# *GoldServe™*

**AuSIM3D™ Gold Series Audio Localizing Server System**

**User's Guide and Reference**

Covering both Hardware and Software

AuSIM, Inc.

Manual Revision 1d, October 2001



AuSIM, Inc. acknowledges all trademarks found in this manual.

**AuSIM, Inc.**
4692 El Camino Real, Suite 101
Los Altos, CA 94022
Phone: (650) 322-8746
FAX: (561) 325-0849

AuSIM has moved to

241 Polaris Avenue
Mountain View, CA 94043

fax is now (772) 325-0849

e-mail: info@audiosimulation.com
web: http://www.ausim-inc.com

# Contents

Chapter 1

# Introduction

## Overview

Interactive real-time auditory localization greatly enhances the effectiveness of virtual environments. By presenting sounds to a listener as located in three-dimensional space, an application transmits more information to the user, stimulating situational awareness and creating a sense of immersion in a virtual environment. The audio simulation technology, AuSIM3D™, from AuSIM, Inc. uses physical modeling and empirical data to synthesize a sound space in a completely natural and realistic way. When listening to a system incorporating such technology, a user not only feels immersed by real-world, three-dimensional sounds, but also can use natural filtering to discern and comprehend any of several layered concurrent sound streams.

The GoldServe™ audio-localizing systems from AuSIM, Inc. incorporate AuSIM3D™ technology and perform real-time localization of multiple real-time audio sources. For each audio source, the system produces a left and right output pair dependent on the direction of emission from the source, path of propagation, and direction of arrival to the listener. The output pairs corresponding to each source are mixed and played through conventional headphones or nearphones. The processing creates the perception that the source is positioned at any specified location in three-dimensional space.

GoldServe™ is a complete 3D sound-localization server subsystem to be a peripheral to a "host" computer running a user's application. The host can be any modern computer workstation, or several in fact. Host computers can control a GoldServe™ system via an RS-232 communication protocol (called ATRON), which is easily implemented in the user application through a high-level 'C' application programming interface (API). While the API hides the low-level ATRON RS-232 protocol, it is completely disclosed and may be programmed directly for access on as yet unsupported host platforms. From the GoldServe™ perspective, the user's application is the "client" that sends commands to the GoldServe™ "server" for service. The GoldServe™ client/server scheme is functionally diagrammed in Figure 1, below.

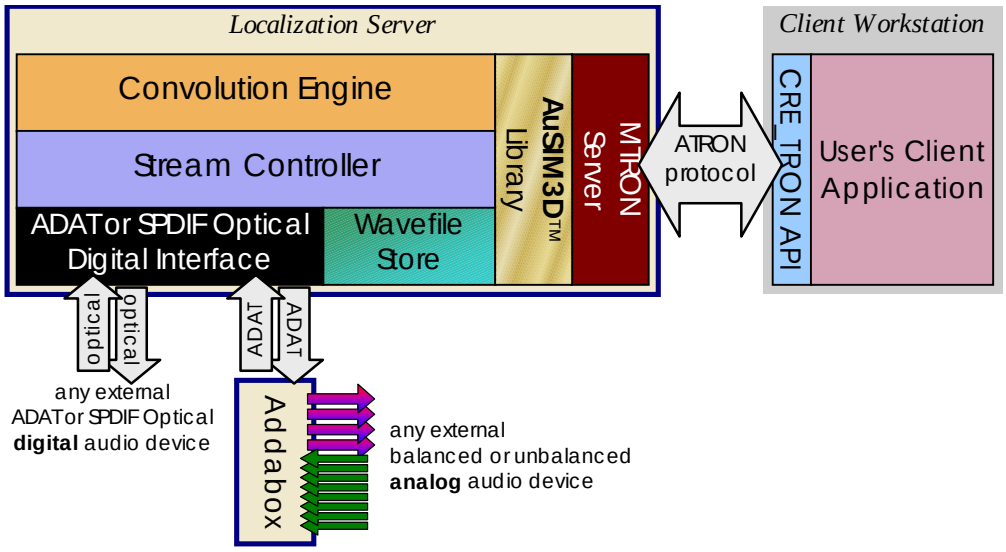



Figure 1. **The GoldServe™ functional schematic.**

Currently, the only supported client API is CRE_TRON, an API that was developed in 1991 and popularized by Crystal River Engineering, Inc. (CRE). The GoldServe™ server responds to the ATRON RS-232 protocol, also a CRE development. These two interfaces (high- and low-level, respectively) were featured on the popular Acoustetron and Acoustetron II sound-localization-servers from CRE. By maintaining backward compatibility with these interfaces, GoldServe™ systems are instantly supported by scores of existing end-user applications.

## GoldServe™ System Components

The system consists of an embedded computer hosting an audio filtering engine and digital audio stream controller, a 16-channel digital audio interface, an 8-channel analog audio interface, and monitoring console. The filtering engine is optimized to filter 8 streams with 64 coefficients per left/right pair, 16 streams with 24 coefficients per pair, or 5 streams with 128 coefficients per pair. All filtering is performed with 32-bit floating-point accuracy. All digital audio streams are maintained with 21-bits of resolution. The analog interface supports 24-bit encode and decode at 44.1 or 48.0 kHz.

GoldServe™ includes server software to monitor the RS-232 ports and translate protocol strings into localization control, as well as over 100 sample wavefiles for prerecorded playback. The system also provides a client library with several examples to allow user applications to easily control the server.

### Configurations

GoldServe™ systems can be packaged in many configurations. The basic package, called GoldMiner, includes:

- a digital audio processing unit (DAPU) supporting sixteen (16) digital audio streams in and out,
- a minimal console (monitor, keyboard and pointing device) for direct DAPU control,
- a pair of high-quality, open, circumaural headphones
- an analog interface for four channels of monaural input and four pairs of binaural output,
- a four-way (4), high-gain headphone amplifier, and
- a portable rackmount enclosure.

The DAPU of any GoldServe™ system may be configured with one, two, or four processors, which are called GoldSolo, GoldDual, and GoldQuad, respectively. When included in the basic package, such processor configurations are called GoldMinerSolo, GoldMinerDual and so forth.

## Organization of this manual

Chapter 1 "Introduction" describes the audio localization process and its implementation on an AuSIM GoldServe™ system.

Chapter 2 "Getting Started" reviews hardware and software requirements for using GoldServe™, explains what comes with a GoldServe™ system (standard and optional) and how to install it, and tells you how to obtain technical support and service.

Chapter 3 "Using the GoldServe™" explains how to test your GoldServe™, and how to write your own programs for controlling the GoldServe™ AuSIM3D system.

Chapter 4 "CRE_TRON Function Reference" describes the function calls that are used to program your GoldServe™.

Chapter 5 "Glossary" contains a list of often-used terms relating to AuSIM3D sound.

