Metaware High C/C++, Version 3.21
- Symantec C++, Version 6.1

DDK Source Files

The following files contain the C language source code from which the DDK libraries are built:

iwinit.c Definitions for functions that perform InterWave IC initialization-related tasks.

iwdma.c Definitions for functions that perform DMA-related tasks.

iwpnp.c Definitions for functions that perform Plug-and-Play related tasks. These functions can be used to debug Plug-and-Play functionality. Some of these functions are instrumental in detecting InterWave IC-based hardware in a PC system.

iwirq.c Definitions for functions that perform IRQ-related tasks. The handlers for the audio logical device functions in the IC are defined here. These handlers route the corresponding interrupt event to an application defined callback if one is registered.

iwcodec.c Definitions for functions that pertain to the InterWave codec.

iwmem.c Definitions for functions that perform local memory management tasks. These functions allow an application to allocate and deallocate Local Memory as needed while keeping track of existing memory.

iwvoice.c Definitions for functions that perform voice-related tasks.

iwutil.c Definitions for functions that perform general tasks. Some of these functions are called by other DDK functions.

For detailed information about a specific function, see the function reference pages in the remaining chapters of Part , "InterWave Game API and Driver Developer's Kit Reference".

DDK Include Files

The following include files are provided with the DDK.

Note: Always include the iwdefs.h, iwprotos.h, and iwcore.h files in your source file. These files contain include statements for the other files.

iwdefs.h Definitions for all of the symbolic constants used by the DDK. Browse through this file and become acquainted with it. Always include this file in your source code.

iwprotos.h Declarations for all of the DDK functions. Always include this file in your source code.

iwcc.h Definitions that allow the user to compile DDK source files with the compilers mentioned above.

iwtypes.h Definitions of data types.

iwcore.h Definition of the global variable `iw`. This variable is initialized to default IO, DMA, and IRQ resources that correspond to the Gravis UltraMax sound board.

DDK Data Types

As a matter of convenience and for readability purposes, the InterWave DDK has defined certain data types. These definitions are found in the **iwtypes.h** file. The following data types are defined:

BYTE **unsigned char**. Use this type to define 8-bit wide variables. It is normally used to store the contents of an InterWave register.

WORD **unsigned short.** Use this type to define 16-bit wide variables. The InterWave IC has a 16-bit data port (I16DP) with which these variables are typically used.

PORT **unsigned short**. Defined for code readability. It is equivalent to the type **WORD**.

ADDRESS **unsigned long**. Use this type to define 32-bit wide variables. It is typically used for local memory addresses.

DWORD **unsigned long**. Defined for code readability. It is equivalent to the type **ADDRESS**.

PVI Pointer to an interrupt vector.

PFV Pointer to a function which does not return a value (type void).

PFI Pointer to a function which returns an integer (type int).

DMA Structure type defined to facilitate the transfer of data between the InterWave hardware and system memory. The following members are defined within **DMA** structure variables:

`page` Type **PORT**, the address of the corresponding DMA channel's page register.

`addr` Type **PORT**, the address of the corresponding DMA's channel base address register. This register stores the base offset to system memory where the transfer is to take place. The page register together with the address register make up the physical address of the data.

`count` Type **PORT**, the address of the corresponding DMA channel's base count register in the DMA controller. The base count register is loaded with the number of bytes to be transferred minus one.

`single` Type **PORT**, the address of the DMA controller's Single-Mask Bit register. Writing to this register allows individual DMA channels to be disabled.

`mode` Type **PORT**, the address of the DMA controller's Mode Register. This register specifies the direction of transfer and whether to place the channel in auto-initialization mode.

`clear_ff` Type **PORT**, the address of the clear byte pointer flip-flop in the DMA controller. This flip-flop allows the reading or writing of the base address, current count, and base count registers. When the flip-flop is cleared, the next I/O access at this port

affects the lower byte. This I/O access toggles the flip-flop and the next access affects the higher byte.

`disable` Type **BYTE**, the bit value needed to disable the corresponding DMA channel by setting the mask for that channel at port `single`.

`enable` Type **BYTE**, the bit value needed to enable the corresponding DMA channel by clearing the mask for that channel at port `single`.

`write` Type **BYTE**, the bit pattern to indicate the transfer of data from a peripheral to system memory through the corresponding DMA channel.

`read` Type **BYTE**, the bit pattern to indicate the transfer of data from system memory to a peripheral through the corresponding DMA channel.

`cur_mode` Type **BYTE**, the bit pattern to be written to port `mode`.

`cur_page` Type **WORD**, points to the current DMA page in PC memory. This member is dynamically updated as needed during DMA transfers.

`cur_addr` Type **WORD**, the base address for the current DMA transfer.

`amnt_sent` Type **WORD**, the value indicating the number of bytes sent during a transfer.

`cur_size` Type **WORD**, the value indicating the number of bytes to be sent during the current transfer.

`nxt_page` Type **WORD**, points to the next page when a DMA transfer must be split into two transfers because of a DMA page overrun.

`nxt_addr` Type **WORD**, points to the next base address when a DMA transfer must be split into two transfers because of a DMA page overrun.

`nxt_size` Type **WORD**, the size of the next transfer when a DMA transfer must be split into two transfers because of a DMA page overrun.

`channel` Type **BYTE**, the number of the DMA channel. This value can range from 0 to 7.

`type` Type **BYTE**, the type of transfer to be carried out. Set this member to the following symbolic constants as needed:

- `DMA_READ` to transfer data from system memory to a peripheral
- `DMA_WRITE` to transfer data from a peripheral to system memory
- `AUTO_READ` to transfer data from system memory to a peripheral with auto-initialization of the DMA channel

- AUTO_WRITE to transfer data from a peripheral to system memory with auto-initialization of the DMA channel

`pc_ram` Type **void far ***, the physical address in PC memory where the data are to be written or read.

`local` Type **ADDRESS**, the DMA starting address within InterWave local memory.

`flags` Type **BYTE**, the status of the corresponding DMA channel. When the channel is busy, the DDK ORs in the symbolic constant DMA_BUSY. If the DMA transfer needs to be broken up into two transfers, the DDK ORs in DMA_SPLIT.

IRQ Structure type defined to facilitate programming the interrupt controller. The following members are defined within **IRQ** structure variables:

`mask` Type **BYTE**, the bit mask for the corresponding interrupt request channel.

`spec_eoi` Type **BYTE**, the bit pattern for specific end of interrupt for the corresponding IRQ channel.

`ocr` Type **BYTE**, the address of the Operation Command Register.

`imr` Type **BYTE**, the address of the Interrupt Mask Register (OCW1). This address is 0xA1 for the slave and 0x21 for the master.

IWAVE Structure type containing information which is critical to the operation of most DDK functions. Among the data stored here are the IO space the InterWave hardware is configured for, the addresses of callbacks to be called by interrupt handlers upon the occurrence of certain events, flags indicating the status of DMA operations, etc. The members are:

`pcodar` Type **PORT**, the base address for the codec registers.

`p2xr` Type **PORT**, the compatibility base address. This value is the address of the Mix Control register (UMCR).

`p3xr` Type **PORT**, the MIDI and synthesizer base address.

`p401ar` Type **PORT**, the base address for the MPU-401 emulation device. This value corresponds to the emulation address of the General Purpose Register 1 (UGP1I).

`p201ar` Type **PORT**, the address of the game logical device. This member is typically set to the legacy value of 201h.

`pataar` Type **PORT**, the base address for the ATAPI interface I/O space.

`pnprdp` Type **PORT**. If the IC is in Plug and Play mode, this member contains the address of the PNP Read Data Port register (PNPRDP) as defined by the Microsoft Plug and Play Specification.

`igidxr` Type **PORT**, the address of the General Index register (IGIDXR) at P3XR+3.

`i16dp`	Type **PORT**, the address of the General 16-bit I/O Data Port (I16DP) at P3XR+4.
`i8dp`	Type **PORT**, the address of General 8-bit I/O Data Port (I8DP) at P3XR+5.
`cdatap`	Type **PORT**, the address of the Codec Indexed Data Port register (CDATAP) at PCODAR + 1. All codec indexed registers are accessed through this data port.
`csr1r`	Type **PORT**, the address of the Codec Status Register 1 (CSR1R) at PCODAR + 2.
`cxdr`	Type **PORT**, the address of the Playback Data register (CPDR) and Record Data register (CRDR) at PCODAR + 3. Writing to this address places data in the playback FIFO. Reading this address removes data from the record FIFO.
`gmxr`	Type **PORT**, the address of the MIDI Control register (GMCR) or the MIDI Status (GMSR) register at P3XR + 0.
`gmxdr`	Type **PORT**, the address of the MIDI Transmit Data register (GMTDR) or the MIDI Receive Data register (GMRDR) at P3XR + 1.
`lmbdr`	Type **PORT**, the address of the LMC Byte Data register (LMBDR) at P3XR+7.
`svsr`	Type **PORT**, the address of the Synthesizer Voice Select register (SVSR) at P3XR+2.
`csn`	Type **BYTE**, the card select number (CSN) assigned to the InterWave IC-based board. This value can range from 1–255.
`cmode`	Type **BYTE**, reflects the current mode of codec operation. This member should always be set either to `CODEC_MODE1`, `CODEC_MODE2`, or `CODEC_MODE3`. The default value is CODEC_MODE3. This member is updated when the code mode is changed by the **IwaveCodecMode** function.
`dma1_chan`	Type **BYTE**, the number of the DMA channel associated with local memory DMA and codec record DMA. This value is the same as that in the PNP Audio DMA Channel 1 Select register (PUD1SI).
`dma2_chan`	Type **BYTE**, the number of the DMA channel associated with codec playback DMA. This value is the same as that in the PNP Audio DMA Channel 2 Select register (PUD2SI).
`ext_chan`	Type **BYTE**, the number of the DMA channel for the external device. This value is the same as that in the PNP CD-ROM DMA Select register (PRDSI).
`dma1`	Pointer to a structure of type **DMA** allocated to the local memory DMA and codec record DMA. This pointer is initialized to `NULL` but an application can register a structure here with the **IwaveRegisterDMA** or **IwaveSetInterface** functions.

`dma2` Pointer to a structure of type **DMA** allocated to the codec playback DMA. This pointer is initialized to `NULL` but an application can register a structure here with the **IwaveRegisterDMA** or **IwaveSetInterface** functions.

`synth_irq` Type **BYTE**, the IRQ number assigned to the synthesizer, codec, and compatibility function. This value is the same as that in the PNP Audio IRQ Channel 1 Select register (PUI1SI).

`midi_irq` Type **BYTE**, the IRQ number assigned to the MIDI functions. This value is the same as that in the PNP Audio IRQ Channel 2 Select register (PUI2SI).

`emul_irq` Type **BYTE**, the IRQ number assigned to the AdLib–Sound Blaster emulation functions. This value is the same as that in the PNP AdLib–Sound Blaster IRQ Select register (PSBISI).

`mpu_irq` Type **BYTE**, the IRQ number assigned to the MPU-401 emulation functions. This value is the same as that in the PNP MPU-401 IRQ Select register (PMISI).

`synth` Pointer to a structure of type **IRQ** defined to provide IRQ services for the synthesizer, codec, and compatibility functions. This pointer is initialized to `NULL` but an application can register a structure here with the **IwaveRegisterIRQ** or **IwaveSetInterface** functions.

`midi` Pointer to a structure of type **IRQ** defined to provide IRQ services for the MIDI functions. This pointer is initialized to `NULL` but an application can register a structure here with the **IwaveRegisterIRQ** or **IwaveSetInterface** functions.

`voices` Type **BYTE**, number of currently active voices.

`vendor` Type **BYTE**, first 32 bits of the PNP serial identifier stored in the serial EEPROM. This member is initialized at start-up time by the **IwaveOpen** function.

`free_mem` Type **ADDRESS**, the address of the first free block of local memory. This member is used by the local memory manager functions.

`reserved_mem`
Type **DWORD**, the amount in bytes of local memory reserved by an application.

`smode` Type **BYTE**, set either to `GUS_MODE` or to `ENHANCED_MODE` to reflect the mode of operation of the InterWave IC.

`size_mem` Type **WORD**, the actual amount of local memory in Kbytes available to the application. In GUS-Compatibility mode, the maximum possible amount is 1 Mbyte (1024 Kbytes). This member is initialized at start-up time by the **IwaveOpen** function.

`old_synth_vect`
: Type **PVI**, the address of the previous handler in the system Interrupt Vector Table (IVT) corresponding to the synthesizer IRQ. This vector must be restored when the DDK application exits.

`old_midi_vect`
: Type **PVI**, the address of the previous handler in the system IVT corresponding to the MIDI IRQ. This vector must be restored when the DDK application exits.

`old_ext_vect`
: Type **PVI**, the address of the previous handler in the system IVT corresponding to the external-device IRQ. This vector must be restored when the DDK application exits.

`midi_xmit_func`
: Type **PFV**, the address of a callback function to be called by the **IwaveHandler** service routine for the transmit function.

`midi_rcv_func`
: Type **PFV**, the address of a callback function to be called by the **IwaveHandler** service routine for the receive function.

`timer1_func`Type **PFV**, the address of a callback function to be called by the **IwaveHandler** service routine for the AdLib timer 1 interrupts.

`timer2_func`Type **PFV**, the address of a callback function to be called by the **IwaveHandler** service routine for the Adlib timer 2 interrupts.

`codec_dma_func`
: Type **PFV**, the address of a callback function to be called by the **IwaveHandler** service routine for the codec DMA interrupts.

`codec_timer_func`
: Type **PFV**, the address of a callback function to be called by the **IwaveHandler** service routine for the codec timer interrupts.

`codec_play_func`
: Type **PFV**, the address of a callback function to be called by the **IwaveHandler** service routine for the codec playback path interrupts.

`codec_rec_func`
: Type **PFV**, the address of a callback function to be called by the **IwaveHandler** service routine for the codec record path interrupts.

`play_dma_func`
: Type **PFV**, the address of a callback function to be called by the **IwaveHandler** service routine for the local memory DMA interrupts.

`rec_dma_func`

Type **PFV**, the address of a callback function to be called by the **IwaveHandler** service routine for the local memory DMA interrupts.

Basic Structure of a DDK Program

The code in Sample 18-1 illustrates the basic skeleton of a DDK program. Use this skeleton as a starting point for the development of a DDK application.

Including Header Files

The sample includes three header files. Every DDK application must include these files. The **iwcore.h** file defines the `iw` variable of type **IWAVE** and initializes it to its default values. The `iw` variable holds all relevant information about the InterWave IC-based hardware such as its assigned I/O space, DMA and IRQ channels, application-defined callback vectors, and so on, and is accessed by most of the DDK functions.

Initializing the DDK and the InterWave Hardware

A DDK program should always initialize the DDK as well as the InterWave IC by calling the **IwaveOpen** function or a similar function. **IwaveOpen** establishes a communication interface with the InterWave IC and, in the process, initializes the `iw` variable with the parameters of the function call and the configuration information read from the PNP serial EEPROM, if present. To obtain that configuration information, the software must know the vendor ID (the first 32 bits of the PNP serial identifier) as defined by the *PNP ISA Specification 1.0A*. The DDK requires the vendor ID to be stored in the `IWAVEID` environment variable. This vendor ID is the most reliable way for the DDK to correctly identify the InterWave IC-based sound board.

The code shown initializes the DDK and the InterWave IC-based hardware to operate in Enhanced mode with 32 active voices.

Most of the application code should follow the call to the **IwaveOpen** function and may also contain calls to other DDK functions. The final call to the **IwaveClose** function is essential and must always be carried out.

Sample 18-1 Basic Structure of a DDK Application

```
#include "iwdefs.h"             /* Always include these three files */
#include "iwprotos.h"
#include "iwcore.h"

...

void main()
{
  DMA dma1;            /* Optional - Codec record and local memory DMA */
  DMA dma2;            /* Optional - Codec playback DMA */
  IRQ irq1;            /* Optional - Synthesizer, codec, and compatibility IRQs */
  IRQ irq2;            /* Optional - MIDI IRQs */

  ...

  IwaveOpen(32, ENH_MODE);  /* Initialize the iw structure and the InterWave */

  ...                          /* Application Code */
```

```
  IwaveClose();                /* Close application */
}
```

Registering Callback Functions for Interrupt Events

Whenever an interrupt event takes place an application may need to gain control and perform certain actions or tasks depending on the source of the interrupt. The DDK provides the means for an application to register or install callback functions to be entered when particular interrupts occur.

To register a callback, an application can use the **IwaveSetCallback** function. An alternative way of registering a callback requires the programmer to become familiar with certain members of the `iw` variable. For example, to register a callback for interrupt events associated with the codec playback path, do either of the following steps:

- Issue the call: `IwaveSetCallback(PlayCallback, CODEC_PLAY_HANDLER)`, or
- Use the assignment: `iw.codec_play_func=(PFV)PlayCallback`

In this example, `PlayCallback` is the name of the callback function. The second call results in smaller code but requires that knowledge of the callback members of `iw`.

Establishing a DMA and IRQ Interface to the InterWave Hardware

If an application needs to conduct DMA transfers between the InterWave hardware and the PC system or to service interrupt requests, it must register variables of type **DMA** or **IRQ** to serve as an interface between the application and the hardware.

The audio logical device of the InterWave IC can use up to two DMA channels and up to two IRQ channels. The DMA Channel Control register (UDCI) determines the DMA channel configuration and the Interrupt Control register (UICI) determines the IRQ channel configuration.

The first IRQ channel, specified in UICI[2:0], is dedicated to interrupt requests originating in the synthesizer, codec, and compatibility sections of the IC. The second channel, specified in UICI[5:3], is dedicated to MIDI interrupts. An application can combine all interrupt sources to trigger interrupts through the first channel selected in UICI[2:0] by setting UICI[6] High and setting UDCI[7] Low. To combine all interrupt sources to trigger interrupts through the second channel selected in UICI[5:3], set both UICI[6] UDCI[7] High. If the application needs to handle both channels, then define two variables. For example, define `irq1` (first channel) and `irq2` (second channel) of type **IRQ** and register them with the call

```
IwaveRegisterIRQ(&irq1, &irq2)
```

If the application does not need to handle interrupt requests from a particular channel, set the corresponding argument to `NULL`. For example, if the application does not handle MIDI interrupts, then issue the call

```
IwaveRegisterIRQ(&irq1, NULL)
```

The first DMA channel, specified in UDCI[2:0], is dedicated to DMA requests for the codec record path and local memory. The second channel, specified in UDCI[5:3], is dedicated to DMA requests from the codec playback path. An application can combine both DMA channels to route all DMA requests to the first channel selected in UDCI[2:0] by setting

UDCI[6] High. If the application needs to handle DMA requests from both channels, then define two variables. For example, define `dma1` (first channel) and `dma2` (second channel) of type **DMA** and register them with the call

```
IwaveRegisterDMA(&dma1, &dma2)
```

If the application does not need to handle requests from a particular channel, set the corresponding argument to `NULL`. For example, if the application services only the codec playback FIFO through DMA, then issue the call

```
IwaveRegisterDMA(NULL, &dma2)
```

GUS-Compatibility Mode versus Enhanced Mode

The InterWave IC support programs written for the Advanced Gravis Ultrasound (GUS) sound boards. The InterWave IC also supports several enhanced features not supported by the GUS. For GUS programs to run correctly, the board must be in GUS-Compatibility mode where the enhanced features are not available.

Note: As AMD expects most InterWave applications to take advantage of the enhanced features, future releases of the DDK may not support GUS-Compatibility mode.