## Audio localization

The localization processing of two independent sound sources is illustrated in Figure 2. This schematic can be superposed to represent any number of sources. Each source is processed through a sequence of dynamic models, each representing independent characteristics of wave propagation. First, a “Source Model” filter accounts for emission amplitude and the directivity of the emitter. A human voice, for instance, sounds dimmer when speaking away from the listener than towards. Next, a "Propagation Model" filter accounts for attenuation due to spreading and coloration due to friction in the medium. Next, the signal is split due to the differences in travel time to each ear, and a Head-Related Transfer Function (HRTF), a pair of filters each representing the directivity of an ear, is applied for the directional effect. Finally, the Left and Right outputs from the respective source pipelines are summed together and output as a binaural mix for headphone presentation.

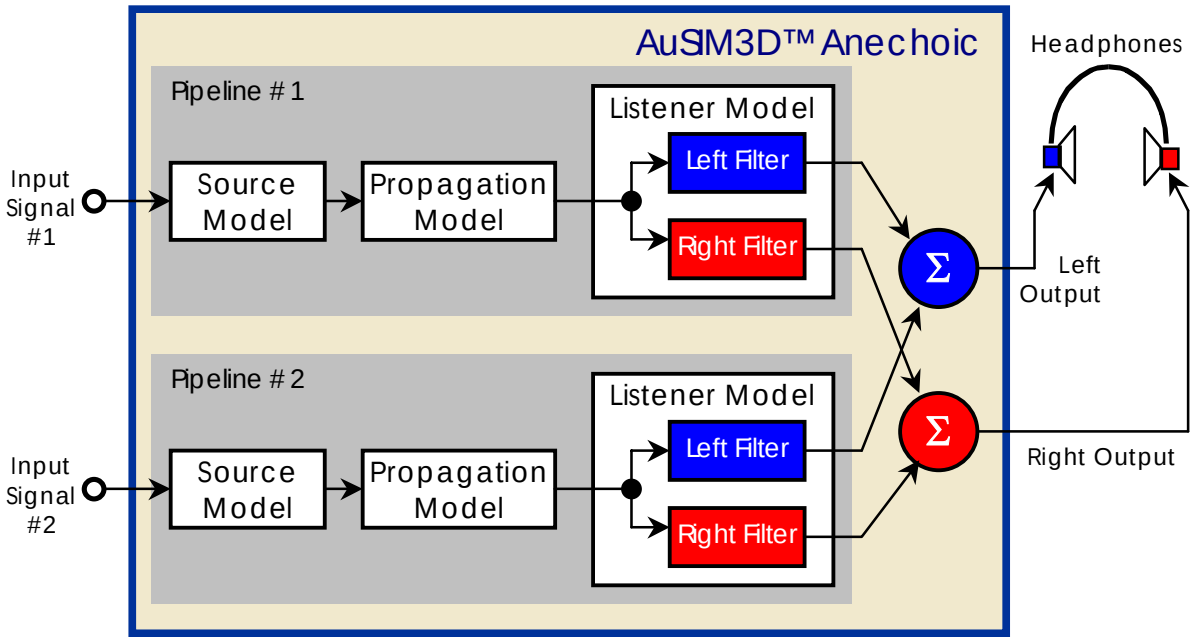



Figure 2. **AuSIM3D™ anechoic localization process on the GOLDSERVE™.**

All models are *dynamic*, in that their parameters can change up to 60 times per second. All AuSIM3D™ systems are optimized to render changes in less than 40 milliseconds, a latency less than the least *psychologically disruptive* delay[1]. Being entirely composed of dynamic models, the system may be driven interactively.

The Listener Model in GoldServe™ uses complete spherical sets of HRTF's called Acoustic Head Maps (AHM's). The filter characteristics comprising an AHM were obtained from actual measurements on a human head. The GoldServe™ uses Finite Impulse Response (FIR) filters (also known as non-recursive filters).

## Localization inputs & outputs

Audio input signals for localization should be as "dry" as possible, meaning without propagation effects. Dry signal transduction is usually accomplished by miking as directly and close to the emitter as possible. "Wet" signals will possess propagation effects that are in conflict with those simulated by the GoldServe™, and thus have a collapsed or confused image. The GoldServe™ offers dynamic selection between two different kinds of monaural sound sources: real-time external inputs and sampled-sound files in pulse-code-modulated (PCM) format. A third form of input, real-time synthesis, is under development. The GoldServe™ does not support mixing multiple inputs (whether hardware inputs or wavefiles) to a single localization channel.

### Real-time signal input

Real-time external inputs may be either analog or digital streams. All input signal data enters the GoldServe™ digital audio processing unit (DAPU) as a digital stream through one of two optical interfaces. Each optical interface may be configured to receive either the ADAT™ optical (8 channels) or the S/P-DIF optical standard (2 channels). The optical, or "lightpipe", interface requires a standard TOSLINK™ cable for interconnection.

---

[1] Humans can *perceive* aural latencies as small as 5 milliseconds when offered a definitive reference. When given no reference, whether aural, visual, psychomotor, or tactile, humans have no means of perceiving lateness.

For analog audio, the GoldServe™ includes an external component (called "Addabox™", see Figure 1), which provides 8 converters for input from analog to digital and 8 converters for output from digital to analog. These high-fidelity converters are provided external of the DAPU to isolate the sensitive Addabox™ components from the digital noise of the server. The Addabox™ provides 24-bit encoding at either 44.1 or 48 kHz. The analog inputs to the Addabox™ accept balanced (tip = non-inverted "hot", ring = inverted "cold", sleeve = ground, "TRS" jacks) or unbalanced signals at the standard -10 dBV level. Hotter signals, up to +4 dBu can be accommodated via factory adjustment. A second Addabox™ may be added to the GoldServe™.

### Sampled playback input

Any sampled signal stored as PCM data (hereafter generically called "wavefiles") may be replayed by the GoldServe™ system as a source input. All wavefiles are converted to 32-bit floating point and resampled to the simulation sample rate at load time. Unlike legacy CRE audio simulation products, the GoldServe™ system loads all wavefiles entirely into memory upon opening, performing preprocessing as necessary. This mechanism enables the GoldServe™ to accommodate wavefiles opened from a variety of locations, including hard disk, CD-ROM, or mounted network drive. GoldServe™ does not currently support CD-A, Internet URL's, or compressed formats such as MP3.

### Displays for localized sound

The key to localizing sounds for a particular listener is to compute the signal shapes as they should be entering the listener's respective ear canals, and then deliver them to the canal entrance. A pair of signals intended for delivery at the ear canal entrance, whether computed or measured, is called *binaural*. To contrast, stereo signals are intended for loudspeaker display, utilizing the listening environment for the final propagation to the listener. If the characteristics of the listening environment are well known (essentially the transfer function from the loudspeakers to the listener), then compensation could be added to binaural signals, allowing loudspeaker display while retaining the true localization. However, measuring and maintaining these listening environment characteristics are very difficult in practice, leaving headphone or nearphone displays the most viable.

The optimal binaural display should occlude the immediate aural environment, so that ambient or local sound does not interfere with or collapse the virtual image depicted by the binaural signals. Closed, circumaural (meaning surrounding the entire pinnae) headphones are thus the best binaural display.

In the continuum between closed headphones and arbitrarily placed loudspeakers, a wide-variety of aural displays define an equally wide variance in localization display quality. Figure 3 illustrates this continuum, with the displays supported by the GoldServe™ highlighted in **bold**. All displays represented assume stereo pairs with single driver elements. Multi-driver loudspeakers diffuse the localizability and increase compensation complexity. Multi-speaker arrays are designed create a physical sound field in the listener's environment, are thus not compatible with binaural audio, and require a very different type of technology to compute compatible signals.

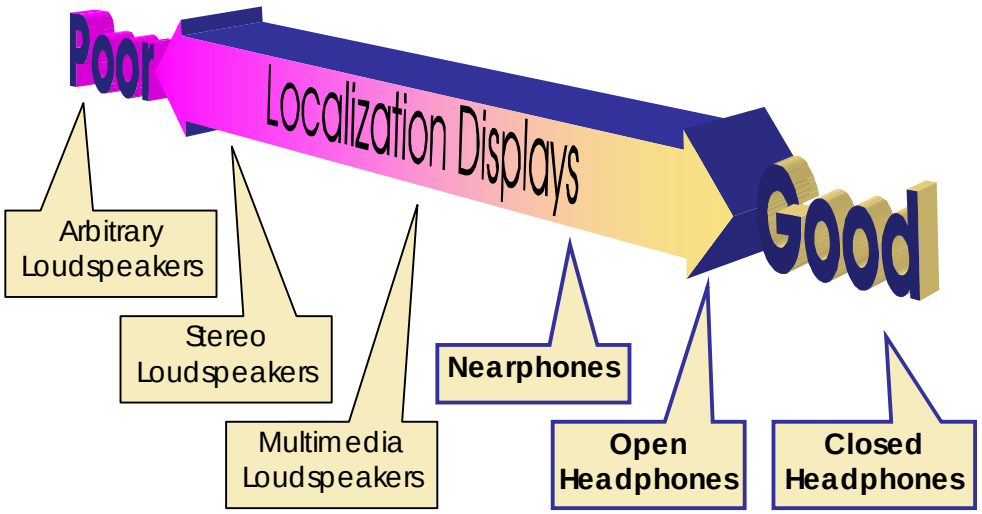



Figure 3. **The continuum of localization displays.**

### Localized output

The basic GoldServe™ system, GoldMiner, offers streaming binaural output for up to four independent listeners. A single Addabox™, as included with the basic system, converts digital streams for four analog output pairs. The Addabox™ outputs are differential line-level signals, but may be tapped with unbalanced gear. These line-level outputs must be amplified to support the full dynamic range localization requires. The GoldServe™ does not support output to file on disk.

Chapter 2

# Getting Started

## Requirements

### Hardware

Since the GoldServe™ is a complete peripheral system, the only hardware requirement is an available RS-232 serial port on the client host system (user's workstation) through which to control the GoldServe™ audio server.

The GoldServe™ also provides network access to its file system via the standard Common Internet File System (CIFS). To gain direct access to the GoldServe™ file system, you will need an ethernet port on your TCP/IP network.

### Software

The GoldServe™ is currently available with client software libraries and serial drivers for Win9x, WinNT4, Win2K, MS-DOS, LINUX, IRIX, SOLARIS, and HPUX operating systems.

The graphical-user-interface (GUI) demonstration software for UNIX systems, audioClient, requires an X Windows environment to operate. The audioClient application is specifically configured for Motif.

While all Win32 clients have CIFS built-in, clients running other operating systems may have to verify the existence and proper setup of a CIFS client to gain network access to the GoldServe™ file system.

### Input and output amplifiers

The analog input to the GoldServe's™ Addabox™ is line-level (-10 dBV), and must be pre-amplified for microphone input. When streaming live sound, use a high-quality microphone coupled with a microphone pre-amplifier.

Spatial audio requires a lot of dynamic range to simulate proper, realistic distance attenuation. The GoldServe's™ localized output routed through the Addabox™ analog converters is line-level (-10 dBV), and must be amplified for most headphone displays to achieve full dynamic range. The headphone amplifier is provided for this purpose. Please note that the supplied cables to connect the GoldServe™ Addabox™ to the amplifier are balanced (differential signal) audio cables. ***Use of unbalanced cables will compromise your dynamic range.*** Headphones, nearphones, speakers, or amplifiers can be connected to the headphone amplifier outputs. When the headphone amplifier is properly set, a normalized source located at virtual arm's length with **`cre_amplfy_source`**`()` set to 0 dB, should sound very loud.

## Output devices

Your GoldServe™ can be connected to a number of different output devices. The `cre_define_head(AtrnDISPLAYtype)` function call is used to select an output device. The default device is headphones. Different devices will provide different levels of localization performance. Their ranking in order from best to worst: headphones, nearphones, quad speakers, multimedia stereo speakers, and other stereo speakers.

### Headphones

Headphones are an important part of a virtual acoustic system. Your GoldServe™ system has been optimized for use with circumaural, "diffuse-field equalized" headphones, i.e. headphones which enclose your entire ear, as opposed to headphones that are placed inside the ear canals. For optimal localization results, we strongly recommend usage of high-quality headphones.

The reference headphones for use with AuSIM HRTF models, are Sennheiser HD250 II's (contact AuSIM for availability).

Depending on your virtual audio application, the difference between acoustically open or acoustically closed headphones might be important. Open headphones (such as the Sennheiser HD540 II's) do not provide a tight seal between the ear and the environment. Such headphones are useful in applications where the user has to be able to hear sounds from the surrounding environment (operator's voice, warning signals) during the simulation. Acoustically closed headphones try to suppress all sounds other than the ones delivered by the headphones. They are useful in completely immersive, virtual environments, or to attenuate noisy surroundings (trade shows, noisy workspaces).

If you plan to use electro-magnetic head-tracking devices in conjunction with your GoldServe™, use headphones with as few metal parts as possible, in order to avoid electro-magnetic interference between headphones and tracking sensors. Try to avoid tracking systems that use transmission frequencies between 20-20,000 Hz (the audible region).

## Nearphones

In applications where unobtrusive equipment is important, nearphones can be used instead of headphones. Nearphones are two speakers (left and right signal), placed near (within 25 inches) the user's ear. An example of where nearphones are applicable would be a simulator cab with a projection screen and two speakers mounted next to the user's seat. The user can get in and out of the cab by simply sitting down in a chair. To achieve optimal results the user's head should not move out of the range of the speakers, or turn more than 45 degrees in any direction. The closer the speakers are placed to the user's ears, the better the resulting localization perception.

## Quad speakers

If speakers cannot be placed close to the listener, a quad speaker setup - left front/back, and right front/back - is recommended. In this case, the left output of the GoldServe™ would be wired to both left speakers, the right output to both right speakers. The listener should be placed near the center of the square formed by the four speakers.

## Multimedia stereo speakers

The term 'multimedia speakers' refers to a stereo speaker setup where the speakers are built into a computer monitor or are located close to the sides of a monitor. In such a setup, the user's position is assumed to be directly in front of the monitor, forming a more or less fixed geometry between the listener and the two speakers. Special processing of the left and right audio signals is applied to enhance the 3D effect in such a setup. Please note that multimedia speaker processing is sweet spot (see glossary) limited.