Applications that utilize the Enhanced mode of the InterWave IC should always place the InterWave IC-based hardware back into GUS-Compatibility mode before terminating so that subsequent GUS applications can run. The **IwaveClose** DDK function, which should be called at the end of the InterWave application, automatically resets the hardware to GUS-Compatibility mode.

Creating DDK Libraries For Specific C Compilers

The DDK includes **make** files for each of the tested C compilers. This section describes how to use the provided **make** file for each compiler.

Creating DDK Libraries with Borland C

With the Borland C/C++ compiler, use the **makeborl** file provided with the DDK. The command line for the **make** utility that creates a DDK library is

```
make -DMODEL=? -fmakeborl
```

where ? is one of `t`, `s`, `m`, `l` for the tiny, small, medium or large memory models, respectively, or `x` for the TNT-DOS extended library.

The library is created in the current directory. **makeborl** makes the following assumptions:

- The source file directory is **c:\iwave\c**.
- The include file directory is **c:\iwave\h**.
- The borland compiler is installed in the **c:\bc4** directory.

If any of these assumptions does not hold true for your programming environment, make the necessary modifications to the **makeborl** file.

The ***.obj** files from which the library is built are placed in the current directory but deleted as soon as the library is created. The name of the library is **iwbc?.lib** where ? is the single character code used in the command line to specify the memory model.

To compile an application with the 32-bit Borland C compiler, **BCC32**, first compile to object code and then link. For example, to compile the InterWave **peek.c** program, issue the following command to generate the *.obj file:

```
bcc32 -c -Ox -Ic:\iwave\h peek.c
```

Then issue the following command to link the program:

```
386link @bcc32.dos peek.obj iwpharl.obj -exe peek
```

Creating DDK Libraries with Microsoft Visual C++

With the Microsoft Visual C++ compiler, use the **makesoft** file provided with the DDK. The command line for the **nmake** utility that creates a DDK library is

```
nmake -fmakesoft MODEL=?
```

where ? is one of `T`, `S`, `M`, or `L` for the tiny, small, medium or large memory models, respectively.

The library is created in the current directory. **makesoft** makes the following assumptions:

- The source file directory is **c:\iwave\c**.
- The include file directory is **c:\iwave\h**.
- The Microsoft compiler is installed in the **c:\msvc** directory.

If any of these assumptions do not hold true for your programming environment, make the necessary modifications to the **makesoft** file.

The ***.obj** files from which the library is built are placed in the current directory but deleted as soon as the library is created. The name of the library is **iwmscx.lib** where x is the single character code used in the command line to specify the memory model.

Creating DDK Libraries with Watcom C/C++^{32}

With the Watcom C/C++^{32} compiler, use the **makewat** file provided with the DDK to create a protected mode library named **iwwc*x.lib** where * is the single character code representing the memory model. Follow the instructions within **makewat** to build a suitable library for the application.

Note: In Protected mode, the InterWave DDK's DMA module works only when using the flat memory model.

Creating DDK Libraries with MetaWare High C/C++

With the MetaWare High C/C++ compiler, use the following command to compile each of the **iw*.c** files in the DDK into a ***.obj** file:

```
hc386 -fsoft -c -O3 -Ic:\highc\inc -Ic:\iwave\c c:\iwave\c\iw*.c
```

To create the libraries, use the Phar Lap **386lib.exe** library utility. Issue the command:

```
386lib iwmw.lib @resp.dat
```

MetaWare High C/C++ supports DOS extenders other than Phar Lap; however, the InterWave DDK was only tested with Phar Lap's TNT DOS extender under this compiler. The above commands assume the High C compiler is installed in the **c:\highc** directory and that the DDK source files are in the **c:\iwave\c** directory.

Creating DDK Libraries with Symantec C/C++

With the Symantec C/C++ compiler, use the **makesym** file provided with the DDK. The command line for the **makesym** utility that creates a DDK library is

```
make -fmakesym MODEL=?
```

where ? is one of `c`, `s`, `m`, `l`, `x`, or `p` for the compact, small, medium, large, 32-bit DOSX, and Phar Lap memory models, respectively.

The library is created in the current directory. **makesym** makes the following assumptions:

- The source file directory is **c:\iwave\c**.
- The include file directory is **c:\iwave\h**.
- The Symantec compiler is installed in the **c:\sc** directory.

If any of these assumptions does not hold true for your programming environment, make the necessary modifications to the **makesym** file.

The ***.obj** files from which the library is built are placed in the current directory but deleted as soon as the library is created. The name of the library is **iwsc?.lib** where ? is the single character code used in the command line to specify the memory model.

The Plug and Play Interface

The InterWave IC is a PNP-compliant device capable of being configured by standard PNP software. The PNP protocol isolates and numbers each PNP adapter present in a PC system before arbitrating and assigning resources to all of them.

Full automation of the configuration processes is a difficult task because of the extensive base of old or legacy adapters which cannot report their resource requirements. Therefore, a means must be provided to inform the standard PNP software of the requirements needed by such cards. Legacy adapters require some intervention, such as a software driver, to register the resources used by the adapter and may possibly require the changing of jumpers to avoid conflicts with PNP-compliant adapters. The software driver is typically provided by a vendor in the absence of a PNP manager or a PNP BIOS.

In a non-PNP system (a system with no PNP manager, PNP BIOS, or PNP operating system), the InterWave IC PNP interface serves as no more than a set of configuration registers providing a jumperless solution to the automatic configuration problem.

Accessing InterWave Registers with the DDK

The DDK has two functions that can be used to access any register within the InterWave IC: **IwaveRegPeek** and **IwaveRegPoke**. These functions are generic and can be used in applications that read from or write to many registers; however, they should not be used in time-critical situations.

IwaveRegPeek reads a register. It takes as its only argument the mnemonic of the register whose contents are to be read. It returns a value of type WORD (16 bits). If the register is only 8 bits wide, the upper byte is 0. For example, to read the Local Memory DMA Control register (LDMACI), do the following:

```
WORD val;
...
val = IwaveRegPeek(LDMACI);
```

IwaveRegPoke writes to a register. For example, to write the value 0Bh to the Mix Control (UMCR) register, issue the call:

```
IwaveRegPoke(UMCR, 0x0B);
```

When using these functions always make sure that you are performing the correct operation for the specified register. Trying to read from a write-only register or write to a read-only register yields meaningless results.

The **iwdefs.h** file contains definitions for all of the InterWave register mnemonics. These definitions contain coded information that allows the register I/O functions to access the registers.

CHAPTER

19 DDK QUICK REFERENCE

The sections of this chapter list and briefly describe the DDK functions. For more detailed information about a particular DDK function, go to the reference page shown in parentheses at the end of each description.

System Control Functions

The DDK system control functions fall into one of three categories: initialization, utilities, and interrupt control.

Initialization Functions

These functions are defined in the **iwinit.c** and **iwpnp.c** files.

IwavePnpBIOS
Checks for the presence of a valid PNP BIOS. (20-9)

IwavePnpBIOS40
Queries the PNP BIOS for the number of PNP adapters in the system and the address of the PNP Read Data Port. (20-10)

IwavePnpPing Detects the presence of InterWave IC-based hardware in PNP card mode. (20-13)

IwaveOpen Initializes the `iw` variable and prepares the InterWave IC-based hardware for operation. (20-7).

IwaveGusReset
Resets the InterWave IC to GUS-Compatibility mode. (20-4)

IwaveClose Closes down the DDK and resets the InterWave IC-based hardware. (20-2)

IwavePnpPower
Enables or disables major sections of the InterWave IC. (20-14)

IwavePnpKey
Issues the PNP initiation key. (20-12)

IwavePnpIsol Performs the PNP isolation protocol to assign a unique card select number (CSN) to each PNP card in the system. (20-12)

IwavePnpIOcheck
Performs a conflict check on the I/O port to be used by a logical device. (20-11)

IwavePnpSerial
Reads the vendor ID and the serial number from the PNP serial EEPROM. (20-14)

IwavePnpPeek
Returns a specified number of resource data bytes from the serial EEPROM. (20-13)

IwavePnpWake
Issues a PNP wake command to the InterWave IC. (20-15)

IwavePnpDevice
Selects any of the five logical devices in the InterWave IC. (20-10)

IwavePnpActivate
Activates or deactivates any of the five logical devices in the InterWave IC. (20-9)

IwavePnpSetCfg
Reads the configuration members of an `iw` variable and configures the InterWave IC accordingly. (20-15)

IwavePnpGetCfg
Reads the PNP registers of the InterWave IC-based hardware with the specified card select number (CSN). (20-11)

IwaveSetInterface
Establishes a DMA and an IRQ interface to the InterWave IC-based hardware. (20-20)

IwaveRegisterDMA
Registers DMA structures with the DDK to establish an interface for InterWave DMA events. (20-16)

IwaveRegisterIRQ
Registers IRQ structures with the DDK to establish an interface for InterWave IRQ events. (20-17)

Utility Functions

These functions are defined in the **iwutil.c** file.

IwaveRegPeek
Reads an InterWave register. (20-13)

IwaveRegPoke
Writes to an InterWave register. (20-13)

IwaveAddrTrans
Performs the address translation required for 16-bit DMA channels or 16-bit data. (20-13)

IwaveDelay Introduces a delay or wait for a specified number of milliseconds. (20-13)

IwaveFreeDOS
Releases or deallocates real-mode or DOS memory. (20-13)

IwaveAllocDOS
Allocates real-mode memory. (20-2)

IwaveRealAddr
Translates the contents of a pair of synthesizer address registers into a real address. (20-16)

IwaveGetAddr
Retrieves the value of the local memory address pointer. (20-4)

WriteEnable Enables the write operation to the PNP serial EEPROM. (20-24)

WriteOPCode
Sends the write opcode to the PNP serial EEPROM. (20-24)

ReadOPCode
Sends the read opcode to the PNP serial EEPROM. (20-23)

IwavePokeEEPROM
Writes data to the PNP serial EEPROM. (20-15)

IwavePeekEEPROM
Reads the contents of the PNP serial EEPROM. (20-8)

GetSamplePosition
Reads the DMA count register and determines how many bytes have been sent to or received from the codec FIFOs. (20-1)

ReadWaveHeader
Reads the header from a ***.wav** file. (20-23)

_peek Reads an 8-bit hardware port and returns the value. (20-22)

_peekw Reads a 16-bit hardware port and returns the value. (20-22)

_poke Writes a byte to an 8-bit hardware port. (20-22)

_pokew Writes a word to a 16-bit hardware port. (20-23)

Interrupt Control Functions

These functions are defined in the **iwirq.c** file.

IwaveGetVect Retrieves the address of the current handler for a specified interrupt number. (20-4)

IwaveSetVect Specifies a new handler for a specified interrupt number in the Interrupt Vector Table (IVT). (20-21)

IwaveResetIvt
Restores the addresses corresponding to the interrupts assigned to the synthesizer and MIDI devices, thus restoring the system's Interrupt Vector Table (IVT) to its previous state. (20-18)

IwaveSetIvt Revectors the IRQ lines assigned to the MIDI and synthesizer devices. (20-20)

IwaveUmaskIrqs
Enables IRQs at the programmable interrupt controllers (PICs). (20-21)

IwaveMaskIrqs
Disables IRQs at the programmable interrupt controllers (PICs). (20-6)

IwaveHandleDma
Handles local memory DMA interrupts. (20-5)

IwaveHandleVoice
Handles voice interrupts. (20-6)

IwaveHandler The main IRQ router for both MIDI and synthesizer–codec interrupts. (20-6)

IwaveSynthHandler
The synthesizer interrupt request handler. (20-21)

IwaveHandleCodec
Handles codec interrupts. (20-5)

IwaveMidiHandler
The MIDI interrupt request handler. (20-7)

IwaveSetCallback
Registers a callback for interrupt events. (20-19)

IwaveDefFunc
The default callback function. (20-3)

Codec Functions

These functions are defined in the **iwcodec.c** file.

IwaveCodecMode
Specifies the codec mode of operation. (21-2)

IwaveCodecStatus
Retrieves the contents of any of the codec status registers: CSR1R, CSR2I, or CSR3I. (21-3)

IwaveCodecIrq
Enables or disables the codec interrupts. (21-2)

IwaveDataFormat
Specifies the playback or record data formats. (21-4)

IwaveSetFrequency
Specifies the playback or record sample frequency. (21-13)

IwaveCodecAccess
Selects either programmed I/O or DMA to service both of the codec FIFOs. (21-1)

IwavePlayAccess
Selects either programmed I/O or DMA to service the codec playback FIFO. (21-10)

IwaveRecordAccess
Selects either programmed I/O or DMA to service the codec record FIFO. (21-10)

IwaveCodecCnt
Loads the codec sample counters. (21-1)

IwaveCodecTrigger
Enables the codec record path, the codec playback path, or both. (21-3)

IwavePlayData
Sets up and starts a DMA transfer of data through the codec playback FIFO. (21-11)

IwaveRecordData
Sets up and starts a DMA transfer of data through the codec record FIFO. (21-12)

IwaveStopDma
Stops an active DMA transfer from the record FIFO or to the playback FIFO. (21-14)

IwaveDisableLineIn
Disables the stereo line inputs. (21-6)

IwaveEnableLineIn
Enables the stereo line inputs. (21-6)

IwaveDisableOutput
Disables the stereo line outputs. (21-6)

IwaveEnableOutput
Enables the stereo line outputs. (21-7)

IwaveEnableMicIn
Enables the stereo microphone inputs. (21-6)

IwaveDisableMicIn
Disables the stereo microphone inputs. (21-5)

IwaveLineMute
Mutes or unmutes the line inputs or outputs. (21-9)

IwaveLineLevel
Specifies the level of attenuation or gain for inputs or outputs. (21-1)

IwaveMonoMute
Mutes or unmutes the InterWave mono input or output. (21-10)

IwaveMonoAtten
Specifies the attenuation level for the mono input. (21-10)

IwaveInputSource
Selects the input source for the left or right ADC. (21-8)

IwaveInputGain
Specifies the gain level for the left or right ADC source. (21-8)

IwaveDacAtten
Specifies the attenuation level for the left or right DAC input. (21-4)

Synthesizer Functions

These functions are defined in the **iwvoice.c** file.

IwaveSynthGlobal
Controls the global mode of operation of the synthesizer (affects all voices). (22-7)

IwaveSynthMode
Switches the InterWave IC between GUS-Compatibility mode and Enhanced mode. (22-7)

IwaveReadyVoice
Prepares a selected voice for playback. It does not start the voice. (22-7)

IwaveSetLoopMode
Specifies the looping mode for both addressing and volume of a voice. (22-3)

IwaveStartVoice
Starts the voice. (22-5)

IwaveStopVoice
Stops the output of a voice. (22-6)

IwaveReadVoice
Returns the real local memory address from which a playing voice is currently fetching data. (22-2)

IwaveSetVoicePlace
Specifies the local memory address location from where a voice fetches data. (22-4)

IwaveSetVoiceEnd
Specifies a new local memory end boundary address for a voice. (22-4)

IwaveReadVolume
Returns the current looping volume value for a voice. (22-2)

IwaveSetVolume
Specifies the current looping volume value for a voice. (22-5)

IwaveRampVolume
Ramps the volume of a voice. (22-1)

IwaveStopVolume
Stops the codec timer. (22-6)

IwaveVoicePitch
Specifies the sampling frequency of a voice in Enhanced mode. (22-9)

IwaveVoiceFreq
Specifies the sampling frequency of a voice in GUS-Compatibility mode. (22-8)

IwaveVoicePan
Specifies a voice's position in the stereo field. (22-8)

Local Memory Functions

The local memory functions fall into two categories: memory management and DMA.

Memory Management Functions

These functions are defined in the **iwmem.c** file.

IwaveMemSize
Returns the number of Kbytes available as local memory attached to the InterWave IC. (23-9)

IwaveMemCfg
Determines the current DRAM configuration of the InterWave IC-based hardware. (23-6)

IwaveMemInit
Initializes local memory into a local memory pool for allocation and de-allocation. (23-7)

IwaveMemAvail
Returns the total amount of memory available in the local memory pool. (23-5)

IwaveMaxAlloc
Returns the size in bytes of the largest block of memory that can be still be allocated from the local memory pool. (23-5)

IwaveMemAlloc
Allocates a block of memory from the local memory pool. (23-5)

IwaveMemFree
Releases or de-allocates previously allocated blocks of local memory. (23-6)

IwaveMemPoke
Writes a byte of data to local memory. (23-8)

IwaveMemPokeW
Writes a word (16 bits) of data to local memory. (23-8)

IwaveMemPeek
Reads a byte of data from local memory. (23-7)

IwaveMemPeekW
Reads a word (16 bits) of data from local memory. (23-8)

IwavePokeBlock
Writes a block of data to local memory through the LMC Byte Data register (LMBDR). (23-10)

IwavePokeBlockW
Writes a block of data to local memory through the LMC 16-Bit Access register (LMSBAI). (page 23-10)

IwavePeekBlock
Reads a block of data from local memory through the LMC Byte Data register (LMBDR). (23-9)

IwavePeekBlockW
Reads a block of data from local memory through the LMC 16-Bit Access register (LMSBAI). (23-10)

DMA Functions

These functions are defined in the **iwdma.c** file.

IwaveDmaCtrl
Prepares the DMA controller on the PC for an impending DMA transfer. (23-5)

IwaveDmaPgm
Programs the DMA controller for an impending DMA transfer. (23-3)

IwaveDmaXfer
Programs the DMA controller and the InterWave IC for a DMA transfer to or from local memory, then starts the transfer. (23-4)

IwaveDmaIleaved
Programs the InterWave IC for an interleaved DMA transfer and starts the transfer. (23-5)

IwaveDmaPage

Sets up the InterWave IC and then initiates the transfer of up to one DMA page (64 Kbytes) to or from local memory. (23-3)

IwaveDmaWait

Blocks program activity until a specific DMA transfer is completed. (23-3)

IwaveGetDmaPos

Reads the count register of the DMA controller to determine the current position in a DMA transfer. (23-4)

IwaveDmaNext

If the data to be transferred crosses over one DMA page in system memory, the DMA handler calls this function to send the data in the second DMA page. (23-2)

IwaveDmaMalloc

Allocates memory that lies within a DMA page. (23-2)

CHAPTER

20 SYSTEM CONTROL DDK FUNCTIONS

GetSamplePosition — iwutil.c

Function

Reads the DMA count register and determines how many bytes have been sent to or received from the codec FIFOs.

Syntax

```
WORD GetSamplePosition(DMA *dma)
```

Remarks

GetSamplePosition reads the DMA count register and determines how many bytes have been sent to or received from the codec FIFOs. The *dma* argument points to a previously registered **DMA** structure for the relevant DMA channel.

Return Value

GetSamplePosition returns the number of bytes sent by the DMA transfer.

See Also

IwaveGetDmaPos

IwaveAddrTrans — iwutil.c

Function

Performs the address translation required for 16-bit DMA channels or 16-bit data.

Syntax

```
ADDRESS IwaveAddrTrans(ADDRESS local)
```

Remarks

IwaveAddrTrans performs the address translation required for 16-bit DMA channels or 16-bit data. If the InterWave IC is in GUS-Compatibility mode, **IwaveAddrTrans** shifts the first 17 bits of the value in *local* one bit to the right while preserving bits 18 and 19. In Enhanced mode, the function multiplies the real address value in *local* by two (a right shift). For more information, see “Address Translation” on page 8-8. **IwaveAddrTrans** determines the mode by looking at `iw.smode`. The address returned can then be written to an address register or pair of address registers. Call this function to determine the logical address if you are using 16-bit data while writing to synthesizer or local memory address counters or when writing to DMA address registers while using a 16-bit DMA channel.