## Other stereo speakers

This category includes all other forms of stereo speaker setups. A speaker layout that does not fit the nearphone, quad, or multimedia categories, is unlikely to produce a convincing 3D effect, and is therefore not recommended.

## System components and specifications

A GoldServe™ base system (GoldMiner) consists of the following items:

- GoldServe™ digital audio processing unit (DAPU) with wavefile archive, dual ADAT optical digital audio interface, and GoldServe™ server software,
- GoldServe™ Client Software Library and Demos on CD,
- GoldServe™ Manual,
- Minimal console (monitor, keyboard, and pointing device),
- Enclosing, pre-wired rackmount case with power distribution bar,
- Open, circumaural headphones and a headphone amplifier,
- Pre-installed wavefile collection (100 wavefiles of general type: vehicles, animals, machines, explosions, effects, instruments),
- Cables: two 1/4 inch balanced audio, four TOSLINK optical, and one RS-232 null-modem serial cable (specific to your host computer).

The system specifications:

- Localization:
  see the performance document that came with your system.
- Pitch: 20-500% shift control for all sources
- Dynamic update rate:
  see the performance document that came with your system.
- Active audio buffering: over 20 minutes (configurable to over 100 minutes)
- Analog Input: 128X oversampled, 24 bit A/D converters
- Analog Output: 8X oversampled, interpolating filters
- Stereo crosstalk: 100Hz-100dBV, 1kHz-80dBV, 10kHz-60dBV
- Wavefile archive: over 20 hours

# Installation

## Hardware

Follow these steps to setup and test your GoldServe™:

1. Connect the GoldServe™ cables as follows:
    - Connect the ¼ inch stereo cable labeled "MON" to the ADAT jack marked "MON".
    - Connect a pair of headphones to the headphone jack on the headphone amplifier.
    - Connect the headphone amplifier power cord into the power distribution strip mounted in the back of the rackmount case,
    - Connect the black power cable labeled "POWER" to the GoldServe™ DAPU receptacle marked "POWER".
    - Plug the heavy power cable from the power distribution strip located in the back of the rackmount case into an external 120vAC-power supply.
    - Connect the mouse and keyboard cables labeled "MSE" and "KBD" to the GoldServe™ DAPU jacks labeled "MSE" and "KBD"
    - Connect the monitor power cord into the power distribution strip mounted in the back of the rackmount case, and plug the other monitor cable into the DAPU jack labeled "VGA".
    - Plug the end of the RS-232 serial cable labeled "AuSIM Server (COM1)" into the jack labeled "COM1" on the back of the DAPU, and plug the other end into the "COM2" port also on the back of the server DAPU. (This will allow the server installation to be verified; later, the second end of the RS-232 serial cable will need to be plugged into an available COM port on the client machine.)
    - Toggle the power distribution strip switch so that the red light is illuminated, indicating that the strip is providing power.
    - Boot the DAPU by toggling the power switch on the front of the DAPU unit, and let the system boot into WinNT.

2. Press the "CTRL", "ALT", and "DEL" keys simultaneously when prompted, and at the logon prompt, type in "Golddigger" (without the quotes) for the user name and "nugget" (without the quotes) for the password.

3. When the Windows NT desktop appears, the server window should also appear, accompanied by a double beep to indicate the AuSIM server application has been activated. You should also see a spinning tumbler in the upper right-hand corner of the server window that indicates the server is serving properly. If there is no tumbler, advance to the next chapter on startup problems.

4. Click on the "START" button on the lower left-hand side of the screen, then select "AuSIM", from the menu, then "Test", then select from the listed test applications. You should hear a demo running on your system. If you can see the tumbler spinning on the screen, but there is no audio, check the connection from the GoldServe™ to the headphone amp, and from the headphone amp to the headphones. If you hear the demo, your server is functioning properly.

5. Next, connect the GoldServe™ to your client system (PC, SGI, SUN, or HP) via the serial cable that was provided for your specific system. On the server end, the serial cable will be connected to the "COM1" jack; on the client end, the default serial port is port number one (COM1 on PC systems, TTYD1 on UNIX systems). To select a different serial port, please refer to the environment variable section.

6. Install the GoldServe™ client software onto your client system (from floppy disk switch to floppy drive and type install for instructions, from DAT, or ¼-inch tape, use tar command to extract software).

7. To test the client/server connection move to the Client\Test directory on your client system and select "demo" or "test" to start up a demo sequence that is controlled from your client system.

8. If your GoldServe™ successfully initializes and plays sounds, your system is installed and ready to use. If the GoldServe™ is not responding correctly, please proceed to the next section on startup problems.

## Startup problems

**Problem**: Your local demo program does not produce sound and does not start properly (no spinning tumbler on the screen):
**What to Do**: Either your server environment variables, PC startup files (config.sys and autoexec.bat,) or DSP card address switches have been changed. A call to the factory (phone number on front cover) is your best bet.

**Problem**: Your local demo program does not produce sound but there is a tumbler:

**What to Do**: The server is up and running and very likely producing sound. The problem must be in the audio connection between the server and your ears. Check all connectors and make sure the headphone amp is powered on.

**Problem**: The server runs in local demo mode, but does not respond to the client:

**What to Do**: The communication link is most likely the problem.

Serial communication link:

Make sure both ends are operating at the same baud (see chapters on Startup menus and Environment variables for details).

On WinNT systems, the client and server baud should both be set to 115200. The server is so set at the factory, but the client may be set by setting the environment variable TRONCOM, by adding the following line in the client's AUTOEXEC.BAT file: set TRONCOM=1@1152,30