Return Value

IwaveAddrTrans returns a logical address that can be written to any of the InterWave IC's address registers.

IwaveAllocDOS — iwutil.c

Function

Allocates real-mode memory.

Syntax

```
void *IwaveAllocDOS(WORD nbytes, WORD *pseg, WORD *psel)
```

Remarks

IwaveAllocDOS allocates *nbytes* of real-mode memory. *pseg* points to the real mode segment and *psel* points to the selector. This memory could be used as a DMA buffer. This function is for use with the Watcom protected mode compiler.

Return Value

Pointer to real-mode buffer.

See Also

IwaveFreeDOS

IwaveClose — iwinit.c

Function

Closes down the DDK and resets the InterWave IC-based hardware.

Syntax

```
void IwaveClose(void)
```

Remarks

IwaveClose closes down the DDK and resets the InterWave IC-based hardware. It should be the last function the DDK application calls. The function performs the following actions:

- Mutes all sound outputs.
- Resets the sound hardware to GUS-Compatibility mode.
- Multiplexes the synthesizer into the mixer section and enables codec mode 3 interrupts.
- Restores the system's Interrupt Vector Table (IVT).
- Disables the generation of Interrupts and DMA requests.

Return Value

None.

See Also

IwaveOpen, **IwaveGusReset**

IwaveDefFunc — iwirq.c

Function

The default callback function.

Syntax

```
void IwaveDefFunc(void)
```

Remarks

IwaveDefFunc is the default callback function. This function is called in an application that does not install any callbacks of its own.

Return Value

None.

IwaveDelay — iwutil.c

Function

Introduces a delay or wait for a specified number of milliseconds.

Syntax

```
void IwaveDelay(WORD count)
```

Remarks

IwaveDelay causes a delay in program execution for *count* milliseconds. *count* can range from 0 to 52.

Return Value

None.

IwaveFreeDOS — iwutil.c

Function

Releases or deallocates real-mode or DOS memory.

Syntax

```
void IwaveFreeDOS(WORD sel)
```

Remarks

IwaveFreeDOS releases or deallocates real-mode memory previously allocated with the **IwaveAllocDOS** function. *sel* is the selector value. This function is for use with the Watcom compiler protected mode compiler.

Return Value

None.

See Also

IwaveAllocDOS

IwaveGetAddr — iwutil.c

Function

Retrieves the value of the local memory address pointer.

Syntax

```
ADDRESS IwaveGetAddr(void)
```

Remarks

IwaveGetAddr retrieves the value of the local memory address pointer from the Local Memory Address High and Local Memory Address Low registers (LMAHI and LMALI).

Return Value

IwaveGetAddr returns the address read from the Local Memory Address High and Local Memory Address Low registers (LMAHI and LMALI).

IwaveGetVect — iwirq.c

Function

Retrieves the address of the current handler for a specified interrupt number.

Syntax

```
PVI IwaveGetVect(int int_no)
```

Remarks

IwaveGetVect retrieves the address of the current handler for the interrupt number specified in *int_no* from the Interrupt Vector Table (IVT).

Return Value

IwaveGetVect returns the address of the current handler.

See Also

IwaveSetVect

IwaveGusReset — iwinit.c

Function

Resets the InterWave IC to GUS-Compatibility mode.

Syntax

```
void IwaveGusReset(void)
```

Remarks

IwaveGusReset resets the InterWave IC to GUS-Compatibility mode by setting the *Reset GUS* bit of the GUS Reset register (URSTI[0]) Low for at least 22 µs and then pulling it back high. For details of what actions are accomplished by this reset, see “URSTI—GUS Reset” on page 12-14.

All Enhanced mode applications should reset GUS on exit to ensure that any subsequent GUS-only applications work correctly. The **IwaveClose** function calls **IwaveGusReset**.

Return Value

None.

See Also

IwaveClose

IwaveHandleCodec — iwirq.c

Function

Handles codec interrupts.

Syntax

```
static void IwaveHandleCodec(void)
```

Remarks

IwaveHandleCodec determines the source of an interrupt within the codec and activates the appropriate registered callback function to process the interrupt request.

Return Value

None.

See Also

IwaveMidiHandler, **IwaveSynthHandler**

IwaveHandleDma — iwirq.c

Function

Handles local memory DMA interrupts.

Syntax

```
static void IwaveHandleDma(void)
```

Remarks

IwaveHandleDma determines the source of an interrupt within the local memory controller and activates the appropriate registered callback function to process the interrupt request. This handler processes both normal and interleaved DMA related interrupts.

Return Value

None.

See Also

IwaveMidiHandler, **IwaveSynthHandler**

IwaveHandler — iwirq.c

Function

The main IRQ router for both MIDI and synthesizer–codec interrupts.

Syntax

```
static void IwaveHandler(void)
```

Remarks

IwaveHandler acts as a routing point to service all interrupt requests from the InterWave IC. It identifies the source of the interrupt by reading register the Interrupt Status register (UISR) or the Codec Status Register 1 (CSR1R) and then calls all necessary callback functions. This function reads these registers until all interrupts are serviced.

Return Value

None.

See Also

IwaveMidiHandler, **IwaveSynthHandler**

IwaveHandleVoice — iwirq.c

Function

Handles voice interrupts.

Syntax

```
static void IwaveHandleVoice(void)
```

Remarks

IwaveHandleVoice handles interrupts caused by a particular synthesizer voice when such voice has crossed either an address boundary or a volume boundary. This function reads the Synthesizer Voices IRQ register (SVII) to determine the number of the voice that caused the interrupt as well as the source of the interrupt (wavetable or volume ramp).

Return Value

None.

See Also

IwaveMidiHandler, **IwaveSynthHandler**

IwaveMaskIrqs — iwirq.c

Function

Disables IRQs at the programmable interrupt controllers (PICs).

Syntax

```
void IwaveMaskIrqs(void)
```

Remarks

IwaveMaskIrqs disables IRQs at the PICs by properly setting the mask register for the particular levels to be disabled at the input to the controller. For this purpose, it issues the appropriate OCW1. The interrupt levels disabled are those associated with *iw.synth_irq* and *iw.midi_irq*.

Return Value

None.

See Also

IwaveUmaskIrqs

IwaveMidiHandler — iwirq.c

Function

The MIDI interrupt request handler.

Syntax

```
static void interrupt IwaveMidiHandler(void)
```

Remarks

IwaveMidiHandler is the interrupt handler for the MIDI interrupt requests. This code takes over immediately after the interrupt occurs. It sends an EOI instruction to the programmable interrupt controllers (PICs) to allow further interrupts and then it calls a function that eventually routes the request to the appropriate function.

Return Value

None.

See Also

IwaveMidiHandler

IwaveOpen — iwinit.c

Function

Initializes the `iw` variable and prepares the InterWave IC-based hardware for operation.

Syntax

```
void IwaveOpen(BYTE voices, BYTE mode)
```

Remarks

IwaveOpen initializes the `iw` variable (an **IWAVE** structure) and prepares the InterWave IC-based hardware for operation. The `iw` variable is necessary for creating DDK applications for the InterWave IC. To initialize the IC, the function performs the following actions:

- Detects the InterWave IC-based board and extracts its configuration. During this process, the function assigns the card select number (CSN) to the board and determines the PNP Read Data Port address.

- Initializes the global `iw` variable to reflect the configuration of the InterWave IC-based board.
- Performs a GUS reset.
- Places the codec in mode 3 operation.
- Multiplexes the synthesizer and enables the codec mode 3 interrupts.
- Programs the synthesizer for Enhanced or GUS-Compatibility *mode* (`ENH_MODE` or `GUS_MODE`).
- Determines the size of local memory and sets *iw.size_mem*.
- Specifies the number of *voices* (from 0 to 31, where 0 = 1 voice) and sets `iw.voices`.
- Enables interrupts and DMA requests onto the ISA bus.
- Disables line inputs, microphone inputs, and line outputs.
- Writes 0 to the Left/Right Output Attenuation registers (CLOAI and CROAI) to prevent them from affecting mode 1 and mode 2 operation. For mode 3 operation, the software must write the appropriate values to these registers.

The `iw` variable contains pointers to the DMA and IRQ structures that provide an interface to the InterWave IC-based hardware. **IwaveOpen** initializes these pointers to `NULL`. If the application requires IRQ and DMA services, it should register the IRQ or DMA structures by a call to **IwaveSetInterface**, **IwaveRegisterDMA**, or **IwaveRegisterIRQ**.

Return Value

None.

See Also

IwaveClose, **IwaveGusReset**

IwavePeekEEPROM — iwutil.c

Function

Reads the contents of the PNP serial EEPROM.

Syntax

```
void IwavePeekEEPROM(BYTE *data)
```

Remarks

IwavePeekEEPROM reads the contents of the serial EEPROM through the PNP Serial EEPROM Control register (PSECI[3:0]). First, it places the serial EEPROM in direct-control mode (PSEENI[0]=1). Then, it fills the buffer pointed to by *data* with the contents of the serial EEPROM. The function assumes the InterWave IC is in the configuration state (a CSN has been assigned to the board). **IwavePeekEEPROM** takes care of all the signaling and control of the serial EEPROM by appropriately writing to PSECI[3:0]. See Sample 5-2 on page 5-24 for an example program using this function.

Note: This function programs the serial EEPROM unit KM93C66 which is a 4K-bit unit organized as 256x16. Compatible units can be programmed by this function.

Return Value

None.

See Also

IwavePokeEEPROM, **WriteEnable**, **WriteOPCode**, **ReadOPCode**.

IwavePnpActivate — iwpnp.c

Function

Activates or deactivates any of the five logical devices in the InterWave IC.

Syntax

```
void IwavePnpActivate(BYTE dev, BYTE state)
```

Remarks

IwavePnpActivate activates or deactivates any one of the five logical devices in the InterWave IC. Set *dev* to one of the following symbolic constants:

- `AUDIO` for the audio device (synthesizer, codec and compatibility sections)
- `EXT` for the external function (usually a CD-ROM)
- `GAME` for the game control device
- `EMULATION` for the AdLib–Sound Blaster emulation device
- `MPU401` for the MPU401 emulation device

To activate a device, set *state* to `ON`. To deactivate a device, set *state* to `OFF`.

Return Value

None.

See Also

IwavePnpDevice

IwavePnpBIOS — iwpnp.c

Function

Checks for the presence of a valid PNP BIOS.

Syntax

```
BYTE far *IwavePnpBIOS(void)
```

Remarks

IwavePnpBIOS checks for the presence of a valid PNP BIOS in the systems. It can be used within the initialization part of a program.

Return Value

IwavePnpBIOS returns a pointer to an installation check structure if there is a valid PNP BIOS in the system. Otherwise, it returns `NULL`.

See Also

IwavePnpBIOS40

IwavePnpBIOS40 iwpnp.c

Function

Queries the PNP BIOS for the number of PNP adapters in the system and the address of the PNP Read Data Port.

Syntax

```
WORD IwavePnpBIOS40(BYTE far *ptrS, BYTE far *cfig)
```

Remarks

If a PNP BIOS system is available, **IwavePnpBIOS40** queries the BIOS through function 40h to determine the total number of PNP cards in the system as well as the location of the PNP Read Data Port. This is a real-mode function and should not be used by protected mode programs. *ptrS* is a pointer to a PNP BIOS installation check structure. cfig is a pointer to a structure containing three elements:

- A member of type **BYTE** to store the revision number
- A member of type **BYTE** to store the total number of PNP cards
- A member of type **PORT** to store the address of PNPRDP

Return Value

IwavePnpBIOS40 returns 0 if it succeeds, non-0 if it fails.

See Also

IwavePnpBIOS

IwavePnpDevice iwpnp.c

Function

Selects any of the five logical devices in the InterWave IC.

Syntax

```
void IwavePnpDevice(BYTE dev)
```

Remarks

IwavePnpDevice selects any one of the five logical devices in the InterWave IC. Set *dev* to one of the following symbolic constants:

- `AUDIO` for the audio device (synthesizer, codec and compatibility sections)
- `EXT` for the external function (usually a CD-ROM)
- `GAME` for the game control device
- `EMULATION` for the AdLib–Sound Blaster emulation device
- `MPU401` for the MPU401 emulation device

Return Value

None.

See Also

IwavePnpActivate

IwavePnpGetCfg — iwpnp.c

Function

Reads the PNP registers of the InterWave IC-based hardware with the specified card select number (CSN).

Syntax

```
void IwavePnpGetCfg(void)
```

Remarks

IwavePnpGetCfg reads the configuration data for the I/O space, DMA and IRQ channels for all logical devices from the InterWave IC PNP interface, and loads that information into the configuration members of the `iw` variable. Call this function when an InterWave sound application needs to retrieve the InterWave IC configuration. **IwavePnpGetCfg** assumes that the card has been properly isolated, selected, and that it is in the configuration mode so that its configuration registers can be read.

Return Value

None.

See Also

IwavePnpSetCfg

IwavePnpIOCheck — iwpnp.c

Function

Performs a conflict check on the I/O port to be used by a logical device.

Syntax

```
PORT IwavePnpIOCheck(PORT base, BYTE no_ports)
```

Remarks

IwavePnpIOCheck performs a conflict check on the I/O port to be used by a logical device. The function receives the *base* address of the I/O range as well as the number of ports in the range (*no_ports*) and then performs the I/O check protocol. This function assumes that the logical device has been deactivated and that the PNP interface is in the configuration mode.

Return Value

IwavePnpIOCheck returns `IO_CHK` if no conflicts are detected. Otherwise, it returns the first address of the I/O port in conflict within the range.

IwavePnpIsol — iwpnp.c

Function

Performs the PNP isolation protocol to assign a unique card select number (CSN) to each PNP card in the system.

Syntax

```
BYTE IwavePnpIsol(PORT *pnprdp)
```

Remarks

IwavePnpIsol isolates each PNP card in the system by assigning a unique CSN to it. To accomplish this task, the function reads 72 pairs of 16-bit values from the PNP Read Data Port register (PNPRDP), which is initially assumed to be at 203h. These 72 pairs are translated into a 64-bit value and an 8-bit checksum value by converting each pair into a 1 or 0 bit value. For each pair read, the function checks for the 0x55–0xAA sequence. If the sequence is read, the function assumes a 1 in that bit position. Otherwise, a 0 is assumed. The first 64 bits produced by this process are converted into an 8-bit checksum value and compared to the last 8 bits produced. If the first pair of reads does not produce the 0x55–0xAA sequence or the checksum values do not match, the function assumes a PNPRDP conflict and relocates the port. If after several tries the same situation arises, then the function assumes that there are no PNP cards in the system.

The function stores the address of the PNPRDP register at *pnprdp*. All indexed registers in the Plug and Play section are read through this port.

Return Value

IwavePnpIsol returns PNP_ABSENT if it is not able to either successfully assign a location for PNPRDP or if it does not detect the 0x55–0xAA sequence in the first pair of reads.

See Also

IwavePnpKey, **IwavePnpWake**

IwavePnpKey — iwpnp.c

Function

Issues the PNP initiation key.

Syntax

```
void IwavePnpKey(void)
```

Remarks

IwavePnpKey issues the initiation key that places the Plug and Play logic into the configuration state. The PNP interface is quiescent at power up and must be enabled by software. First, the function resets the LFSR to its initial value by writing 00h to the PNP Index Address register (PIDXR) twice. Then, it writes the 32-byte initiation key to PIDXR.

Return Value

None.

See Also

IwavePnpWake

IwavePnpPeek iwpnp.c

Function

Returns a specified number of resource data bytes from the serial EEPROM.

Syntax

```
void IwavePnpPeek(PORT pnprdp, WORD bytes, BYTE *data)
```

Remarks

IwavePnpPeek reads *bytes* number of bytes of resource data from the serial EEPROM to the memory block pointed to by *data*. The function does NOT reset the serial EEPROM logic with successive calls so it can read the entire EEPROM by repeated calls. This function assumes that the InterWave IC is not in the sleep or wait-for-key states. Also, on the first call, if the calling program needs to read from the beginning of the serial EEPROM, the EEPROM logic must be reset with a PNP wake command. *pnprdp* is the I/O address of the PNP Read Data Port register (PNPRDP).

Return Value

None.

See Also

IwavePnpKey, **IwavePnpWake**

IwavePnpPing iwpnp.c

Function

Detects the presence of InterWave IC-based hardware in PNP card mode.

Syntax

```
BYTE IwavePnpPing(DWORD vendor)
```

Remarks

IwavePnpPing detects an InterWave IC-based sound board and returns its assigned card select number (CSN) so that an application can access the board's PNP interface and determine the board's current configuration. In conducting its search for the InterWave IC, the function uses the first 32 bits of the board's serial identifier (the *vendor* ID). The last four bits of the identifier represent a revision number for the particular product and are not included in the search. The function assigns the identifier to the `vendor` member of the `iw` variable. The calling application should check the revision bits to make sure it is compatible with the board. Also, this function determines the PNP data port and stores it in the `pnprdp` member of the `iw` variable. The **IwaveOpen** function calls this function.

Return Value

IwavePnpPing returns the CSN. Use this number to select the InterWave PNP interface at any time to read resource data or to reconfigure its registers. If the InterWave hardware is not detected then a value of 0 (FALSE) is returned.

See Also

IwavePnpOpen, **IwavePnpGetCSN**

IwavePnpPower iwpnp.c

Function

Enables or disables major sections of the InterWave IC.

Syntax

```
void IwavePnpPower(BYTE mode)
```

Remarks

IwavePnpPower enables or disables major sections of the InterWave IC. Disabling places the section in low-power mode and prevents it from loading the ISA bus. Power modes are controlled by writing the value of *mode* to the PNP Power Mode register (PPWRI). The function assumes that the PNP interface is in configuration mode. For a description of the sections of the IC which can be enabled or disabled and the value of *mode*, see “PPWRI—PNP Power Mode” on page 12-29.

Return Value

None.

IwavePnpSerial iwpnp.c

Function

Reads the vendor ID and the serial number from the PNP serial EEPROM.

Syntax

```
void IwavePnpSerial(PORT pnprdp, BYTE csn, BYTE *vendor, DWORD serial)
```

Remarks

IwavePnpSerial reads the first nine bytes of data from the PNP serial EEPROM through the *pnprdp* I/O address. First, it resets the EEPROM control logic by issuing a PNP wake command using *csn*. The function writes an ASCII string for the vendor ID in the VVVNNNN format into the char array pointed to by *vendor*. The function assigns the serial number to *serial*. The ninth byte is read but not used as it is invalid when the serial identifier is read through the PNP Resource Data register (PRESDI). This function assumes that the PNP interface is not in the wait-for-key state. Otherwise, meaningless results are obtained.

Return Value

None.

See Also

IwavePnpSearch

IwavePnpSetCfg — iwpnp.c

Function

Reads the configuration members of an `iw` variable and configures the InterWave IC accordingly.

Syntax

```
void IwavePnpSetCfg(void)
```

Remarks

IwavePnpSetCfg reads the configuration members of the `iw` variable and configures the I/O space, DMA and IRQ channels on the InterWave IC accordingly for all logical devices. This function is called when an InterWave sound application wishes to reconfigure the InterWave IC. **IwavePnpSetCfg** assumes that the card has been properly isolated and is in the configuration mode so that its configuration registers can be written.