## Client software and directory organization

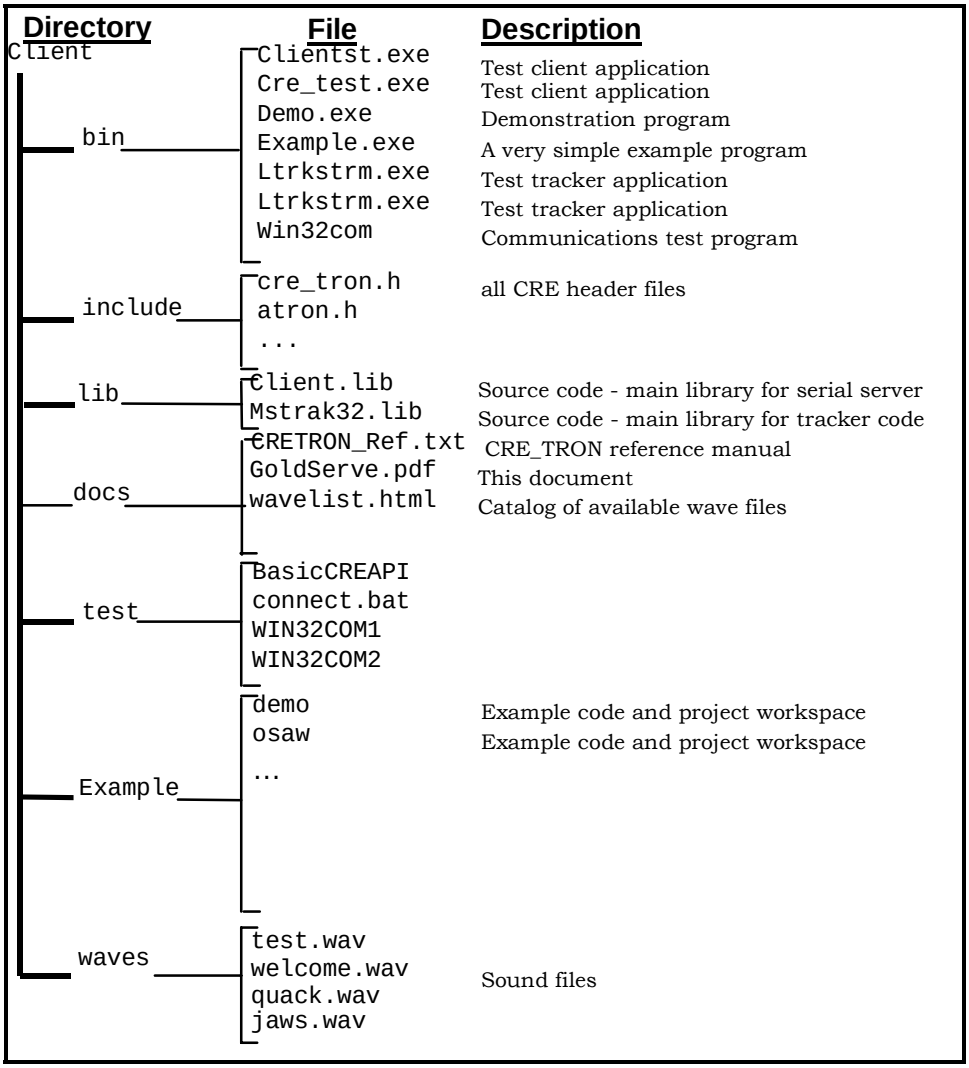



Figure 4. **Client directory organization, WinNT/Win9x client.**

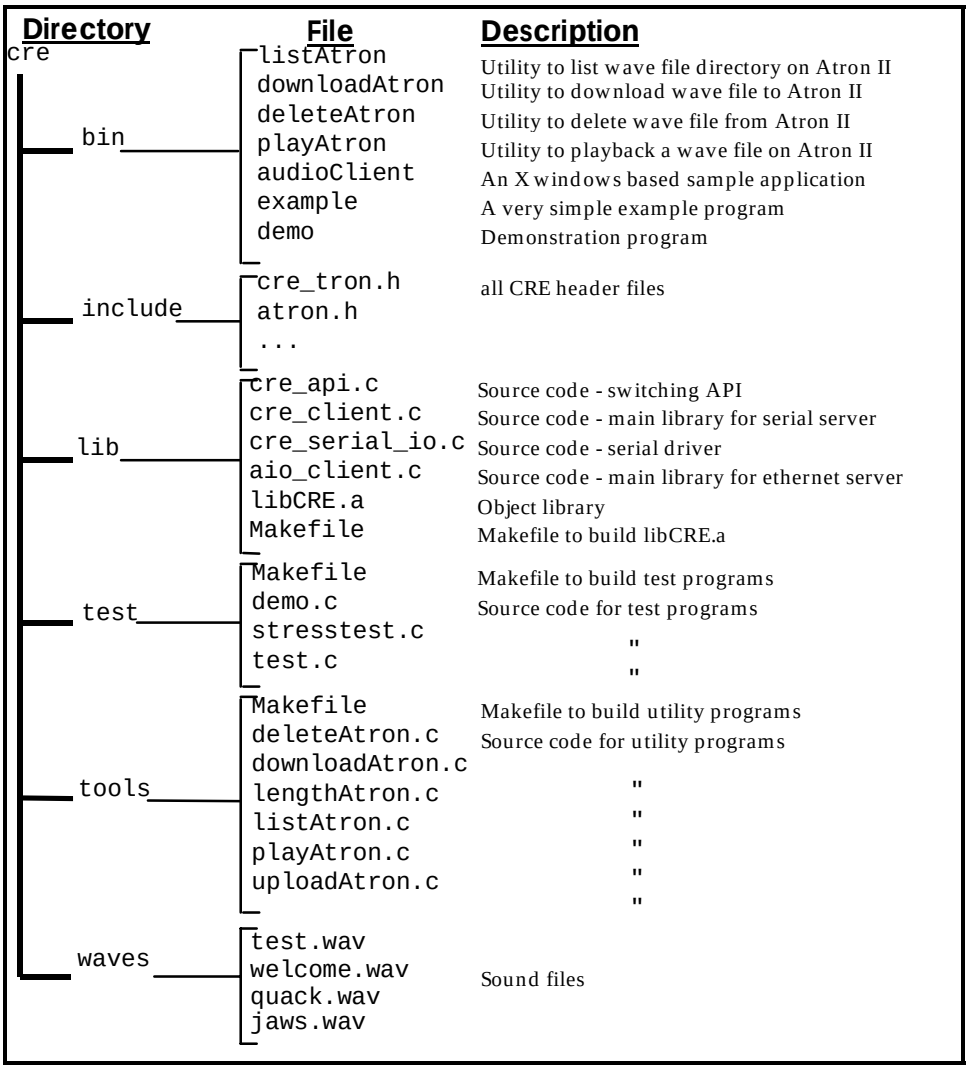



Figure 5. **CRE directory organization, UNIX client.**

# Problems

## Technical support

If you are having difficulties with the operation of your GoldServe™, be sure to review the Installation procedure described earlier.

If you can't solve your problem, you should contact technical support at AuSIM, at the address or phone numbers listed in the inside front page of this manual. Please be sure to have available as much as possible of the following information:

- GoldServe™ software version
- GoldServe™ serial number (from label near power on switch)
- If possible, example code that allows us to reproduce your problem at the factory

## Repair

Before returning faulty equipment or media for service, you first need to obtain authorization from AuSIM or from your distributor.

## Bugs

Please report suspected or confirmed software problems to AuSIM, at the e-mail address or phone numbers listed in the inside front page of this manual. It is essential that you include a complete description of the problem, in sufficient detail that we can reproduce it.

Chapter 3

# Using the GoldServe™

## System start up

On startup, the GoldServe™ system performs a number of self-tests, and then boots up the server. It is useful to talk about two different modes of operation for your GoldServe™: *development* or *playback*. During development of an application, the server should be connected to its monitor and keyboard, in order to make several options and run-time information accessible to the developer. Once an application is developed, and the GoldServe™ is used for run-time playback only, it can be run stand-alone without the need for a monitor or keyboard.

### Development usage

On boot-up of the system the following menu is displayed for a short while:

```
Please make a choice (you have 5 seconds before the default
gets selected)
  1. boot in MS-DOS mode
  2. boot in WinNT mode (selected)
  3. boot in Windows 98 mode
```

Wait five seconds, and the server will boot to WinNT, the default OS, and the correct one for normal development usage of the GoldServe™ system.