Return Value

None.

See Also

IwavePnpGetCfg

IwavePnpWake — iwpnp.c

Function

Issues a PNP wake command to the InterWave IC.

Syntax

```
void IwavePnpWake(BYTE csn)
```

Remarks

IwavePnpWake issues a PNP wake command to the InterWave IC's PNP interface. If *csn* matches the card select number (CSN) stored in the PNP Card Select Number register (PCSNI), the particular card enters the configuration state. Otherwise, the card enters the sleep state. This function assumes the PNP interface is not in the wait-for-key state. The card select number can be from 1 to 255.

Return Value

None.

See Also

IwavePnpKey

IwavePokeEEPROM — iwutil.c

Function

Writes data to the PNP serial EEPROM.

Syntax

```
void IwavePokeEEPROM(BYTE *data)
```

Remarks

IwavePokeEEPROM programs the contents of the serial EEPROM through the PNP Serial EEPROM Control register (PSECI[3:0]). First, the function places the serial EEPROM in direct-control mode (PSEENI[0]=1). Then, it writes the serial EEPROM starting at address 0 with the contents of the buffer pointed to by *data*. **IwavePokeEEPROM** assumes the InterWave IC is in the PNP configuration state (a CSN has been assigned to the board). This functions takes care of all the signaling and control of the serial EEPROM by appropriately writing to PSECI[3:0]. See Sample 5-2 on page 5-24 for an example program using this function.

Note: This function programs the serial EEPROM unit KM93C66 which is a 4K-bit unit organized as 256x16. Compatible units can be programmed by this function.

Return Value

None.

See Also

IwavePeekEEPROM, **WriteEnable**, **WriteOPCode**, **ReadOPCode**.

IwaveRealAddr — iwutil.c

Function

Translates the contents of a pair of synthesizer address registers into a real address.

Syntax

```
ADDRESS IwaveRealAddr(WORD high, WORD low, BYTE mode)
```

Remarks

IwaveRealAddr translates the contents of a *high* (SAHI, SAEHI, SEAHI or SASHI) and *low* (SALI, SASLI, SAELI or SEALI) pair of synthesizer address registers into a real address, the actual address to local memory. The contents of the Synthesizer Address Control register (SACI) in *mode* tells the function whether local memory is being addressed as 8-bit or 16-bit data. The conversion from register or logical addresses to real addresses is based on the formulas shown in Table 8-6 on page 8-8.

Return Value

IwaveRealAddr returns a real local memory address.

IwaveRegisterDMA — iwinit.c

Function

Registers DMA structures with the DDK to establish an interface for InterWave DMA events.

Syntax

```
void IwaveRegisterDMA(DMA *dma1, DMA *dma2)
```

Remarks

IwaveRegisterDMA registers DMA structures for an application requiring a DMA interface to the InterWave IC. There are two possible DMA channels associated with the audio device in the InterWave IC. The first channel corresponds to codec record DMA events and DMA transfers between system memory and InterWave local memory. *dma1* points to a DMA structure related to DMA request services for that channel. The second channel corresponds to codec playback DMA events. *dma2* points to a DMA structure related to interrupt request services for that channel. It is possible to combine or route all DMA sources within the InterWave IC to trigger DMA requests through the first channel. The DDK takes this into account when setting up the interface. If the application does not need a DMA interface to the InterWave hardware, it should not call this function.

Return Value

None.

See Also

IwaveRegisterIRQ, **IwaveSetInterface**

IwaveRegisterIRQ — iwinit.c

Function

Registers IRQ structures with the DDK to establish an interface for InterWave IRQ events.

Syntax

```
void IwaveRegisterIRQ(IRQ *irq1, IRQ *irq2)
```

Remarks

IwaveRegisterIRQ registers IRQ structures for an application requiring an IRQ interface to the InterWave IC. There are two possible IRQ channels associated with the audio device in the InterWave IC. The first channel corresponds to interrupt events originating in the codec, synthesizer, Sound Blaster and AdLib sections of the InterWave IC. *irq1* points to an IRQ structure related to interrupt request services for that channel. The second channel corresponds to MIDI interrupt events. *irq2* points to an IRQ structure related to interrupt request services for that channel. It is possible to combine or route all interrupt sources within the InterWave IC to trigger interrupt events through either the first or the second channel only. The DDK takes this into account when setting up the interface. If the application does not process interrupt events from the audio device, it should not call this function.

Return Value

None.

See Also

IwaveRegisterDMA, **IwaveSetInterface**

IwaveRegPeek — iwutil.c

Function

Reads an InterWave register.

Syntax

```
WORD IwaveRegPeek(DWORD reg_name)
```

Remarks

IwaveRegPeek reads the register specified in *reg_name*. The register is specified by using a symbolic constant named after the actual register within the InterWave IC. To correctly use this function, the program must use the mnemonics for register names defined in the **iwdefs.h** header file. These mnemonics contain coded information used by the function to properly access the desired register. An attempt to read from a write-only register returns meaningless data.

Return Value

IwaveRegPeek returns the value stored in any readable register.

See Also

IwaveRegPoke

IwaveRegPoke — iwutil.c

Function

Writes to an InterWave register.

Syntax

```
void IwaveRegPoke(DWORD reg_name, WORD value)
```

Remarks

IwaveRegPoke writes a *value* to the register specified in *reg_name*. To correctly use this function, the program must use the mnemonics for register names defined in the **iwdefs.h** header file. These mnemonics contain coded information used by the function to properly access the desired register. This function does not guard against writing to read-only registers. The program must ensure that the writes are to valid registers.

Return Value

None.

See Also

IwaveRegPeek

IwaveResetIvt — iwirq.c

Function

Restores the addresses corresponding to the interrupts assigned to the synthesizer and MIDI devices, thus restoring the system's Interrupt Vector Table (IVT) to its previous state.

Syntax

```
void IwaveResetIvt(void)
```

Remarks

IwaveResetIvt restores the addresses corresponding to the interrupts assigned to the synthesizer and MIDI devices, thus restoring the system's Interrupt Vector Table (IVT) to its previous state. **IwaveClose** calls this function to de-install the **IwaveSynthHandler** and **IwaveMidiHandler** functions.

Return Value

None.

See Also

IwaveSetIvt

IwaveSetCallback — iwirq.c

Function

Registers a callback for interrupt events.

Syntax

```
PFV IwaveSetCallback(PVF handler, BYTE handle)
```

Remarks

IwaveSetCallback installs or registers the callback function pointed to by *handler* for the IRQ event specified in *handle*. It registers all callbacks in the `iw` variable. All callbacks are set at start up time to point to a default callback function which simply executes a return. This default setting prevents possible system crashes should spurious interrupts occur for events without registered callbacks.

Set handle to one of the following symbolic constants:

- `PLAY_DMA_HANDLER` for local memory DMA events (To local memory).
- `REC_DMA_HANDLER` for local memory DMA events (From local memory).
- `MIDI_TX_HANDLER` for MIDI transmit interrupt requests.
- `MIDI_RX_HANDLER` for MIDI receive interrupt requests.
- `TIMER1_HANDLER` for AdLib timer 1 interrupts.
- `TIMER2_HANDLER` for AdLib timer 2 interrupts.
- `WAVE_HANDLER` for interrupts caused by voices crossing wavetable boundaries.
- `VOLUME_HANDLER` for interrupts caused by voices crossing volume boundaries.
- `CODEC_TIMER_HANDLER` for codec timer interrupts.
- `CODEC_PLAY_HANDLER` for codec play path interrupts.
- `CODEC_REC_HANDLER` for codec record path interrupts.

Return Value

IwaveSetCallback returns the address of the previous registered callback for the particular type of event.

IwaveSetInterface — iwinit.c

Function

Establishes a DMA and an IRQ interface to the InterWave IC-based hardware.

Syntax

```
void IwaveSetInterface(DMA *dma1, DMA *dma2, IRQ *irq1, IRQ *irq2)
```

Remarks

IwaveSetInterface initializes and registers particular DMA and IRQ structures that are intended to carry out DMA and IRQ handling operations between the system and the InterWave IC-based sound board. The four arguments represent pointers to these structures. **IwaveSetInterface** should be called after the call to **IwaveOpen**. If an application does not need the services supported by one of the structures, then the argument corresponding to that service should be set to `NULL`. If an application needs only a DMA interface, it should call the **IwaveRegisterDMA** function. If an application needs only an IRQ interface, it should call the **IwaveRegisterIRQ** function.

dma1 points to a DMA structure related to codec record transfers and to transfers between system memory and local memory.

dma2 points to a DMA structure related to codec playback transfers.

irq1 points to an IRQ structure related to interrupt request services for the codec, the synthesizer, and compatibility emulation.

irq2 points to an IRQ structure related to interrupt request services for the MIDI interface.

Return Value

None.

See Also

IwaveRegisterDMA, IwaveRegisterIRQ

IwaveSetIvt — iwirq.c

Function

Revectors the IRQ lines assigned to the MIDI and synthesizer devices.

Syntax

```
void IwaveSetIvt(void)
```

Remarks

IwaveSetIvt revectors the interrupt requests *iw.synth_irq* and *iw.midi_irq* respectively to point to the **IwaveSynthHandler** and **IwaveMidiHandler** functions. The **IwaveRegisterIRQ** function calls this function to install these two handlers.

Return Value

None.

See Also

IwaveResetIvt

IwaveSetVect — iwirq.c

Function

Specifies a new handler for a specified interrupt number in the Interrupt Vector Table (IVT).

Syntax

```
void IwaveSetVect(int int_no, PVI isr)
```

Remarks

IwaveSetVect specifies a new handler for the interrupt specified in *int_no* in the Interrupt Vector Table (IVT). *isr* contains the address for the new handler.

Return Value

None.

See Also

IwaveGetVect

IwaveSynthHandler — iwirq.c

Function

The synthesizer interrupt request handler.

Syntax

```
static void interrupt IwaveSynthHandler(void)
```

Remarks

IwaveSynthHandler is the interrupt handler for the interrupt requests originating in the codec, synthesizer, and compatibility sections of the InterWave IC. This code takes over immediately after the interrupt occurs. It sends an EOI instruction to the programmable interrupt controllers (PICs) to allow further interrupts and then it calls a function that eventually routes the request to the appropriate function.

See Also

IwaveMidiHandler

IwaveUmaskIrqs — iwirq.c

Function

Enables IRQs at the programmable interrupt controllers (PICs).

Syntax

```
void IwaveUmaskIrqs(void)
```

Remarks

IwaveUmaskIrqs enables IRQs at the PICs by properly setting the mask register for the particular levels to be enabled at the input to the controller. For this purpose, it issues the appropriate OCW1. The interrupt levels enabled are those associated with *iw.synth_irq* and *iw.midi_irq*.

Return Value

None.

See Also

IwaveMaskIrqs

_peek — iwutil.c

Function

Reads an 8-bit hardware port and returns the value.

Syntax

```
BYTE _peek(PORT port)
```

Remarks

_peek reads the 8-bit hardware port at address *port*.

Return Value

_peek returns the 8-bit value read from the port.

_peekw — iwutil.c

Function

Reads a 16-bit hardware port and returns the value.

Syntax

```
WORD _peekw(PORT port)
```

Remarks

_peekw reads the 16-bit hardware port at address *port*.

Return Value

_peekw returns the 16-bit value from the port.

_poke — iwutil.c

Function

Writes a byte to an 8-bit hardware port.

Syntax

```
void _poke(PORT port, BYTE value)
```

Remarks

_poke writes an 8-bit *value* to the hardware port at address *port*.

Return Value

None.

_pokew iwutil.c

Function

Writes a word to a 16-bit hardware port.

Syntax

```
void _pokew(PORT port, WORD value)
```

Remarks

_pokew writes a 16-bit *value* to the hardware port at address *port*.

Return Value

None.

ReadOPCode iwutil.c

Function

Sends the read opcode to the PNP serial EEPROM.

Syntax

```
void ReadOPCode(void)
```

Remarks

ReadOPCode sends the bit string that specifies a read operation (100) to the KM93C66 serial EEPROM. This code must precede the address bit string. **IwavePeekEEPROM** calls **ReadOPCode** for each read.

Return Value

None.

See Also

IwavePokeEEPROM, **IwavePeekEEPROM**, **WriteEnable**, **WriteOPCode**

ReadWaveHeader iwutil.c

Function

Reads the header of a **.WAV** sound file.

Syntax

```
BOOL ReadWaveHeader(BYTE *fname, WAV *wav)
```

Remarks

ReadWaveHeader reads the header of the *fname* **.WAV** sound file and places the information (sample rate, length of sound data, data width, etc.) inside the elements of a structure pointed to by *wav*. This function can be used as part of a wave file player.

Return Value

ReadWaveHeader returns TRUE if it succeeds and FALSE if it fails.

WriteEnable — iwutil.c

Function

Enables the write operation to the PNP serial EEPROM.

Syntax

```
void WriteEnable(void)
```

Remarks

WriteEnable enables the write operation to the PNP serial EEPROM. The KM93C66 always powers up in a write-disabled state to protect itself against accidental writes. After power-up, any write operation to the EEPROM must be preceded by a write enable. **IwavePokeEEPROM** calls **WriteEnable** to make sure that all writes are successful. This write-enabled condition is preserved until power is removed or the write disable instruction is sent. To keep the code small, **IwavePokeEEPROM** does not issue a write disable instruction.

Return Value

None.

See Also

IwavePokeEEPROM, **ReadEnable**, **WriteOPCode**, **ReadOPCode**.

WriteOPCode — iwutil.c

Function

Sends the write opcode to the PNP serial EEPROM.

Syntax

```
void WriteOPCode(void)
```

Remarks

WriteOPCode sends the bit string that specifies a write operation (101) to the KM93C66 serial EEPROM. This code must precede the address bit string. **IwavePokeEEPROM** calls **WriteOPCode** for each write.

Return Value

None.

See Also

IwavePokeEEPROM, **IwavePeekEEPROM**, **WriteEnable**, **ReadOPCode**.

CHAPTER

21 CODEC/MIXER DDK FUNCTIONS

IwaveCodecAccess — iwcodec.c

Function

Selects either programmed I/O or DMA to service both of the codec FIFOs.

Syntax

```
void IwaveCodecAccess(BYTE type)
```

Remarks

IwaveCodecAccess selects either programmed I/O or DMA to service both the codec record FIFO and codec playback FIFO.

Set *type* to one of the following symbolic constants:

- `DMA_ACCESS` for DMA access
- `DMA_ACCESS|DMA_SIMPLEX` for single-channel DMA access, where either record or playback operation is allowed, but not both.
- `PIO_ACCESS` for programmed I/O access.

Return Value

None.

See Also

IwavePlayAccess, **IwaveRecordAccess**

IwaveCodecCnt — iwcodec.c

Function

Loads the codec sample counters.

Syntax

```
void IwaveCodecCnt(BYTE index, WORD cnt)
```

Remarks

IwaveCodecCnt loads the codec sample counters with a value determined from *cnt* (the total number of bytes comprising the samples) and the format of the data. The codec sample counters are each made up of two 8-bit registers: the Upper Playback Count and Lower Playback Count registers (CUPCTI and CLPCTI) for playback and the Upper Record Count and Lower Record Count registers (CURCTI and CLRCTI) for record. **IwaveCodecCnt** writes the lower byte first because writing the upper byte causes the full 16-bit count to be loaded.

The value loaded into the counters depends on the format of the data being transferred through the codec, as determined from *iw.cdatap*.

index selects either the playback or record sample counter. Set this argument to one of the following symbolic constants:

- `PLAY` or `_CLPCTI` for the playback sample counter
- `REC` or `_CLRCTI` for the record sample counter

These constants are defined in the **iwdefs.h** include file.

Return Value

None.

IwaveCodecIrq — iwcodec.c

Function

Enables or disables the codec interrupts.

Syntax

```
void IwaveCodecIrq(BYTE mode)
```

Remarks

IwaveCodecIrq enables or disables the codec interrupts. To enable the interrupts, it sets the *Global Interrupt Enable* bit of the External Control register (CEXTI[1]) High, thus causing all codec interrupt sources (CSR3I[6:4]) to pass onto the IRQ pin. To disable the interrupts, it sets CEXTI[2] Low. **IwaveCodecIrq** clears any codec interrupt requests presently pending before enabling or disabling interrupts.

To enable codec interrupts, set *mode* to `CODEC_IRQ_ENABLE`. To disable the interrupts, set *mode* to `~CODEC_IRQ_ENABLE`.

Return Value

None.

IwaveCodecMode — iwcodec.c

Function

Specifies the codec mode of operation.

Syntax

```
void IwaveCodecMode(BYTE mode)
```

Remarks

IwaveCodecMode specifies the codec mode of operation. The InterWave codec defaults to mode 1 after reset. Modes 1 and 2 provide compatibility with the CS4231. Mode 3 provides enhanced features over the CS4231.

Set *mode* to one of the following symbolic constants:

- `CODEC_MODE1`

- CODEC_MODE2
- CODEC_MODE3

Return Value

None.

IwaveCodecStatus — iwcodec.c

Function

Retrieves the contents of any of the codec status registers: CSR1R, CSR2I, or CSR3I.

Syntax

```
void IwaveCodecStatus(BYTE index)
```

Remarks

IwaveCodecStatus retrieves the contents the codec status register specified in *index*.

Note: Reading CSR1R causes bits CSR3I[3:0] and CSR2I[7:6] to be cleared, if any are set. If the application needs these bits, make sure they have been retrieved before reading CSR1R.

Set *index* to one of the following symbolic constants:

- CODEC_STATUS1
- CODEC_STATUS2
- CODEC_STATUS3

Return Value

None.

IwaveCodecTrigger — iwcodec.c

Function

Enables the codec record path, the codec playback path, or both.

Syntax

```
void IwaveCodecTrigger(BYTE path)
```

Remarks

IwaveCodecTrigger enables the codec record or codec playback *path* or both by setting one or both of the *Record Enable* and *Playback Enable* bits of the Configuration Register 1 (CFIG1I[1:0]), thereby causing the flow of data through the DACs or ADCs. These bits are not write-protected.

Set *path* to one of the following symbolic constants: RECORD or PLAYBACK. To enable both paths, OR both constants together (RECORD | PLAYBACK).

Return Value

None.

IwaveDacAtten — iwcodec.c

Function

Specifies the attenuation level for the left or right DAC input.

Syntax

```
void IwaveDacAtten(BYTE level, BYTE index)
```

Remarks

IwaveDacAtten specifies the attenuation for the left or right DAC input by writing *level* to the *DAC Attenuation Select* field of the Left Playback DAC Control or Right Playback DAC Control registers (CLDACI[5:0] or CRDACI[5:0]), as specified by *index*.

The value of *level* can range from 0–63 representing attenuation from 0 dB to –94.5 dB in steps of 1.5 dB.

Set *index* to one of the following symbolic constants: `LEFT_DAC` or `RIGHT_DAC`.

Return Value

None.

IwaveDataFormat — iwcodec.c

Function

Specifies the playback or record data formats.

Syntax

```
void IwaveDataFormat(BYTE data, BYTE index)
```

Remarks

IwaveDataFormat specifies the playback or record data formats in the Playback Data Format register (CPDFI) or the Record Data Format register (CRDFI). It writes *data*, which contains the data format and whether the data is stereo, to CPDFI[7:4] or CRDFI[7:4], depending on the value of *index*.

Set *data* to one of the following symbolic constants:

`BIT8_ULAW` μ-Law

`BIT8_ALAW` A-Law

`BIT8_LINEAR`
8-bit unsigned

`BIT16_BIG` 16-bit signed big endian

`BIT16_LITTLE`
16-bit signed little endian

`IMA_ADPCM` IMA compliant ADPCM

To select stereo data OR in the mnemonic `STEREO`. The default is mono data.

Set *reg* to one of the following symbolic constants:

`_CPDFI` playback

`_CRDFI` record

Note: Remember that in mode 1, both the playback and record data formats are specified in CPDFI. In modes 2 and 3, the playback data format is specified in CPDFI and the record data format is specified in CRDFI.

Example

To set the record data format to 8-bit µ-law stereo:

```
IwaveDataFormat(BIT8_ULAW | STEREO, _CRDFI)
```

Return Value

None.

IwaveDisableLineIn — iwcodec.c

Function

Disables the stereo line inputs.

Syntax

```
void IwaveDisableLineIn(void)
```

Remarks

IwaveDisableLineIn disables the stereo line inputs, both left and right channels, by setting the *Enable Line In* bit of the Mix Control register (UMCR[0]) High.

Return Value

None.

See Also

IwaveEnableLineIn

IwaveDisableMicIn — iwcodec.c

Function

Disables the stereo microphone inputs.

Syntax

```
void IwaveDisableMicIn(void)
```

Remarks

IwaveDisableMicIn disables the stereo microphone inputs, both left and right channels, by setting the *Enable Stereo Microphone Input* bit of the Mix Control register (UMCR[2]) Low.

Return Value

None.

See Also

IwaveEnableMicIn

IwaveDisableOutput — iwcodec.c

Function

Disables the stereo line outputs.

Syntax

```
void IwaveDisableOutput(void)
```

Remarks

IwaveDisableOutput disables the stereo line outputs, both left and right channels, by setting the *Enable Line Out* bit of the Mix Control register (UMCR[1]) High.

Return Value

None.

See Also

IwaveEnableOutput

IwaveEnableLineIn — iwcodec.c

Function

Enables the stereo line inputs.

Syntax

```
void IwaveDisableLineIn(void)
```

Remarks

IwaveEnableLineIn enables the stereo line inputs, both left and right channels, by setting the *Enable Line In* bit of the Mix Control register (UMCR[0]) Low.

Return Value

None.

See Also

IwaveDisableLineIn

IwaveEnableMicIn — iwcodec.c

Function

Enables the stereo microphone inputs.

Syntax

```
void IwaveEnableMicIn(void)
```

Remarks

IwaveEnableMicIn enables the stereo microphone inputs, both left and right channels, by setting the *Enable Stereo Microphone Input* bit of the Mix Control register (UMCR[2]) High.

Return Value

None.

See Also

IwaveDisableMicIn

IwaveEnableOutput — iwcodec.c

Function

Enables the stereo line outputs.

Syntax

```
void IwaveEnableOutput(void)
```

Remarks

IwaveEnableOutput enables the stereo line outputs, both left and right channels, by setting the *Enable Line Out* bit of the Mix Control register (UMCR[1]) Low.

Return Value

None.

See Also

IwaveDisableOutput

IwaveInputGain — iwcodec.c

Function

Specifies the gain level for the left or right ADC source.

Syntax

```
void IwaveInputGain(BYTE index, BYTE gain)
```

Remarks

IwaveInputGain specifies the gain level for the left or right ADC input source by writing *gain* to the *ADC Input Gain Select* field of the Left ADC Input Control or Right ADC Input Control register (CLICI[3:0] or CRICI[3:0]), as specified in *index*.

Set *index* to either `LEFT_SOURCE` (or `_CLICI`) or `RIGHT_SOURCE` (or `_CRICI`).

The value of *level* can range from 0–15 representing 0 dB to 22.5 dB in steps of 1.5 dB.

Return Value

None.

IwaveInputSource — iwcodec.c

Function

Selects the input source for the left or right ADC.

Syntax

```
void IwaveInputSource(BYTE index, BYTE source)
```

Remarks

IwaveInputSource selects one of several possible sources to the left or right ADC by writing *source* to the *ADC Source Select* field of the Left ADC Input Control or Right ADC Input Control registers (CLICI[7:6] or CRICI[7:6]), as specified by *index*.

Set *index* to one of the following symbolic constants: `LEFT_SOURCE` or `RIGHT_SOURCE`.

Set *source* to one of the following symbolic constants:

- `LINE_IN`
- `AUX1_IN`
- `MIC_IN`
- `MIX_IN`

Return Value

None.

IwaveLineLevel — iwcodec.c

Function

Specifies the level of attenuation or gain for inputs or outputs.

Syntax

```
void IwaveLineLevel(BYTE level, BYTE index)
```

Remarks

IwaveLineLevel specifies the attenuation or gain for the following InterWave inputs or outputs by writing *level* to a register indexed by *index*:

- the auxiliary 1/synthesizer line input (left or right)
- the auxiliary2 line input (left or right)
- the left or right line output
- the left or right line input

level specifies either an attenuation level or a gain level depending on its value. The value of *level* can range from 0–31 and represents the values +12 dB (0) to –34.5 dB (31) in steps of 1.5dB.

Set *index* to one of the following symbolic constants:

- `_CLAX1I` or `LEFT_AUX1_INPUT` (all modes)

- _CRAX1I or RIGHT_AUX1_INPUT (all modes)
- _CLAX2I or LEFT_AUX2_INPUT (all modes)
- _CRAX2I or RIGHT_AUX2_INPUT (all modes)
- _CLOAI or LEFT_LINE_OUT (mode 3 only)
- _CROAI or RIGHT_LINE_OUT (mode 3 only)
- _CLLICI or LEFT_LINE_IN (modes 2 and 3)
- _CRLICI or RIGHT_LINE_IN (modes 2 and 3)
- _CLMICI or LEFT_MIC_IN (mode 3 only)
- _CRMICI or RIGHT_MIC_IN (mode 3 only)

Return Value

None.

IwaveLineMute — iwcodec.c

Function

Mutes or unmutes the line inputs or outputs.

Syntax

```
void IwaveLineMute(BYTE state, BYTE index)
```

Remarks

IwaveLineMute mutes or unmutes one of the following inputs or outputs by writing the *state* (ON for muting or OFF for unmuting) to a register indexed by *index*:

- the auxiliary 1/synthesizer line input (left or right)
- the auxiliary 2 line input (left or right)
- the left or right line output
- the left or right line input

Set *index* to one of the following symbolic constants:

- _CLAX1I or LEFT_AUX1_INPUT (all modes)
- _CRAX1I or RIGHT_AUX1_INPUT (all modes)
- _CLAX2I or LEFT_AUX2_INPUT (all modes)
- _CRAX2I or RIGHT_AUX2_INPUT (all modes)
- _CLOAI or LEFT_LINE_OUT (mode 3 only)
- _CROAI or RIGHT_LINE_OUT (mode 3 only)
- _CLLICI or LEFT_LINE_IN (modes 2 and 3)
- _CRLICI or RIGHT_LINE_IN (modes 2 and 3)
- _CLMICI or LEFT_MIC_IN (mode 3 only)
- _CRMICI or RIGHT_MIC_IN (mode 3 only)

Return Value

None.

IwaveMonoAtten — iwcodec.c

Function

Specifies the attenuation level for the mono input.

Syntax

```
void IwaveMonoAtten(BYTE level)
```

Remarks

IwaveMonoAtten specifies the attenuation level for the mono input by writing *level* to the *Mono Input Attenuation* field of the Mono Input and Output Control register (CMONOI[3:0]).

The value of *level* can range from 0–15 representing attenuation from 0 dB to 45 dB in steps of 3 dB.

Return Value

None.

IwaveMonoMute — iwcodec.c

Function

Mutes or unmutes the InterWave mono input or output.

Syntax

```
void IwaveMonoMute(BYTE state, BYTE io)
```

Remarks

IwaveMonoMute mutes or unmutes the mono input or output by writing the *state* (`ON` or `OFF`) to the *Mono Input Mute Enable* or *Mono Output Mute Enable* bits of the Mono Input and Output Control register (CMONOI[7:6]), as specified in *io* (`MONO_OUTPUT` or `MONO_INPUT`).

Return Value

None.

IwavePlayAccess — iwcodec.c

Function

Selects either programmed I/O or DMA to service the codec playback FIFO.

Syntax

```
void IwavePlayAccess(BYTE type)
```

Remarks

IwavePlayAccess selects either programmed I/O or DMA to service the codec playback FIFO.

Set *type* to one of the following symbolic constants:

- DMA_ACCESS for DMA access
- PIO_ACCESS for programmed I/O access.

Return Value

None.

See Also

IwaveCodecAccess, **IwaveRecordAccess**

IwavePlayData — iwcodec.c

Function

Sets up and starts a DMA transfer of data through the codec playback FIFO.

Syntax

```
FLAG IwavePlayData(WORD size, BYTE wait, BYTE type)
```

Remarks

IwavePlayData sets up and starts a DMA transfer of data through the codec playback path. It programs the sample counters and makes sure that DMA cycles are selected. It then calls the **IwaveCodecTrigger** function to start the transfer.

size specifies the size in bytes of the amount of data to transfer.

Set *wait* to TRUE to cause this function to return only after the transfer is completed. Set it to FALSE to return right away.

Set *type* to one of the following symbolic constants:

DMA_READ	Transfer a specific amount of data one time.
AUTO_READ	Repeat the transfer over and over. When the transfer completes, the controller resets its base address to the start of the DMA buffer and starts the transfer again.

Return Value

IwavePlayData returns DMA_BUSY if the DMA channel is in use or DMA_OK if the transfer was started or completed successfully.

See Also

IwaveRecordData, IwaveStopDma

IwaveRecordAccess — iwcodec.c

Function

Selects either programmed I/O or DMA to service the codec record FIFO.

Syntax

```
void IwaveRecordAccess(BYTE type)
```

Remarks

IwaveRecordAccess selects either programmed I/O or DMA to service the codec record FIFO.

Set *type* to one of the following symbolic constants:

- DMA_ACCESS for DMA access
- PIO_ACCESS for programmed I/O access.

Return Value

None.

See Also

IwaveCodecAccess, **IwavePlayAccess**

IwaveRecordData — iwcodec.c

Function

Sets up and starts a DMA transfer of data through the codec record FIFO.

Syntax

```
FLAG IwaveRecordData(WORD size, BYTE wait, BYTE type)
```

Remarks

IwaveRecordData sets up and starts a DMA transfer of data through the codec record path. It programs the sample counters and makes sure that DMA cycles are selected. It then calls the **IwaveCodecTrigger** function to start the transfer.

size specifies the size in bytes of the amount of data to transfer.

Set *wait* to TRUE to cause this function to return only after the transfer is completed. Set it to FALSE to return right away.

Set *type* to one of the following symbolic constants:

DMA_READ	Transfer a specific amount of data one time.
AUTO_READ	Repeat the transfer over and over. When the transfer completes, the controller resets its base address to the start of the DMA buffer and starts the transfer again.

Return Value

IwaveRecordData returns DMA_BUSY if the DMA channel is in use or DMA_OK if the transfer was started or completed successfully.

See Also

IwavePlayData, IwaveStopDma

IwaveSetFrequency — iwcodec.c

Function

Specifies the playback or record sample frequency.

Syntax

```
void IwaveSetFrequency(BYTE index, WORD freq)
```

Remarks

IwaveSetFrequency specifies the playback or record sample frequency by writing appropriate values to the *Clock Divider Select* and *Crystal Select* fields of the Playback Data Format or Record Data Format registers (CPDFI[3:0] or CRDFI[3:0]), as specified in *index*. **IwaveSetFrequency** uses the available frequency closest to the Hertz value in *freq*.

Set *index* to one of the following symbolic constants: _CPDFI or _CRDFI.

Note: In modes 1 and 2, CPDFI controls both the playback and record sampling rates. In mode 3, CPDFI controls the playback rate and CRDFI controls the record rate.

Return Value

None.

IwaveSetTimer — iwcodec.c

Function

Loads a value into the codec timer.

Syntax

```
void IwaveSetTimer(WORD cnt)
```

Remarks

IwaveSetTimer loads the codec timer counter by writing the upper byte of *cnt* into the Upper Timer register (CUTIMI) and then writing the lower byte into the Lower Timer register (CLTIMI). Writing to CLTIMI loads the internal counter. The codec timer has a 10-µs resolution. When the timer starts, the counter decrements to zero at which time an interrupt is generated (if enabled). The internal counter is reloaded on the next clock.

Note: Reading these registers returns the value initially loaded into the registers, not the current value of the counter.

Return Value

None.

See Also

IwaveTimerStart, **IwaveTimerStop**

IwaveStopDma — iwcodec.c

Function

Stops an active DMA transfer from the record FIFO or to the playback FIFO.

Syntax

```
void IwaveStopDma(BYTE path)
```

Remarks

IwaveStopDma stops an active DMA transfer from the record FIFO or to the playback FIFO by disabling both the *path* and the DMA channel.

Set *path* to either PLAYBACK or RECORD.

Return Value

None.

See Also

IwaveRecordData, **IwavePlayData**

IwaveTimerStart — iwcodec.c

Function

Starts the codec timer.

Syntax

```
void IwaveTimerStart(void)
```

Remarks

IwaveTimerStart starts the codec timer by setting the *Timer Enable* bit of the Configuration Register 2 (CFIG2I[6]) High.

Return Value

None.

See Also

IwaveSetTimerStop, **IwaveTimerStop**

IwaveTimerStop — iwcodec.c

Function

Stops the codec timer.

Syntax

```
void IwaveTimerStop(void)
```

Remarks

IwaveTimerStop stops the codec timer by setting the *Timer Enable* bit of the Configuration Register 2 (CFIG2I[6]) Low. Setting this bit Low prevents the codec counter from decrementing and being reloaded.

Return Value

None.

See Also

IwaveSetTimer, **IwaveTimerStart**

CHAPTER

22 SYNTHESIZER DDK FUNCTIONS

IwaveRampVolume — iwvoice.c

Function

Ramps the volume of a voice.

Syntax

```
void IwaveRampVolume(BYTE voice, WORD start, WORD end,
   BYTE rate, BYTE mode)
```

Remarks

IwaveRampVolume ramps the volume of a specified *voice*. The *voice* argument can range from 0–31 (0 is voice 1). Specify the *start* volume level, the *end* volume level, a *rate* and increment, and a volume looping *mode*.

The *start* and *end* values can range from 0-4095.

rate contains two pieces of information. Bits 5–0 specify the increment to be added to or subtracted from the volume level. Bits 7–6 control the rate at which this increment is added or subtracted. For more information, see "SVRI—Synthesizer Volume Rate" on page 14-10. To select a rate OR in one of the following symbolic constants:

`VOLUME_RATE0`
: Adds or subtracts the increment value to the volume level every frame.

`VOLUME_RATE1`
: Adds or subtracts one eighth of the increment value to the volume every frame.

`VOLUME_RATE2`
: Adds or subtracts one eighth of the increment value to the volume every eighth frame.

`VOLUME_RATE3`
: Adds or subtracts one eighth of the increment value to the volume every 64th frame.

The increment value can range from 0–63. For example, to increment the volume by four every eighth frame, set *rate* to `VOLUME_RATE2|32`. `VOLUME_RATE0` is the default rate.

To specify continuous playback, set *mode* to `VC_ROLLOVER` or 0. Specify decreasing volume ramping by ORing in `VC_DIRECT`, but be aware that the function automatically selects this mode if *start* is greater than *end*.

Return Value

None.

See Also

IwaveSetVolume, **IwaveReadVolume**

IwaveReadVoice — iwvoice.c

Function

Returns the real local memory address from which a playing voice is currently fetching data.

Syntax

```
ADDRESS IwaveReadVoice(BYTE voice)
```

Remarks

IwaveReadVoice returns the real local memory address from which a playing *voice* is currently fetching data. The *voice* argument can range from 0–31 (0 is voice 1).

Return Value

The real local memory address from which a playing *voice* is currently fetching data.

IwaveReadVolume — iwvoice.c

Function

Returns the current looping volume value for a voice.

Syntax

```
WORD IwaveReadVolume(BYTE voice)
```

Remarks

IwaveReadVolume returns the current volume level for the specified *voice* as reflected in the Synthesizer Volume Level register (SVLI[15:4]). The *voice* argument can range from 0–31 (0 is voice 1).

Return Value

IwaveReadVolume returns the volume level for the voice.

See Also

IwaveSetVolume

IwaveReadyVoice — iwvoice.c

Function

Prepares a selected voice for playback. It does *not* start the voice.

Syntax

```
BYTE IwaveReadyVoice(BYTE voice, ADDRESS begin, ADDRESS end,
   ADDRESS fetch, BYTE mode)
```

Remarks

IwaveReadyVoice prepares a selected *voice* for playback, but does *not* start the voice playing. The *voice* argument can range from 0–31 (0 is voice 1). Specify the width of the sample data in *mode*. If the boundary address specified in *begin* contains a larger value than the *end* address, wavetable addressing is set to decrement from the *begin* address towards the *end* address. Otherwise, the addressing increments from *begin* to *end*. The direction of addressing is reflected in the return value. Specify the address where the synthesizer starts fetching sample data in *fetch*. The *begin*, *end*, and *fetch* addresses must all lie within the same 4MB local memory bank; that is, bits 23–22 of these addresses must be the same. Otherwise, the results are unpredictable.

Set *mode* to `VC_DATA_WIDTH` if the sample data is 16-bit data. Otherwise, set it to 0 or `FALSE`.

Return Value

IwaveReadyVoice returns the value specified in *mode* if the synthesizer address is set to increment, or *mode*|`VC_DIRECT` if address decrementing has been turned on.

IwaveSetLoopMode — iwvoice.c

Function

Specifies the looping mode for both addressing and volume of a voice.

Syntax

```
void IwaveSetLoopMode(BYTE voice, BYTE amode, BYTE vmode)
```

Remarks

IwaveSetLoopMode specifies the looping mode for both address (*amode*) and volume (*vmode*) of the specified *voice* by writing to the Synthesizer Address Control and Synthesizer Volume Control registers (SACI and SVCI). The *voice* argument can range from 0–31 (0 is voice 1). The function also enables address and volume boundary interrupts. Any features not enabled by the call to this function are disabled.

Use *amode* to specify the following attributes of address looping:

Data Width To specify 16-bit data, OR in the symbolic constant `VC_DATA_WIDTH`. Otherwise, 8-bit data is specified.

Direction To specify decreasing wavetable addressing, OR in the constant `VC_DIRECT`.

Enable Looping
To enable address looping, OR in the symbolic constant `VC_LOOP_ENABLE`. Otherwise, looping is disabled.

Enable Bidirectional Looping
To enable bidirectional address looping, OR in the symbolic constant `VC_BI_LOOP`.

Enable Wavetable IRQ
To enable interrupts when an address boundary is crossed, OR in the symbolic constant `VC_IRQ_ENABLE`.

Use *vmode* to specify the following attributes of volume looping:

Enable Continuous Play
To specify 16-bit data, OR in the symbolic constant VC_ROLLOVER. Otherwise, 8-bit data is specified.

Direction To specify decreasing volume ramp, OR in the constant VC_DIRECT.

Enable Looping
To enable volume looping, OR in the symbolic constant VC_LOOP_ENABLE. Otherwise, looping is disabled.

Enable Bidirectional Looping
To enable bidirectional volume looping, OR in the symbolic constant VC_BI_LOOP.