If the ethernet option is installed the menu will be slightly different:

Press the “CTRL”, “ALT”, and “DEL” keys simultaneously when prompted, and at the logon prompt, type in “Goldminer” (without the quotes) for the user name and “nugget” (without the quotes) for the password.

When the Windows NT desktop appears, the server window should also appear, accompanied by a double beep to indicate the AuSIM server application has been activated. You should also see a spinning tumbler in the upper right-hand corner of the server.

The GoldServe™ server has now started up and is waiting for communications from the client, ready to render sounds. Once the server is running, a few keyboard commands can be issued to control it:

SPACEBAR toggles the console display from a static screen to a screen that shows all on-going communication between the client and server (on start-up it is set to static). ***Please note that printing updates to the screen slows down the framerate of the GoldServe™***.

ESCAPE KEY will restart the server without rebooting the entire system.

The server and client normally operate at 115200 baud. Changing the baud rate is not recommended, unless some special circumstances require it. Baud rate change can be accomplished by changing the environment variables. To see how to do this for your GoldServe™ system, see the section entitled "Environment Variables" below.

### Run-time usage

To use your GoldServe™ with an existing application, simply connect the client and server systems and connect all audio equipment (headphone amp, cables, and headphones). The monitor and keyboard are not necessary for operation, since the system can simply be powered up, and is ready to be accessed from the client after the GoldServe™ has emitted a dual beep.

## Test and example programs

Once hardware and software installation is complete, you may test the GoldServe™ system in a number of ways:

### demo

The demo sequence can be started from both the server and the client end. It serves as a verification of server functionality.

### Stress creOrbit

A test program available on your client system. It will start up eight sounds that are pre-installed on your GoldServe™ system, and will orbit them around the listener's head randomly until the program is stopped (ESC). This program challenges the system resources, and can be used to determine whether the resources available to a given system are capable of supporting the expected number of sound sources.

### example

A very simple client example program that moves a sound around in space. This program is a good place to look for a simple code sample to get you started on writing your own GoldServe™ applications.

## Application programs

The following sample application is provided on UNIX platforms:

### audioClient

If running X Windows with the Motif widget set, the audioClient program can be used to verify the functionality of your GoldServe™. Once started on the client, it presents a graphical user interface, and a two dimensional graphical representation of the listening space, that allow you to load wave files, play them, and move the sounds and the listener around in a space that includes sound reflecting walls.

## Environment variables

The GoldServe™ server can be controlled using the following (optional) "environment variables" on the client system:

| Variable | Description | Default (if not set) |
|---|---|---|
| TRONCOM | Selects communication parameters | 1@1152,30 |
| TRONDEV | Overrides the serial port device name on non-PC client hosts. | `/dev/ttyd1` (UNIX) |

The TRONCOM variable needs to be set if you want to operate your GoldServe™ on a different setting than serial port 1, at 115200 baud.

#### WinNT

For WinNT clients, environment variables are set in the System panel within the control panel. (need image of System Control Panel Environment Sheet)

#### Win95, Win98, MS-DOS

For Win9x and MS-DOS clients, the environment variables should be entered in the C:\AUTOEXEC.BAT to make them available to client programs invoked from the command processor. The syntax for the AUTOEXEC.BAT file (or equally at a command prompt in a console window) is as follows:

```
set TRONCOM=x@yyy,zzz
```

#### UNIX

For UNIX, the environment variables should be set in a .cshrc or similar start-up script so that their values are available in any shell. The syntax is as follows:

```
setenv TRONCOM x@yyy,zzz
```

where x is the serial port number (TTYDx or COMx), yyy the baudrate divided by 100 (100's of bits per second), and zzz the time-out period (the amount of time the client will wait for a response from the server on an init() call). The time-out parameter is optional.

The TRONDEV variable is optional and only needed if your client system's serial port has a different file descriptor than the defaults. For example,

```
setenv TRONDEV /dev/ttya
```

sets the serial port device to `TTYA` from `TTYD1`. TRONDEV will override the port number defined by TRONCOM the variable.

# CRE_TRON Software Interface (API)

## Overview

CRE_TRON is a 3D audio application-programming interface (API) that was developed by Crystal River Engineering in the early 1990's to facilitate the creation of interactive three-dimensional AuSIM3D sound spaces.

The goal of the CRE_TRON API is to allow a user or developer to build up a sound space using the concepts of physical reality without having to know about the underlying algorithms, implementation or audio hardware.

This API implements the concept of a sound space in the form of easy-to-understand objects. Objects include sound emitting sources, sound reflecting surfaces, and sound receiving listeners. Sounds get created by sources, such as a ringing phone, propagate through space, bouncing off passive objects such as walls, and finally reach a listener's ears, where they are received and interpreted.

The C function calls listed below are used to write programs to control the GoldServe™. They are described in detail in the "CRE_TRON Function Reference". A good place to start programming is by expanding on the `demo.c` example code in the AuSIM3D\Client\Examples\demo directory. This directory also contains a sample workspace file demo.dsw that can be used in conjunction with the demo.c file to create a sample client application.

The function calls that allow a programmer to interact with the GoldServe™ can be grouped into the following categories:

- **General system** functions declared in CRE_TRON.H:

```
cre_init          (driver, head, sources, mode);
cre_update_audio  ();
cre_close         (driver, head);
cre_end           ();
cre_detect        (prm);
```

- **Listener head** functions declared in CRE_TRON.H:

```
cre_define_head   (id, prm, pts, data[]);
cre_locate_head   (id, hloc);
```

- **Audio source** functions declared in CRE_TRON.H:

```
cre_define_source (id, prm, pts, data[]);
cre_locate_source (id, sloc);
cre_amplfy_source (id, dB);
cre_select_source (id, channel);
```

```
cre_pmeter_source (id, &power);
```

- **Propagation medium** specific functions, declared in CRE_TRON.H:
```
cre_define_medium (volm, prm, pts, data[]);
```

- **Waveform** functions declared in CRE_WAVE.H:
```
cre_open_wave     (wavefile, mode);
cre_ctrl_wave     (src, wave, cmd, data);
cre_close_wave    (wave);
```

- **MIDI** functions, declared in CRE_MIDI.H:
```
cre_send_midi     (src, midistr);
cre_set_midi      (src, cmd, data);
cre_msg_midi      (src, msg, chnl, data1, data2);
```

- **Acoustic research** functions, declared in CRE_TRON.H:
```
cre_set_rel_pos   (head, src, azim, elev, gain);
cre_get_polar     (head, src, polar);
```

- **Acoustetron test** function, declared in ATRON.H:
```
cre_test_atron    (verbose);
```

## Sample rates and driver selection

The GoldServe™ can run multiple software drivers. Each driver implements the CRE_TRON software interface, but might offer different functionality (see the description of **`cre_init`**`()` for specific driver details). One of the important distinctions between drivers is the sample rate at which they will run the hardware. Two of the more common sample rates are:

- **48,000 Hz:** the sample rate for professional audio. Advantage: good resilience to intensive processing, which results in sustained post-process high-fidelity audio quality and good localization. AuSIM is shifting its processing to 48 kHz. Disadvantage: very high computational requirements.
- **44,100 Hz:** the sample rate for CDs. Advantage: moderate quality for audio and localization. Disadvantage: high computational requirements.
- **22,050 Hz:** the sample rate common in multimedia titles and video games. Advantage: lower system bandwidth, computational resource, and storage requirements. Disadvantage: poor audio and localization quality. 22 kHz sampling is not supported by AuSIM3D. AuSIM intends to upsample all 22 kHz wavefiles in future releases.

Different drivers may be selected at start-up time by one of the parameters of the **cre_init**() call.

Please note that independent of driver sample rate, both 22.05kHz and 44.1kHz wave files can be played back.

## Example code

```
#include <conio.h>
#include <stdio.h>
#pragma hdrstop

#include "cre_tron.h"
#include "cre_wave.h"

#define SourceID     0
#define HeadID       0
#define Sources      2
#define WaveFile  "TEST.WAV"
#define PanLimit   100.0f

void main(void)
{
  float step = -0.2f;
  float SrcLoc[6]  = { 10.0f, PanLimit, 0.0, 0.0, 0.0, 0.0 };
  float HeadLoc[6] = {  0.0, 0.0, -10.0f, 0.0, 0.0, 0.0 };
  wavFt *wave;

  /* initialize two Tron sources, with verbose report */
  if (cre_init(Atrn_ASM1, HeadID, Sources,
              _CONSOLE_|_VERBOSE_) < Ok)
    return;

  /* open WAV file and load wave form using all buffers */
  if (!(wave = cre_open_wave(WaveFile, 0))) {
    printf("\nwave load error.");
    return;
  }
  /* play open wave form as SourceID with repeat loop */
  cre_ctrl_wave (SourceID, wave, WaveCTRL_LOOP, NULL);
  /* locate listener once (not moving) */
```

```
    cre_locate_head (HeadID, HeadLoc);

    printf("\nPress Any Key to Exit ... ");
    /* enter simulation loop until key is pressed */
    while(!kbhit()) {
      SrcLoc[AtrnY] += step;    /* move source location */
      if ((SrcLoc[AtrnY]<-PanLimit) || (SrcLoc[AtrnY]>PanLimit))
         step = -step;          /* reverse panning direction */
      /* set new location as location of source 0 in space */
      cre_locate_source(SourceID, SrcLoc);
      cre_update_audio();       /* flush all changes */
    }
    /* stop wave form playback and detach from SourceID */
    cre_ctrl_wave(SourceID, wave, WaveCTRL_STOP, NULL);
    cre_close_wave(wave);       /* close waveform */

    cre_close(Atrn_CLOS, HeadID); /* close Tron */
    printf("\n");
  }
```

The example program listed above is a "minimal" program which initializes the GoldServe™, loads and plays a wave file, turns on a single source to be used, positions a listener in space, moves a source between two points in space, and "displays" the sound space by updating the hardware.

The **`cre_init`**`()` call will locate and initialize sufficient hardware to localize two sources, locate the listener's head at the origin, and locate a sound source 50 inches directly in front of the head (by default). Since a zero was specified for the units argument, locations will be interpreted in inches, the default units. The `HeadLoc` variable therefore refers to a position 10 inches below the origin. The source is by default a uniform radiator.

After successful initialization, the call **`cre_amplfy_source`**`()` turns source #0 on. Note that all sources are initialized with *no amplification*. In order to hear anything from a source after initialization, **`cre_amplfy_source`**`()` must be used.

The **`cre_open_wave`**`()` and **`cre_ctrl_wave`**`()` calls are used to load a waveform from disk and play it.

The listener is located once to move from the default position to the one specified by `HeadLoc`.

Then the program moves the location of source #0 using **`cre_locate_source`**`()`, and uses **`cre_update_audio`**`()` to flush all changes to the hardware, in order to display the new sound space for the listener, until any character is typed, at which point the hardware is closed.

## Coordinate system

The environment in which localized sounds can be experienced is described by a three-dimensional coordinate system. Within this coordinate system, six-dimensional vectors are used to specify the position and orientation of the listener's head and of all sound sources. The inputs to the GoldServe™ (external or wave forms) are mapped to the corresponding locations in the coordinate system relative to the listener's location.

The GoldServe™ software library represents a six-dimensional location vector as an array of six `floats` (32-bit floating-point number) In this array, the first three elements specify the *x*, *y*, and *z* position in space, in number of "units" (units are selected at initialization of the GoldServe™—see **`cre_init`**`()`). The second three vector elements specify the yaw, pitch, and roll, in radians. They define the orientation of the source or head at position (*x*,*y*,*z*).

The coordinate system is adopted from the vehicle dynamic simulation world. As illustrated in Figure 4, the system is right-handed, with the positive *x*-axis straight ahead and the positive *z*-axis ascending vertically. Orientations are specified as right-handed radian Euler rotations, *roll*, *pitch*, and *yaw*, about respective *x*, *y*, and *z* axes. The six-element vector employed in the GoldServe™ software (in using **`cre_locate_head`**`()` and **`cre_locate_source`**`()`) is ordered <*x*, *y*, *z*, *yaw*, *pitch*, *roll*>. The order of rotations depends upon the rotation basis. With respect to the global coordinate system, from a local coordinate system that initially coincides with the global one, an object is rolled, pitched, yawed, and finally translated.

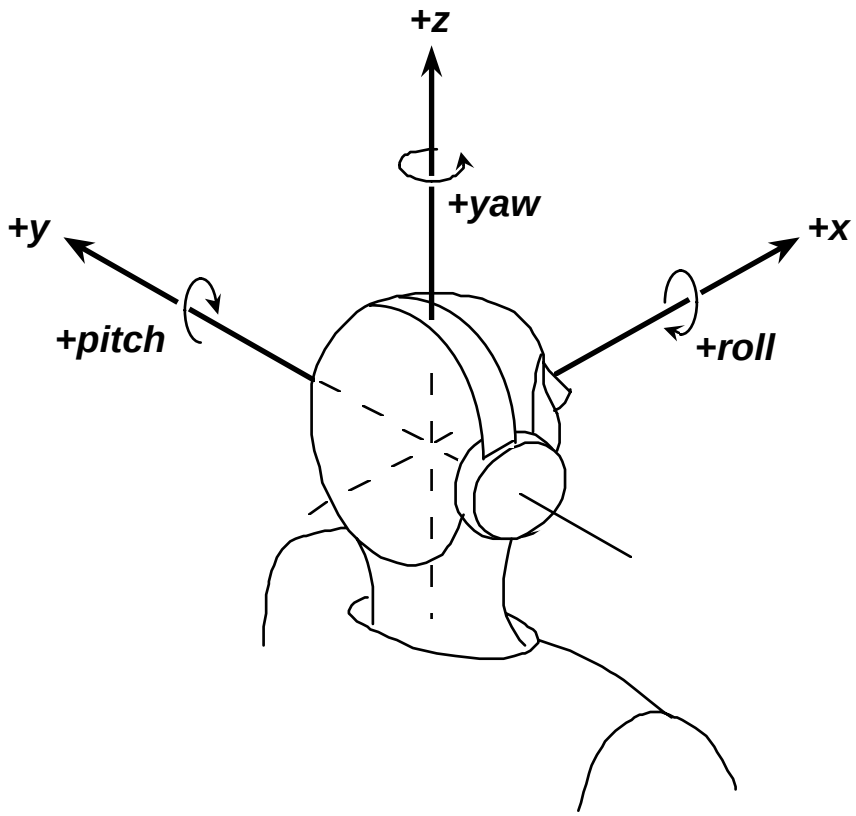



Figure 6 Six-dimensional coordinate system with listener located at (0, 0, 0, 0, 0, 0).