Enable Volume IRQ
To enable interrupts when a volume boundary is crossed, OR in the symbolic constant VC_IRQ_ENABLE.

Return Value

None.

See Also

IwaveReadyVoice

IwaveSetVoiceEnd — iwvoice.c

Function

Specifies a new local memory end boundary address for a voice.

Syntax

```
void IwaveSetVoiceEnd(BYTE voice, ADDRESS local)
```

Remarks

IwaveSetVoiceEnd specifies a new real *local* memory end boundary address for a particular *voice*. The *voice* argument can range from 0–31 (0 is voice 1).

This function can be used in conjunction with the **IwaveSetLoopMode** function to play a sampled decay. Set the new end boundary with **IwaveSetVoiceEnd** and turn off looping with **IwaveSetLoopMode** so the synthesizer accesses the wavetable data after the end boundary.

Return Value

None.

IwaveSetVoicePlace — iwvoice.c

Function

Specifies the local memory address location from where a voice fetches data.

Syntax

```
void IwaveSetVoicePlace(BYTE voice, DWORD local)
```

Remarks

IwaveSetVoicePlace specifies the address *local* in local memory where a *voice* fetches data from to a new position. The *voice* argument can range from 0–31 (0 is voice 1). Every voice currently fetching data from memory is summed into the output even if the voice is not running. Pops in the audio may result if a voice's fetching position is set to a location with a significant value.

Return Value

None.

See Also

IwaveReadVoice

IwaveSetVolume — iwvoice.c

Function

Specifies the current looping volume value for a voice.

Syntax

```
void IwaveSetVolume(BYTE voice, WORD volume)
```

Remarks

IwaveSetVolume specifies the current *volume* level for the specified *voice* to be written to the Synthesizer Volume Level register (SVLI[15:4]).The *voice* argument can range from 0–31 (0 is voice 1).

Return Value

None.

See Also

IwaveReadVolume

IwaveStartVoice — iwvoice.c

Function

Starts the voice.

Syntax

```
void IwaveStartVoice(BYTE voice)
```

Remarks

IwaveStartVoice starts the specified *voice* playing by setting the *Stop 1* and *Stop 0* bits of the Synthesizer Voice Control register (SACI[1:0]) Low. The *voice* argument can range from 0–31 (0 is voice 1). **IwaveStartVoice** assumes that the characteristics of the voice have already been programmed.

Return Value

None.

IwaveStopVoice — iwvoice.c

Function

Stops the output of a voice.

Syntax

```
void IwaveStopVoice(BYTE voice)
```

Remarks

IwaveStopVoice stops the output of a particular *voice* by stopping any volume looping and then preventing the synthesizer from fetching any more samples from local memory. It also disables the voice's ability to generate a wavetable interrupt. The *voice* argument can range from 0–31 (0 is voice 1).

Example

To stop voice 23:

```
IwaveStopVoice(22)
```

Return Value

None.

See Also

IwaveStopVolume

IwaveStopVolume — iwvoice.c

Function

Stops the volume looping for a voice.

Syntax

```
void IwaveStopVolume(BYTE voice)
```

Remarks

IwaveStopVolume stops the volume looping component of a particular *voice* by setting the *Stop 1* and *Stop 0* bits of the Synthesizer Voice Control register (SVCI[1:0]) High. The *voice* argument can range from 0–31 (0 is voice 1).

Example

To stop the volume looping component of voice 23:

```
IwaveStopVolume(22)
```

Return Value

None.

See Also

IwaveStopVoice

IwaveSynthGlobal iwvoice.c

Function

Controls the global mode of operation of the synthesizer (affects all voices).

Syntax

```
void IwaveSynthGlobal(BYTE mode, BOOL state)
```

Remarks

IwaveSynthGlobal controls the global modes of operation of the synthesizer which affects all voices. *mode* specifies the mode or modes of operation and *state* specifies whether to turn the modes `ON` or `OFF`. You can enable or disable one or more modes at the same time, but you can not both enable and disable a particular mode at the same time. To turn on or off only Enhanced mode, consider using the **IwaveSynthMode** function instead.

Set *mode* to one of the following symbolic constants:

- `ENH_MODE`
- `ENABLE_LFOS`
- `NO_WAVE_TABLE`
- `ENH_MODE|ENABLE_LFOS`
- `ENH_MODE|ENABLE_LFOS|NO_WAVE_TABLE`

Examples

To turn on Enhanced mode:

```
IwaveSynthGlobal(ENH_MODE, ON)
```

To enable Enhanced mode and all LFOS:

```
IwaveSynthGlobal(ENH_MODE|ENABLE_LFOS, ON)
```

To disable Enhanced mode (enables GUS-Compatibility mode):

```
IwaveSynthGlobal(ENH_MODE, OFF)
```

Return Value

None.

See Also

IwaveSynthMode

IwaveSynthMode iwvoice.c

Function

Switches the InterWave IC between GUS-Compatibility mode and Enhanced mode.

Syntax

```
void IwaveSynthMode(BYTE mode)
```

Remarks

IwaveSynthMode switches the InterWave IC between GUS-Compatibility mode and Enhanced mode and sets the `smode` member of the `iw` variable. *mode* should be set to the symbolic constant `GUS_MODE` or `ENH_MODE`. If the application uses the DDK memory management function to manage local memory, switching modes invalidates the current memory-management structure. After switching modes, reinitialize DRAM by calling **IwaveMemInit**.

Return Value

None.

IwaveVoiceFreq — iwvoice.c

Function

Specifies the sampling frequency of a voice in GUS-Compatibility mode.

Syntax

```
void IwaveVoiceFreq(BYTE voice, DWORD freq)
```

Remarks

IwaveVoiceFreq specifies the sampling frequency of the specified *voice* by writing *freq* to the Synthesizer Frequency Control register (SFCI). The *voice* argument can range from 0–31 (0 is voice 1). SFCI has two fields: SFCI[15:10] is an integer part indicating how many times faster than it was recorded to play the sample data. SFCI[9:0] is a fractional part that allows more resolution for the frequency (sample interpolation). SFCI[0] is not available in GUS-Compatibility mode. Use this function in GUS-Compatibility-mode applications. Otherwise, use the **IwaveVoicePitch** function for less code.

Return Value

None.

See Also

IwaveVoicePitch

IwaveVoicePan — iwvoice.c

Function

Specifies a voice's position in the stereo field.

Syntax

```
void IwaveVoicePan(BYTE voice, WORD right, WORD left)
```

Remarks

IwaveVoicePan specifies the stereo position of a *voice* by writing to the Synthesizer Right Offset and Synthesizer Left Offset registers (SROI and SLOI). The *voice* argument can range from 0–31 (0 is voice 1).

When the *Offset Enable* bit of the Synthesizer Mode Select register (SMSI[5]) is Low, the GUS-compatible pan mode is in effect and SROI[11:8] controls both the left and right stereo

offsets. In this mode, *right* can range from 0h–1Fh, with 0 placing the voice full left and 1Fh placing it full right. *left* is ignored.

When SMSI[5] is High, the enhanced offset mode is in effect and SROI[15:4] and SLOI[15:4] control the right and left offsets respectively. In this mode, *right* and *left* can each range from 0h–0FFFh.

For more information, see “Stereo Positioning—Offset and Pan” on page 7-19.

Return Value

None.

IwaveVoicePitch — iwvoice.c

Function

Specifies the sampling frequency of a voice in Enhanced mode.

Syntax

```
void IwaveVoicePitch(BYTE voice, DWORD freq)
```

Remarks

IwaveVoicePitch specifies the sampling frequency of the specified *voice* by writing *freq* to the Synthesizer Frequency Control register (SFCI). The *voice* argument can range from 0–31 (0 is voice 1). SFCI has two fields: SFCI[15:10] is an integer part indicating how many times faster than it was recorded to play the sample data. SFCI[9:0] is a fractional part that allows more resolution for the frequency (sample interpolation). SFCI[0] is not available in GUS-Compatibility mode. Use this function in Enhanced-mode applications only. Otherwise, use the **IwaveVoiceFreq** function.

Return Value

None.

See Also

IwaveVoiceFreq.

CHAPTER

23 LOCAL MEMORY CONTROL DDK FUNCTIONS

IwaveDmaCtrl — iwmem.c

Function

Prepares the DMA controller on the PC for an impending DMA transfer.

Syntax

```
FLAG IwaveDmaCtrl(DMA *dma, WORD size)
```

Remarks

IwaveDmaCtrl readies the DMA controller on the PC for an impending DMA transfer of *size* bytes and initializes the DMA structure pointed to by *dma*, which must have been properly initialized by a call to the **IwaveRegisterDMA** function. It then calls the **IwaveDmaPgm** function to program the DMA controller. It does *not* start the transfer.

Return Value

IwaveDmaCtrl returns `DMA_OK` if it succeeds or `~DMA_OK` if it fails.

See Also

IwaveDmaPage, **IwaveDmaXfer**, **IwaveDmaNext**, **IwaveDmaIleaved**

IwaveDmaIleaved — iwmem.c

Function

Programs the InterWave IC for an interleaved DMA transfer and starts the transfer.

Syntax

```
FLAG IwaveDmaIleaved(DMA *dma, WORD ctrl, BYTE tracks, WORD size)
```

Remarks

IwaveDmaIleaved programs the InterWave IC for an interleaved DMA transfer of *size* bytes for a specified number of *tracks* and initializes the DMA structure pointed to by *dma*, which must have been properly initialized by a call to the **IwaveRegisterDMA** function. The *tracks* argument can range from 0–31. It writes the LMC DMA Interleaved Control register (LDICI) and the LMC DMA Interleaved Base Address register (LDIBI). The local memory base address must be aligned to a 256-byte boundary. This function starts the DMA transfer.

Set *ctrl* to one of the following symbolic constants:

- 0 for 8-bit data with no MSB inversion.
- `IDMA_INV` for 8-bit data with MSB inverted.
- `IDMA_WIDTH_16` for 16-bit data with no MSB inversion.

- IDMA_INV|IDMA_WIDTH_16 for 16-bit data with MSB inverted.

Return Value

IwaveDmalleaved returns DMA_OK if it succeeds or ~DMA_OK if it fails.

See Also

IwaveDmaPage, **IwaveDmaXfer**, **IwaveDmaNext**, **IwaveDmaCtrl**

IwaveDmaMalloc — iwmem.c

Function

Allocates memory that lies within a DMA page.

Syntax

```
void far *IwaveDmaMalloc(WORD buffsize)
```

Remarks

IwaveDmaMalloc allocates *buffsize* bytes of memory that lies within a DMA page in system memory. The *buffsize* argument can be up to 64 Kbytes (a DMA page). This utility is provided so that applications can perform DMA transfers with the DMA controller operating in auto-initialization mode. Use this function only if allocating space for this type of transfer; otherwise, use the regular routines that come with your compiler.

Return Value

IwaveDmaMalloc returns NULL if it fails to allocate the buffer. Otherwise, it returns a valid pointer.

See Also

IwaveAllocDOS, **IwaveFreeDOS**

IwaveDmaNext — iwmem.c

Function

If the data to be transferred crosses over one DMA page in system memory, the DMA handler calls this function to send the data in the second DMA page.

Syntax

```
void IwaveDmaNext(DMA *dma)
```

Remarks

If the data to be DMA transferred crosses over one DMA page in system memory, the DMA handler calls **IwaveDmaNext** to send the data in the second DMA page. The possibility exists that a DMA buffer may cross the boundary between two contiguous DMA pages. in such a case, the DMA transfer is automatically split into two transfers: one for the data in the first page and another for the data in the second page. The DMA structure pointed to by *dma* must have been properly initialized by a call to the **IwaveRegisterDMA** function.

Return Value

None.

IwaveDmaPage — iwmem.c

Function

Sets up the InterWave IC and then initiates the transfer of up to one DMA page (64 Kbytes) to or from local memory.

Syntax

```
FLAG IwaveDmaPage(DMA *dma, WORD size)
```

Remarks

IwaveDmaPage programs the InterWave IC and then initiates the transfer of up to one DMA page to or from local memory. The DMA structure pointed to by *dma* must have been properly initialized by a call to the **IwaveRegisterDMA** function.

Return Value

IwaveDmaPage returns `DMA_OK` if it succeeds or `~DMA_OK` if it fails.

See Also

IwaveDmalleaved, **IwaveDmaXfer**, **IwaveDmaNext**, **IwaveDmaCtrl**

IwaveDmaPgm — iwmem.c

Function

Programs the DMA controller for an impending DMA transfer.

Syntax

```
void IwaveDmaPgm(DMA *dma)
```

Remarks

IwaveDmaPgm programs the DMA controller with the appropriate settings for the DMA channel reflected in the variable pointed to by *dma*.

Return Value

None.

See Also

IwaveDmaCtrl

IwaveDmaWait — iwmem.c

Function

Blocks program activity until a specific DMA transfer is completed.

Syntax

```
void IwaveDmaWait(void)
```

Remarks

IwaveDmaWait blocks execution until a specific DMA transfer has completed. The function monitors the third bit in `iw.flags`, which is cleared by the interrupt handler when the DMA transfer completes.

Return Value

None.

IwaveDmaXfer — iwmem.c

Function

Programs the DMA controller and the InterWave IC for a DMA transfer to or from local memory, then starts the transfer.

Syntax

```
FLAG IwaveDmaXfer(DMA *dma, WORD size)
```

Remarks

IwaveDmaXfer is an upper level driver that programs the DMA controller and the InterWave IC for a DMA transfer of *size* bytes to or from local memory, then starts the transfer. The *dma* argument points to a DMA structure, which must have been properly initialized by a call to the **IwaveRegisterDMA** function.

Return Value

IwaveDmaXfer returns `DMA_OK` if it succeeds or `~DMA_OK` if it fails.

See Also

IwaveDmaPage, **IwaveDmaCtrl**, **IwaveDmaNext**, **IwaveDmalleaved**

IwaveGetDmaPos — iwmem.c

Function

Reads the count register of the DMA controller to determine the current position in a DMA transfer.

Syntax

```
WORD IwaveGetDmaPos(DMA *dma)
```

Remarks

IwaveGetDmaPos reads the current count register within the DMA controller to determine the current position in a DMA transfer. The DMA structure pointed to by *dma* must have been properly initialized by a call to the **IwaveRegisterDMA** function.

Return Value

IwaveGetDmaPos returns a 16-bit value representing the contents of the DMA controller's current count register (down counter).

See Also

GetSamplePosition

IwaveMaxAlloc — iwmem.c

Function

Returns the size in bytes of the largest block of memory that can be still be allocated from the local memory pool.

Syntax

```
DWORD IwaveMaxAlloc(void)
```

Remarks

IwaveMaxAlloc returns the size in bytes of the largest block of memory currently available from the local memory pool. If the InterWave IC is operating in GUS-compatibility mode, the block can not be greater than 256 Kbytes. Use this function to determine if there is an available block of memory large enough to honor an allocation request.

Return Value

IwaveMaxAlloc returns the size in bytes of the largest block of memory currently available from the local memory pool.

See Also

IwaveMemInit, **IwaveMemAvail**, **IwaveMemFree**, **IwaveMemAlloc**

IwaveMemAlloc — iwmem.c

Function

Allocates a block of memory from the local memory pool.

Syntax

```
ADDRESS IwaveMemAlloc(DWORD size)
```

Remarks

IwaveMemAlloc allocates a block of memory from the local memory pool to the requesting application. In GUS-compatibility mode, the function rounds the requested *size* in bytes up to the next 32-byte boundary and the size of the allocated block can not be greater than 256 Kbytes. In enhanced mode, the function rounds the requested *size* to an even byte and the size is limited only by the amount of local memory present.

Return Value

If the allocation is successful, **IwaveMemAlloc** returns the base address of the allocated local memory block. Otherwise, it returns `ALLOC_FAILURE`.

See Also

IwaveMemInit, **IwaveMemAvail**, **IwaveMemFree**, **IwaveMaxAlloc**.

IwaveMemAvail — iwmem.c

Function

Returns the total amount of memory available in the local memory pool.

Syntax

```
DWORD IwaveMemAvail(void)
```

Remarks

IwaveMemAvail returns the total amount of memory in bytes currently available in the local memory pool. This value represents the addition of all available memory chunks.

Return Value

The total amount of memory in bytes available from the local memory pool.

See Also

IwaveMemInit, **IwaveMaxAlloc**, **IwaveMemFree**, **IwaveMemAlloc**.

IwaveMemCfg — iwmem.c

Function

Determines the current DRAM configuration of the InterWave IC-based hardware.

Syntax

```
void IwaveMemCfg(void)
```

Remarks

IwaveMemCfg determines the current DRAM configuration of the InterWave IC-based hardware, then writes an appropriate value to the LMC Configuration register (LMCFI) and stores the total amount of DRAM in Kbytes into `iw.size_mem`. The function first places the IC in enhanced mode to allow full access to all DRAM locations. Then it selects full addressing span (LMCFI[3:0]=0Ch). Finally, it determines the amount of DRAM in each bank and, from those values, the actual DRAM configuration. If a configuration other than one shown in Table 15-2 on page 15-5 is implemented, this function selects full addressing span (LMCFI[3:0]=0Ch).

Return Value

None.

See Also

IwaveMemSize

IwaveMemFree — iwmem.c

Function

Releases or de-allocates previously allocated blocks of local memory.

Syntax

```
BOOL IwaveMemFree(DWORD size, ADDRESS blk_addr)
```

Remarks

IwaveMemFree releases or de-allocates a block of *size* bytes of local memory located at the address in *blk_addr*. The block must have been previously allocated with

IwaveMemAlloc. The function returns the block to the free-block chain. As a last step, the function merges together all adjacent free blocks into a single block.

Return Value

IwaveMemFree returns TRUE if successful or FALSE if it failed to release memory.

See Also

IwaveMemInit, **IwaveMemAvail**, **IwaveMemAlloc**, **IwaveMaxAlloc**

IwaveMemInit — iwmem.c

Function

Initializes local memory into a local memory pool for allocation and de-allocation.

Syntax

```
BOOL IwaveMemInit(void)
```

Remarks

IwaveMemInit initializes the local memory into a local memory pool for allocation and de-allocation. In GUS-compatibility mode, the greatest chunk of memory that can be allocated is 256 Kbytes. In enhanced mode, the maximum block size for allocation is the size of physical memory.

Return Value

IwaveMemInit returns TRUE if successful or FALSE if it fails to set up the free memory chain.

See Also

IwaveMemFree, **IwaveMemAvail**, **IwaveMemAlloc**, **IwaveMaxAlloc**

IwaveMemPeek — iwmem.c

Function

Reads a byte of data from local memory.

Syntax

```
BYTE IwaveMemPeek(ADDRESS addr)
```

Remarks

IwaveMemPeek reads a byte of data from the address specified in *addr* through the LMC Byte Data register (LMBDR).

Return Value

IwaveMemPeek returns a byte of data from local memory.

See Also

IwavePokeBlock, **IwavePeekBlock**, **IwaveMemPoke**, **IwaveMemPokeW**, **IwaveMemPeekW**

IwaveMemPeekW — iwmem.c

Function

Reads a word (16 bits) of data from local memory.

Syntax

```
WORD IwaveMemPeekW(ADDRESS addr)
```

Remarks

IwaveMemPeekW reads a word (16 bits) of data from the address specified in *addr* through the LMC Byte Data register (LMBDR).

Return Value

IwaveMemPeekW returns a word (16 bits) of data from local memory.

See Also

IwavePokeBlock, **IwavePeekBlock**, **IwaveMemPoke**, **IwaveMemPokeW**, **IwaveMemPeekW**

IwaveMemPoke — iwmem.c

Function

Writes a byte of data to local memory.

Syntax

```
void IwaveMemPoke(ADDRESS addr, BYTE value)
```

Remarks

IwaveMemPoke writes the byte *value* to local memory at the address specified in *addr* through the LMC Byte Data register (LMBDR).

Return Value

None.

See Also

IwaveMemPokeW, IwavePokeBlock, **IwavePeekBlock**, **IwaveMemPeek**, **IwaveMemPeekW**

IwaveMemPokeW — iwmem.c

Function

Writes a word (16 bits) of data to local memory.

Syntax

```
void IwaveMemPokeW(ADDRESS addr, WORD value)
```

Remarks

IwaveMemPokeW writes the 16-bit *value* to local memory at the address specified in *addr* through the LMC Byte Data register (LMSBAI).

Return Value

None.

See Also

IwaveMemPoke, IwavePokeBlock, **IwavePeekBlock**, **IwaveMemPeek**, **IwaveMemPeekW**

IwaveMemSize — iwmem.c

Function

Returns the number of Kbytes available as local memory attached to the InterWave IC.

Syntax

```
WORD IwaveMemSize(void)
```

Remarks

IwaveMemSize returns the number of Kbytes available as local memory attached to the InterWave IC. The value returned by this function reflects the effective or actual amount of DRAM that can be accessed based on the mode of operation of the InterWave IC, as determine by the *Enhanced Mode* bit of the Synthesizer Global Mode register (SGMI[0]).

Return Value

IwaveMemSize returns the number of Kbytes of DRAM attached to the InterWave IC.

See Also

IwaveMemCfg

IwavePeekBlock — iwmem.c

Function

Reads a block of data from local memory through the LMC Byte Data register (LMBDR).

Syntax

```
void IwavePeekBlock(BYTE far *block, DWORD len, ADDRESS addr)
```

Remarks

IwavePeekBlock reads *len* bytes of data from local memory starting at address *addr* into the buffer pointed to by *block* a byte at a time. It starts by enabling the auto-increment feature of the InterWave IC whereby each access of the LMC Byte Data register (LMBDR) causes the local memory address to increment by one.

Return Value

None.

See Also

IwavePokeBlock, **IwavePokeBlockW**, **IwavePeekBlockW**

IwavePeekBlockW — iwmem.c

Function

Reads a block of data from local memory through the LMC 16-Bit Access register (LMSBAI).

Syntax

```
void IwavePeekBlockW(WORD far *block, DWORD len, ADDRESS addr)
```

Remarks

IwavePeekBlockW reads *len* bytes of data from local memory starting at address *addr* into the buffer pointed to by *block* a word (16 bits) at a time. It starts by enabling the auto-increment feature of the InterWave IC whereby each access of the LMC 16-Bit Access register (LMSBAI) causes the local memory address to increment by two.

Return Value

None.

See Also

IwavePokeBlock, **IwavePokeBlockW**, **IwavePeekBlock**

IwavePokeBlock — iwmem.c

Function

Writes a block of data to local memory through the LMC Byte Data register (LMBDR).

Syntax

```
void IwavePokeBlock(BYTE far *block, DWORD len, ADDRESS addr)
```

Remarks

IwavePokeBlock writes *len* bytes of data from the buffer pointed to by *block* to local memory starting at address *addr* a byte at a time. It starts by enabling the auto-increment feature of the InterWave IC whereby each access of the LMC Byte Data register (LMBDR) causes the local memory address to increment by one.

Return Value

None.

See Also

IwavePeekBlock, **IwavePokeBlockW**, **IwavePeekBlockW**

IwavePokeBlockW — iwmem.c

Function

Writes a block of data to local memory through the LMC 16-Bit Access register (LMSBAI).

Syntax

```
void IwavePokeBlockW(WORD far *block, DWORD len, ADDRESS addr)
```

Remarks

IwavePokeBlockW writes *len* bytes of data from the buffer pointed to by *block* to local memory starting at address *addr* a word (16 bits) at a time. It starts by enabling the auto-increment feature of the InterWave IC whereby each access of the LMC 16-Bit Access register (LMSBAI) causes the local memory address to increment by two.

Return Value

None.

See Also

IwavePokeBlock, **IwavePeekBlock**, **IwavePeekBlockW**

APPENDIX

A PACKAGING AND PIN DESIGNATIONS

The InterWave audio IC is packaged in a 160-pin (Am78C201) or 120-pin (Am78C202) plastic quad flat pack (PQFP). This chapter lists the pins for both packages of the InterWave audio IC and then groups and describes the pins by function.

Am78C201 Pin Designations

Table A-1. Am78C201 Pin Designations

System Control		Codec		Local Memory		Ports, Misc.	
Pin Name	**# Pins**	**Pin Name**	**# Pins**	**Pin Name**	**# Pins**	**Pin Name**	**# Pins**
SD15–SD0	16	MIC[L,R]	2	MA10–MA0	11	XTAL1I	1
SA11–SA0*	12	AUX1[L,R]	2	MD7–MD0	8	XTAL1O	1
$\overline{\text{SBHE}}$	1	AUX2[L,R]	2	$\overline{\text{BKSEL3}}$–$\overline{\text{BKSEL0}}$	4	XTAL2I	1
DRQ[7,6,5,3,1,0]	6	LINEIN[L,R]	2	$\overline{\text{ROMCS}}$	1	XTAL2O	1
$\overline{\text{DAK}}$[7,6,5,3,1,0]	6	LINEOUT[L,R]	2	$\overline{\text{RAHLD}}$	1	MIDIRX	1
TC	1	MONOIN	1	RA21–RA20	2	MIDITX	1
IRQ[15,12,11]	3	MONOOUT	1	$\overline{\text{MWE}}$	1	GAMIN3–GAMIN 0*	4
IRQ[7,5,3,2/9]	4	IREF	1	$\overline{\text{RAS}}$	1	GAMIO3–GAMIO 0	4
$\overline{\text{IRQ4, IRQ10}}$	2→	GPOUT1–GPOUT0	*				
$\overline{\text{IOCHK}}$	1	CFILT	1				
$\overline{\text{IOR}}$	1	AREF	1				
$\overline{\text{IOW}}$	1		2				
$\overline{\text{IOCS16}}$	1						
IOCHRDY	1						
AEN	1						
EX_IRQ*	1→	ESPCLK	*				
EX_DRQ*	1→	ESPDIN	*				
$\overline{\text{EX\_DAK}}$*	1→	ESPSYNC	*				
$\overline{\text{EX\_CS}}$*	1→	ESPDOUT	*				
RESET	1						
$\overline{\text{SUSPEND}}$*	1→			$\overline{\text{FRSYNC}}$	*		
C32KHZ*	1→			$\overline{\text{EFFECT}}$	*		
PNPCS	1						
Power & Ground	37						

Pin Descriptions by Functional Group

Table A-2 through Table A-7 list pins by function and describe each pin.

System Bus Interface Pins

Table A-2. System Bus Interface Pins

Name	Qty.	Type	Description
AEN	1	input	**Address Enable** from the ISA bus, used to distinguish between DMA and I/O cycles. This signal must be driven low when the bus performs and I/O access to the IC.
$\overline{\text{DAK7}}$, $\overline{\text{DAK6}}$, $\overline{\text{DAK5}}$, $\overline{\text{DAK3}}$, $\overline{\text{DAK1}}$, $\overline{\text{DAK0}}$	6	input	The selectable **DMA Acknowledge** lines from the ISA bus. $\overline{\text{DAK 0}}$, $\overline{\text{DAK1}}$, and $\overline{\text{DAK3}}$ are used for 8-bit DMA transfers and DAK5, DAK6, and DAK7 are used for 16-bit DMA. The device can select up to three of the six supported DMA channels; the allocation of DMA channels is fully programmable using the Plug and Play registers.
DRQ7, DRQ6, DRQ5, DRQ3, DRQ1, DRQ0	6	3-state output	The selectable **DMA Request** lines to the ISA bus. DRQ0, DRQ1, and DRQ3 are used for 8-bit DMA transfers and DRQ5, DRQ6, and DRQ7 are used for 16-bit DMA. The device can select up to three of the six supported DMA channels; the allocation of DMA channels is fully programmable using the Plug and Play registers.
IOCHRDY	1	oc output	**I/O Channel Ready** to the ISA bus is used to extend the I/O bus cycle when deasserted. IOCHRDY high indicates that the device is ready to complete the current I/O bus cycle.
$\overline{\text{IOCS16}}$	1	oc output	**I/O Chip Select 16** is asserted low by the device during an I/O read or write operation to indicate that a 16-bit port is supported at the current address.
$\overline{\text{IOR}}$	1	input	**I/O Read** on the ISA bus is driven low by the host to indicate that an input/output read operation is taking place. $\overline{\text{IOR}}$ is valid only if the AEN signal is also low.
$\overline{\text{IOW}}$	1	input	**I/O Write** on the ISA bus is driven low by the host to indicate that an input/output write operation is taking place. $\overline{\text{IOW}}$ is valid only if the AEN signal is also low.
IRQ15, IRQ12, IRQ11, IRQ7, IRQ5, IRQ3, IRQ2	7	3-state output	The selectable **Interrupt Requests** to the ISA bus. IRQ4 and IRQ10 are multiplexed with GPOUT1 and GPOUT0 and are listed in Table A-5. The device can select up to three of the nine supported interrupts; the allocation of interrupt signals is fully programmable using the Plug and Play registers. Internally, interrupt sources can be assigned to the available interrupt request signals as required by software.
$\overline{\text{IOCHK}}$	1	oc output	**I/O Check**. Channel or I/O channel check on the ISA bus. $\overline{\text{IOCHK}}$ is asserted low by the device to generate an NMI (non-maskable interrupt).
PNPCS	1	bidir	**Plug and Play Serial EEPROM Chip Select**. Active high output used as chip select for the Plug and Play serial EEPROM. This is an input during reset; its state is latched by the trailing edge of reset to determine whether the IC is in PNP-compliant mode (low) or PNP-system mode (high).
RESET	1	input	**Reset** from the ISA bus. When RESET is asserted high on the ISA bus, the device performs an internal system reset. The RESET pin must be asserted for at least 10 ms before being deasserted. While in the reset state, the device ignores all ISA bus activity and no local memory cycles take place. On the trailing edge of RESET, the state of some I/O pins are latched to determine the configuration of certain multifunction pins.
$\overline{\text{SBHE}}$	1	input	The **System Byte High Enable** signal indicates the high byte of the system data bus is to be used. When connecting to an 8-bit ISA bus, this pin must be disconnected.
SD15–SD0	16	bidir	The ISA **System Data Bus** is used to transfer data to and from the device. The entire data bus, SD15–SD0, is active during 16-bit I/O access. During 8-bit I/O accesses, the lower data bus, SD7–SD0, is active when accessing an even byte, and the upper data bus, SD15–SD8, is active when accessing an odd byte.

Table A-2. System Bus Interface Pins (continued)

Name	Qty.	Type	Description
TC	1	input	**Transfer Complete** or **Terminal Count** is driven active high by the master or slave DMAC when the word or byte transfer count for a DMA channel is complete.
XTAL1I	1	input	**Crystal 1 Input**. Input from the 24.576-MHz crystal. The clock used by the codec module to support select sampling rates is derived from the 24.576-MHz crystal attached to the XTAL1 pins. An external 24.576-MHz CMOS compatible clock is not supported.
XTAL1O	1	output	**Crystal 1 Output**. Output from the 24.576-MHz crystal.
XTAL2I	1	input	**Crystal 2 Input**. Input from the 16.9344-MHz crystal. The main clocks used throughout the IC are derived from the 16.9344-MHz crystal attached to the XTAL2 pins. An external 16.9344-MHz CMOS-compatible clock is not supported.
XTAL2O	1	output	**Crystal 2 Output**. Output from the 16.9344-MHz crystal.

Note:

oc = open collector or spen drain.

Codec/Mixer Pins

Table A-3. Codec/Mixer Pins

Name	Qty	Type	Description
AREF	1	analog	The **Analog Reference** pin provides a reference voltage which can be used by external amplifier circuitry. When V_{CC} is at +5 V, the value of this output pin is 0.376 times V_{CC}, nominal. When V_{CC} is at +3.3 V, the value of AREF is 0.303 times V_{CC}, nominal. AREF is capable of sinking up to 250 microamps or sourcing up to 3.0 milliamps without degradation and can be placed into high-impedance mode, as controlled by the Mono Input and Output Control (CMONOI) register.
AUX1L, AUX1R	2	analog	The **Stereo Auxiliary 1 Inputs** provide an alternative input path and are multiplexed with the synthesizer DAC outputs. Only one of these sources can be mixed and supplied to LINEOUT as selected in Configuration Register 3 (CFIG3I). Either of these sources can be selected for analog-to-digital conversion through the Record Multiplexer. The AUX1 input impedance is at least 20 kΩ.
AUX2L, AUX2R	2	analog	The **Stereo Auxiliary 2 Inputs** can always be independently mixed or muted. Typically, these inputs are used for mixing analog CD stereo audio. The AUX2 input impedance is at least 20 kΩ.
CFILT	1	analog	The **Capacitor Filter** input must be connected to analog ground through a 0.1µF capacitor and a 10 µF capacitor.
GPOUT1, GPOUT0	2	output	The **General Purpose Digital Outputs** are two general purpose digital outputs controlled by bits located in the External Control (CEXTI) register. These pins are multiplexed with IRQ4 and IRQ10. GPOUT1 and GPOUT0 are selected by IEIRQI[7] = 1.
IREF	1	analog	The Current Reference input pin must be connected to analog ground through a 61.9 kΩ 1% tolerance resistor.
LINEINL, LINEINR	2	analog	The **Stereo Line Inputs** can always be independently mixed or muted. These inputs can also be selected for analog-to-digital conversion through the Record Multiplexer. Typically, these inputs are used for mixing or recording analog stereo audio from a variety of external stereo audio sources. The LINEIN input impedance is at least 20 kΩ.
LINEOUTL, LINEOUTR	2	analog	The **Stereo Line Outputs** are stereo single-ended 600 Ω line drivers which are the sum of the left and right mixer channels. The LINEOUTs can be independently attenuated or muted and these mixer outputs can also be selected for analog-to-digital conversion through the Record Multiplexer. Typically, these outputs are used for driving powered speakers or connected to speaker or headphone drivers.

Table A-3. Codec/Mixer Pins (continued)

Name	Qty	Type	Description
MICL, MICR	2	analog	The **Stereo Microphone Inputs** can be independently mixed or muted. These inputs can also be selected for analog-to-digital conversion through the Record Multiplexer. Typically, these inputs are used for mixing or recording a preamplified signal from a stereo microphone. The MIC input impedance is at least 20 kΩ.
MONOIN	1	analog	The **Mono Input** can always be independently mixed or muted and feeds both the left and right mixer output paths. Typically, this input is used for mixing PC speaker audio. The MONOIN input impedance is at least 20 kΩ.
MONOOUT	1	analog	The **Mono Output** is a single-ended 600 Ω line driver which provides the sum of the left and right LINEOUT signals and is independently mutable. Typically, this output is connected to a speaker driver for a PC speaker.

Local Memory Controller Pins

Table A-4. Local Memory Controller Pins

Name	Qty	Type	Description
$\overline{\text{BKSEL3}}$–$\overline{\text{BKSEL0}}$	4	output	The **Bank Select** signals are used to control the $\overline{\text{CAS}}$ input of each DRAM bank or the Output Enable input of each ROM bank.
MA10–MA3	8	bidir	The **Memory Address** signals are multiplexed row-column address lines for DRAM cycles. For ROM access cycles, they are time multiplexed ROM Latched Address[10:3] outputs and ROM High Byte Data Bus[15:8] inputs. The ROM Latched Addresses must be latched externally using the $\overline{\text{RAHLD}}$ signal.
MA2–MA0	3	output	**Memory address**. The multiplexed row-column address bits for DRAM cycles, and the RLA[2,1,19] outputs for ROM access cycles.
MD7–MD0	8	bidir	The **Memory Data Bus** for DRAM cycles. For ROM access cycles, they are time multiplexed ROM Latched Address[18:11] outputs and ROM Low Byte Data Bus[7:0]. The ROM Latched Addresses must be latched externally using the $\overline{\text{RAHLD}}$ signal. For Plug and Play Serial EEPROM accesses, MD[2] is the Serial Data Clock(SK), MD[1] is the Serial Data Input(DI), and MD[0] is the Serial Data Output(DO).
$\overline{\text{MWE}}$	1	output	The **Memory Write Enable** output controls the $\overline{\text{WE}}$ pin of all the DRAM banks and determines whether a DRAM cycle is a read or write. This pin remains high during all refresh cycles.
RA21–RA20	2	bidir	The high **ROM Address** lines during ROM accesses. At the trailing edge of RESET, these signals become inputs that are used to determine the operation mode of certain multiplexed function pins.
$\overline{\text{RAHLD}}$	1	output	The **ROM Address Hold** output is used to latch the state of ROM Latched Address lines, MD[7:0] (RLA[18:11]) and MA[10:3] (RLA[10:3]), in external latches during ROM accesses.
$\overline{\text{RAS}}$	1	output	The **Row Address Strobe** is asserted low during DRAM accesses and is connected directly to the $\overline{\text{RAS}}$ input of each DRAM in all of the DRAM banks.
$\overline{\text{ROMCS}}$	1	output	The **ROM Chip Select** output is asserted LOW during ROM accesses and is connected directly to the Chip Select/Enable input of each ROM in all of the ROM banks.

Multiplexed Function Pins

Table A-5. Multiplexed Function Pins

Name	Qty	Type	Description
C32KHZ / EFFECT	1	input / output	The **Suspend Mode Refresh Clock** input (C32KHZ) is used for refreshing local memory DRAM when the InterWave IC is in suspend mode. This pin can also be the **Synthesizer Effect Local Memory Writes** output (EFFECT) which is asserted low during writes to local memory DRAM involving synthesizer delay-based effects. The operation mode of this pin is determined by the state of RA[21] at the trailing edge of RESET. If RA[21] is high at the trailing edge of RESET, the C32KHZ pin function is selected; if RA[21] is low, the EFFECT signal is selected.
EX_CS / ESPOUT	1	output	The **External Device Chip Select** consisting of the decode of AEN deasserted (low) and the address specified in the PNP CD-ROM Address High/Low (PRAHI and PRALI) registers. Optionally, this pin can be configured as the **External Serial Port Data Out** signal (ESPDOUT) through the Compatibility (ICMPTI) register.
EX_DAK / ESPSYNC	1	output	The **External Device DMA Acknowledge** output to the external device. Optionally, this pin can be configured as the **External Serial Port Sync** signal (ESPSYNC) through the Compatibility (ICMPTI) register.
EX_DRQ / ESPDIN	1	input	**External Device DMA Request**. Optionally, this pin can be configured as the **External Serial Port Data In** signal (ESPDIN) through the Compatibility (ICMPTI) register.
EX_IRQ / ESPCLK	1	input	**External Device Interrupt Request**. Optionally, this pin can be configured as the 2.1168-MHz **External Serial Port Clock** output signal (ESPCLK) through the Compatibility (ICMPTI) register.
SA11–SA0 / SCS1–SCS0, SA3–SA0	12	input	**System Address Bus**. During internal decoding mode, these inputs are the 12 lower lines of the ISA System Address Bus which are used along with AEN to generate decodes for internal device resources. During external decoding mode, the System Address Bus is redefined as follows: SA[11:6] are not used, SA[5:4] are redefined as **System Chip Selects** (SCS[1:0]), and the lower address lines SA[3:0] are unchanged. The decoding mode is determined by the state of RA[20] at the trailing edge of the RESET signal. If RA[20] is low at the trailing edge of RESET, internal decoding mode is selected; if RA[20] is high, external decoding mode is selected.
SUSPEND / FRSYNC	1	input / output	When the **Suspend** input is asserted low, all chip activity becomes frozen, the oscillators are turned off, the C32KHZ input clock is used to refresh local memory DRAM, and most of the ISA bus inputs and outputs are isolated from the IC. This pin can also be used as the **Frame Sync** output which is asserted low at the start of each synthesizer data frame. The operation mode of this pin is determined by the state of RA[21] at the trailing edge of the RESET signal. If RA[21] is high at the trailing edge of RESET, the SUSPEND input function is selected; if RA[21] is low, the FRSYNC output signal is selected.
IRQ4 / GPOUT0	1	input / output	Register bit IEIRQI[7] selects between the two additional **Interrupt Request** lines (IRQ4 and IRQ10) and the **General Purpose Digital Output** lines (GPOUT0 and GPOUT1). If IEIRQI[7] is Low (the default), the pins are IRQ lines; if High, they become GPOUT lines. For description of the **Interrupt Request** lines, see Table A-2. The **General Purpose Digital Outputs** are two general purpose digital outputs controlled by bits located in the External Control (CEXTI) register.
IRQ10 / GPOUT1	1	input / output	See description for IRQ4 / GPOUT0.

Game Port and MIDI Port Pins

Table A-6. Game and MIDI Port Pins

Name	Qty	Type	Description
GAMIN3–GAMIN 0	4	input	The **Game Inputs** are used to monitor the state of buttons on external joystick(s). The state of these inputs can be read from the Game Control (GGCR) register. These pins are internally pulled up through a nominal 6kΩ resistance.
GAMIO3–GAMIO 0	4	analog	The **Game I/O** pins are used to determine the state of potentiometers on an external joystick to obtain the joystick's X-Y position.
MIDITX	1	output	The **MIDI Transmit** output is used to send serial digital data from the internal Motorola MC6850-compatible UART.
MIDIRX	1	input	The **MIDI Receive** input is used to receive serial digital data into the internal Motorola MC6850-compatible UART.

Power Supply Pins

Table A-7. Power Supply Pins

Name	Qty	Type	Description
AVcc	6	power	**Analog Power**. Supplies power to analog portions of the InterWave IC.
DVcc	8	power	**Digital Power**. Supplies power to digital portions of the InterWave IC.
AVss	8	power	**Analog Ground**. Supplies ground reference to analog portions of the InterWave IC.
DVss	15	power	**Digital Ground**. Supplies ground reference to digital portions of the InterWave IC.

APPENDIX

B SAMPLE PLUG AND PLAY RESOURCE MAP

Sample B-1 contains a sample resource map for an InterWave IC-based board with an EISA-assigned vendor ID of ADV, a vendor assigned product number of 550Ah, and a serial number of 000000001h. Change these items and others to reflect values assigned by your company for your particular board. Note that the checksum in the 9th byte is calculated over the first 8 bytes in a manner prescribed by the Plug and Play ISA specification 1.0A.

Sample B-1. Sample PNP Resource Map

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Plug and Play Header
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x04         ; Vendor ID Byte 0
DB 0x96         ; Vendor ID Byte 1
DB 0x55         ; Vendor assigned product number (byte 0)
DB 0x0A         ; Vendor assigned product number (byte 1)
DB 0x01         ; Serial Number Byte 0
DB 0x00         ; Serial Number Byte 1
DB 0x00         ; Serial Number Byte 2
DB 0x00         ; Serial Number Byte 3
DB 0x3d         ; checksum calculated on above bits
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Plug and Play Version
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
0x0A            ; small item, plug and play version tag
0x10            ; packed BCD, version 1.0
0x10            ; vendor specified product number (1.0)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Identifier String (ANSI)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x82         ; large item, identifier string tag
DB 0x0F         ; length byte 0  (15)
DB 0x00         ; length byte 1
DB 0x49         ; I    (identifier string is InterWave Audio)
DB 0x6E         ; n
DB 0x74         ; t
DB 0x65         ; e
DB 0x72         ; r
DB 0x57         ; W
DB 0x61         ; a
DB 0x76         ; v
DB 0x65         ; e
DB 0x20
DB 0x41         ; A
DB 0x75         ; u
DB 0x64         ; d
DB 0x69         ; i
DB 0x6F         ; o
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Audio Logical Dev ID
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x15        ; small item, logical dev tag
DB 0x04        ; Vendor ID Byte 0
DB 0x96        ; Vendor ID Byte 1
DB 0x00        ; Vendor assigned function ID (Byte 0)
DB 0x00        ; Vendor assigned function ID (Byte 1)
DB 0x02        ; flags[1] - supports commands at index 0x31
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; IRQ Descriptors
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x22        ; small item, irq1 tag (Synth, codec, compatibility)
DB 0xAC        ;  IRQ2/9, IRQ3, IRQ5 and IRQ7 supported
DB 0x98        ;  IRQ15, IRQ12 and IRQ11 supported
;
DB 0x22        ; small item, irq2 tag (MIDI IRQ)
DB 0xAC        ;  IRQ2/9, IRQ3, IRQ5 and IRQ7 supported
DB 0x98        ;  IRQ15, IRQ12 and IRQ11 supported
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DMA  Descriptors
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x2A        ; small item, dma1 tag
DB 0xEB        ; DRQ0, DRQ1,DRQ3,DRQ5,DRQ6, DRQ7 supported
DB 0x01        ; 8 and 16-bit DMA transfer type.
;
DB 0x2A        ; small item, dma2 tag
DB 0xEB        ; DRQ0, DRQ1, DRQ3, DRQ5, DRQ6, DRQ7 supported
DB 0x01        ; 8 and 16-bit DMA transfer type.
;
DB 0x30        ; small item, start dependent function 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; I/O Descriptors
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x47        ; small item, I/O port tag
DB 0x00        ; decodes 10-bit ISA bus only
DB 0x20        ; min base addr 0x220 (P2XR)
DB 0x02        ;
DB 0x20        ; max base addr 0x220 (P2XR)
DB 0x02        ;
DB 0x10        ; align to 16-byte boundary
DB 0x10        ; 16 ports
;
DB 0x47        ; small item, I/O port tag
DB 0x00        ; decodes 10-bit ISA bus only
DB 0x20        ; min base addr 0x320 (P3XR)
DB 0x03        ;
DB 0x20        ; max base addr 0x320 (P3XR)
DB 0x03        ;
DB 0x08        ; align to 8-byte boundary
DB 0x08        ; 8 ports
;
DB 0x30        ; small item, start dependent function 1
;
DB 0x47        ; small item, I/O  port tag
DB 0x00        ; decodes 10-bit ISA bus only
DB 0x30        ; min base addr 0x230 (P2XR)
DB 0x02        ;
```

```
DB 0x30         ; max base addr 0x230 (P2XR)
DB 0x02         ;
DB 0x10         ; align to 16-byte boundary
DB 0x10         ; 16 ports
;
DB 0x47         ; small item, I/O port tag
DB 0x00         ; decodes 10-bit ISA bus only
DB 0x30         ; min base addr 0x330 (P3XR)
DB 0x03         ;
DB 0x30         ; max base addr 0x330 (P3XR)
DB 0x03         ;
DB 0x08         ; align to 8-byte boundary
DB 0x08         ; 8 ports
;
DB 0x30         ; small item, start dependent function 2
;
DB 0x47         ; small item, I/O  port tag
DB 0x00         ; decodes 10-bit ISA bus only
DB 0x40         ; min base addr 0x240 (P2XR)
DB 0x02         ;
DB 0x40         ; max base addr 0x240 (P2XR)
DB 0x02         ;
DB 0x10         ; align to 16-byte boundary
DB 0x10         ; 16 ports
;
DB 0x47         ; small item, I/O port tag
DB 0x00         ; decodes 10-bit ISA bus only
DB 0x40         ; min base addr 0x340 (P3XR)
DB 0x03         ;
DB 0x40         ; max base addr 0x340 (P3XR)
DB 0x03         ;
DB 0x08         ; align to 8-byte boundary
DB 0x08         ; 8 ports
;
DB 0x30         ; small item, start dependent function 3
;
DB 0x47         ; small item, I/O  port tag
DB 0x00         ; decodes 10-bit ISA bus only
DB 0x50         ; min base addr 0x250 (P2XR)
DB 0x02         ;
DB 0x50         ; max base addr 0x250 (P2XR)
DB 0x02         ;
DB 0x10         ; align to 16-byte boundary
DB 0x10         ; 16 ports
;
DB 0x47         ; small item, I/O port tag
DB 0x00         ; decodes 10-bit ISA bus only
DB 0x50         ; min base addr 0x350 (P3XR)
DB 0x03         ;
DB 0x50         ; max base addr 0x350 (P3XR)
DB 0x03         ;
DB 0x08         ; align to 8-byte boundary
DB 0x08         ; 8 ports
;
DB 0x30         ; small item, start dependent function 3
;
DB 0x47         ; small item, I/O  port tag
DB 0x00         ; decodes 10-bit ISA bus only
```

```
DB 0x60         ; min base addr 0x260 (P2XR)
DB 0x02         ;
DB 0x60         ; max base addr 0x260 (P2XR)
DB 0x02         ;
DB 0x10         ; align to 16-byte boundary
DB 0x10         ; 16 ports
;
DB 0x47         ; small item, I/O port tag
DB 0x00         ; decodes 10-bit ISA bus only
DB 0x60         ; min base addr 0x360 (P3XR)
DB 0x03         ;
DB 0x60         ; max base addr 0x360 (P3XR)
DB 0x03         ;
DB 0x08         ; align to 8-byte boundary
DB 0x08         ; 8 ports
;
DB 0x38         ; end dependent function tag
;
DB 0x47         ; I/O descriptor tag - codec's PCODAR
DB 0x00         ; board does not decode the full 16-bit ISA addr
DB 0x00         ; min base addr 0x200
DB 0x02
DB 0xFC         ; max base addr 0x3FC
DB 0x03
DB 0x04         ; alignment
DB 0x04         ; 4 contiguous  ports
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; External Logical Dev ID
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x15         ; small item, logical dev tag
DB 0x04         ; Vendor ID Byte 0
DB 0x96         ; Vendor ID Byte 1
DB 0x00         ; Vendor assigned function ID (Byte 0)
DB 0x01         ; Vendor assigned function ID (Byte 1)
DB 0x02         ; flags[1] - supports commands at index 0x31
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DMA  Descriptor
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x2A         ; small item, dma tag
DB 0xEB         ; DRQ0, DRQ1,DRQ3,DRQ5,DRQ6, DRQ7 supported
DB 0x01         ; 8 and 16-bit DMA transfer type
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; IRQ Descriptor
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x22         ; small item,  irq tag
DB 0xAC         ;  IRQ2/9, IRQ3, IRQ5 and IRQ7 supported
DB 0x98         ;  IRQ15, IRQ12 and IRQ11 supported
;
DB 0x30         ; small item, start dependent function 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; I/O Descriptors
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x47         ; I/O descriptor tag - PCDRAR
DB 0x00         ; board does not decode the full 16-bit ISA addr
DB 0xF0         ; min base addr 0x1F0
DB 0x01
```

```
DB 0xF0         ; max base addr 0x1F0
DB 0x01
DB 0x08         ; alignment
DB 0x08         ; 8 contiguous  ports
;
DB 0x47         ; I/O descriptor tag - PATAAR
DB 0x00         ; board does not decode the full 16-bit ISA addr
DB 0xF6         ; min base addr 0x3F6
DB 0x03
DB 0xF6         ; max base addr 0x3F6
DB 0x03
DB 0x02         ; alignment
DB 0x02         ; 2 contiguous  ports
;
DB 0x30         ; small item, start dependent function 1
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; I/O Descriptors
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x47         ; I/O descriptor tag - PCDRAR
DB 0x00         ; board does not decode the full 16-bit ISA addr
DB 0x00         ; min base addr 0x200
DB 0x02
DB 0xF8         ; max base addr 0x3F8
DB 0x03
DB 0x08         ; alignment
DB 0x08         ; 8 contiguous  ports
;
DB 0x47         ; I/O descriptor tag - PATAAR
DB 0x00         ; board does not decode the full 16-bit ISA addr
DB 0x00         ; min base addr 0x200
DB 0x02
DB 0xFE         ; max base addr 0x3FE
DB 0x03
DB 0x02         ; alignment
DB 0x02         ; 2 contiguous  ports
;
DB 0x38         ; end dependent function tag
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GAME Logical Dev ID
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x15         ; small item, external logical dev tag
DB 0x04         ; Vendor ID Byte 0
DB 0x96         ; Vendor ID Byte 1
DB 0x00         ; Vendor assigned function ID (Byte 0)
DB 0x02         ; Vendor assigned function ID (Byte 1)
DB 0x02         ; flags[1] - supports commands at index 0x31
;
DB 0x30         ; small item, start dependent function 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; I/O Descriptors
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x47         ; I/O descriptor tag - P201AR
DB 0x00         ; board does not decode the full 16-bit ISA addr
DB 0x01         ; min base addr 0x201
DB 0x02
DB 0x01         ; max base addr 0x201
```

```
DB 0x02
DB 0x01         ; alignment
DB 0x01         ; 1 port
;
DB 0x30         ; small item, start dependent function 1
;
DB 0x47         ; I/O descriptor tag - P201AR
DB 0x00         ; board does not decode the full 16-bit ISA addr
DB 0x41         ; min base addr 0x241
DB 0x02
DB 0xC1         ; max base addr 0x3C1
DB 0x03
DB 0x40         ; alignment
DB 0x01         ; 1 port
;
DB 0x38         ; end dependent function tag
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Sound Blaster/AdLib
;   Logical Dev ID
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x15         ; small item, external logical dev tag
DB 0x05         ; Vendor ID Byte 0
DB 0xA4         ; Vendor ID Byte 1
DB 0x03         ; Vendor assigned function ID (Byte 0)
DB 0x03         ; Vendor assigned function ID (Byte 1)
DB 0x02         ; flags[1] - supports commands at index 0x31
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; IRQ Descriptor
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x22         ; small item, irq tag
DB 0xBC         ;  one of IRQ2/9, IRQ3, IRQ4, IRQ5
DB 0x00         ;  none supported
;
DB 0x30         ; small item, start dependent function 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; I/O Descriptors
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x47         ; I/O descriptor tag - P388AR
DB 0x00         ; board does not decode the full 16-bit ISA addr
DB 0x88         ; min base addr 0x388
DB 0x03
DB 0x88         ; max base addr 0x388
DB 0x03
DB 0x02         ; alignment
DB 0x02         ; 2 ports
;
DB 0x30         ; small item, start dependent function 1
;
DB 0x47         ; I/O descriptor tag - P388AR
DB 0x00         ; board does not decode the full 16-bit ISA addr
DB 0x08         ; min base addr 0x208
DB 0x02
DB 0xC8         ; max base addr 0x3C8
DB 0x03
DB 0x40         ; alignment
DB 0x02         ; 2 ports
;
```

```
DB 0x38        ; end dependent function tag
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MPU401Emulation
;   Logical Dev ID
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x15        ; small item, external logical dev tag
DB 0x04        ; Vendor ID Byte 0
DB 0x96        ; Vendor ID Byte 1
DB 0x00        ; Vendor assigned function ID (Byte 0)
DB 0x04        ; Vendor assigned function ID (Byte 1)
DB 0x02        ; flags[1] - supports commands at index 0x31
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; IRQ Descriptor
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x22        ; small item, irq tag
DB 0xBC        ;  IRQ2/9, IRQ3, IRQ4, IRQ5 and IRQ7 supported
DB 0x00        ;  none supported
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; I/O Descriptors
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x47        ; I/O descriptor tag - P401AR
DB 0x00        ; board does not decode the full 16-bit ISA addr
DB 0x00        ; min base addr 0x300
DB 0x03
DB 0x30        ; max base addr 0x330
DB 0x03
DB 0x10        ; alignment
DB 0x02        ; 2 ports
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; End Tag
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DB 0x79        ; end tag
DB 0x00        ; checksum - calculated on all above bytes
```

µ-law	A compression and expansion (companding) technique commonly used to encode speech audio in the United States.
µs	microsecond
ADC	Analog-to-digital converter
ADPCM	Adaptive differential pulse code modulation, also called adaptive delta pulse code modulation. A compression and expansion (companding) technique used to encode audio information. It is the encoding method used in the Compact Disc-Interactive (CD-I) format.
A-law	A compression and expansion (companding) technique commonly used to encode speech audio in Europe.
ANSI	American National Standards Institute
ATAPI	A packet interface protocol which addresses CD-ROM connection issues.
big endian	The mode of accessing 16-bit memory values where the most significant bits are in the higher addressed byte. See **little endian**.
BIOS	Basic Input/Output System
codec	Audio coder/decoder module
CPU	Central processing unit
CSN	Card select number, used in the Plug and Play ISA specification
DAC	Digital-to-analog converter
DAK	DMA-Request Acknowledge
DMA	Direct memory access
DRQ	Direct memory access request
DSP	Digital signal processor
EEPROM	Electrically erasable programmable read-only memory. An EPROM that can be reprogrammed while it is in the computer.
EISA	Enhanced Industry Standard Architecture
EX_IRQ	Interrupt request associated with an external device (e.g., CD-ROM).
FIFO	First-in first-out. A queuing technique in which the next item to be retrieved is the item that has been in the queue for the longest time.
frame	One complete LMC/synthesizer cycle. It is the period of time during which all active synthesizer voices are processed.
GUS	The Advanced Gravis UltraSound sound card

host system The PC in which the InterWave IC-based sound card is installed.

IMA Interactive Multimedia Association

IOCHK Non-maskable interrupt request, ISA bus

IOR I/O read

IOW I/O write

IRQ Interrupt request

ISA Industry Standard Architecture

LFO Low-frequency oscillator

little endian The mode of accessing 16-bit memory values where the most significant bits are in the lower addressed byte. See **big endian**.

LMC The InterWave local memory control module.

LMPF The InterWave local memory playback FIFO.

LMRF The InterWave local memory record FIFO.

local memory DRAM and ROM dedicated to the InterWave hardware functions.

LSB Least significant bit

MIDI Musical Instrument Digital Interface

ms millisecond

MSB Most significant bit

NMI Non-Maskable Interrupt

PCM Pulse code modulation

PIO Programmed input/output. A technique for transfering data into and out of system memory using one or more I/O ports. In the InterWave IC, it is used as an alternative to DMA transfer when DMA channels are not available.

PNP Plug and Play. Used to indicate compliance with the Plug and Play ISA specification which allows

RAM Random-access (read/write) memory

ROM Read-only memory

SB System bus

SBI System bus interface

STSYNC Serial Transfer Sync (synchronization)

subframe The period of time within a frame during which one synthesizer voice is processed.

system memory The host system's random-access memory (RAM).

synth Synthesizer. In most cases, this term refers specifically to the InterWave synthesizer module.

TC Terminal Count (ISA bus)

UART Universal Asynchronous Receiver/Transmitter

WAV file A binary file representing audio data that the synthesizer can use as wavetable data.

wavetable data
Synthesizer data stored in local memory.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

R

S

T

U

V

W