## Head tracking

Closed-loop tracking of head position can provide an important enhancement to real-time audio localization. The proprio-sensation correlation in the human brain between head-movement and aural differences is very tight. Slight incongruities can cause simulated aural images to collapse. Head-tracking allows AuSIM's dynamic audio localization to maintain a consistent source to listener relationship, providing *motion-parallax* localization cues.

The Software applications using AuSIM3D™ libraries work well with many six-degree-of-freedom tracking systems, including:

- Fakespace Labs' BOOM™ integrated mechanical tracker and head-coupled stereo-visual display, or

popular electromagnetic devices such as:

- Polhemus Navigation's Isotrak™, Fastrak™, InsideTRAK™, and 3SPACE® tracker, and
- Ascension Technology's Flock of Birds™ multiple-receiver system, or

popular ultrasonic devices such as:

- Logitech's 6D head-tracker from Fakespace Labs, or

popular inertial/hybrid devices such as:

- InterSense Technology's IS-300, IS-600, IS-900 and the source-less InterTrax.

There is no direct software support for tracking devices in

The AuSIM3D™ client libraries do not directly support these tracking devices, however the vectors resulting from head-tracker sensor sampling are perfectly compatible with AuSIM3D and the CRE_TRON API.

There are many ways to get head-tracker support.

1. All trackers bought through AuSIM, Inc are value-added with AuSIM's AuTrak™ C++ tracker class library for Win32. AuTrak™ provides a single API for all types of trackers, as listed above. AuTrak™ may also be purchased separately.
2. Many simulation toolkits (EAI/Sense8's WorldToolKit, Division's dVise, Paradigm/Multigen's Vega, Centric's EasyScene, etc.) include support for these tracking devices and others.
3. Write your own. Most of these tracking instruments are RS-232 serial devices, for which many operating systems provide good programming tools. Some of the vendors of these instruments provide sample code.
4. Use the AuSIM3D™ service application (GoldServ) running on the GoldServe™ DAPU with its built-in tracking support to directly read your head-tracking instrument and automatically update your listener's position.

For the AuTrak, toolkit, or build-your-own methods see the respective associated documentation. For the GoldServ service application usage, see Appendix A.

# Audio sources

## Sound files

Each GoldServe™ can playback and localize multiple concurrent monophonic sound files. In order to play multiple sound files simultaneously, each sound file must be assigned to a different localization source.

Before a sound file can be played, it has to be loaded onto the GoldServe™. Once a wave file is loaded, it can then be used in an application using the **`cre_open_wave`**() and **`cre_ctrl_wave`**() commands.

If you wish to play the *same* sound file *simultaneously* on multiple sources, you must open that sound file multiple times as well.

The maximum length of a sound file is limited only by the size of the hard disk, since files get automatically played back from disk when playback memory is exhausted. If you hear stuttering (gaps) in the wave file playback, the system bandwidth has reached its limit, and too many files are being started at the same time.

If a sound file's name ends with ".WAV", it *must* be formatted as a "RIFF" file format (per Microsoft Multimedia specifications). WAVE files can be created on the server using the Wave File Editing option available for the GoldServe™, or using other sound recording and editing tools and downloading the sounds from the client. If a sound file's extension is not given, the ".WAV" is assumed and appended.

If a sound file's extension is explicitly something other than ".WAV" the content must be in the CD Audio format: 16-bit PCM words at 44100Hz.

## External inputs

The GoldServe™ can localize concurrent live sounds that are connected via analog or digital inputs. The external inputs can take their data from microphones, CD players, or other sources of analog audio. The physical connectors depend on the GoldServe™ system; see your system-specific guide.

For all drivers.

# Special topics

## Directional radiators

For different audio sources and environments, the sound will travel through the atmosphere in the space in different ways. The directional radiators are used to model this propagation pattern. The following figure shows two different radiation patterns for two different directional sources. The heavy lines indicate contours of constant pressure. As you can see, in front of the speaker, the sound attenuates much more slowly with distance than behind the speaker, where almost no sound propagates. If there were objects placed in the environment, you would see a change in the radiation pattern when the sound is absorbed and reflected from the surfaces of the object.

## Atmospheric absorption

The GoldServe™ includes an "atmospheric absorption" model, which attenuates higher frequencies a greater amount than lower frequencies. The degree of attenuation depends on the distance through which the sound travels in the atmosphere—the further it travels, the greater the relative attenuation. As a result, distant sounds have a lowpass-filtered, or "muffled" characteristic.

This model is controlled by a "distance" parameter. For sounds that are close to the listener, as compared with the absorption distance, the relative high-frequency attenuation will be slight. Conversely, sounds whose range is equal to or greater than this distance will incur correspondingly more high-frequency attenuation.

The atmospheric absorption distance can be accessed and adjusted or disabled from its default value with the **`cre_define_medium`**`()` function.

## Spreading loss roll-off

As sound waves radiate from a point source, their power spreads over an ever-increasing volume of propagation medium. This spreading reduces the sound pressure level as the sound propagates from the source position to the listener position.

The effect of this "spreading loss roll-off" is governed by an exponential curve, which scales a sound's apparent power by the reciprocal of the sound's distance raised to an exponent. In a perfect model, this exponent is 1.0, but anechoic simulations may need some adjustment to sound "right."

The exponential factor for all sources may be examined and adjusted from its default value through the **`cre_define_medium`**`()` function.

Chapter 4

# CRE_TRON API Function Reference

## Data structures

### wavFt

The waveform structure and typedef are provided for the application developer to have detailed information on open wave forms, for the purpose of computing useful estimates such length of play. ***All members are read-only except wavFt::next***.

```
typedef struct wavFs {
  const char    *fname;     /* host soundfile filename       */
  void          *pSignal;   /* pointer to the signal buffers */
  struct wavFs  *next;      /* linked-list pointer for user  */
  void          *synch;     /* synchronize with this signal  */
  int           diskBased;  /* TRUE if file still open       */
  double        sampleRate; /* in Hz; assumes 44.1kHz        */
  short         numChannels,/* 1 = mono;  2 = stereo         */
                sampleSize, /* in bytes: 1=8-bit, 2=16-bit   */
                frameSize,  /* in bytes: samplesize*channels */
                waveId,     /* remote serial identifier      */
                sourceId;   /* value = -1, if not attached   */
  float         pitchFactor;/* pitch shift factor            */
  unsigned long
                numFrames,  /* total frames in file          */
                remFrames,  /* # remaining frames NOT loaded */
                selFrames,  /* length of signal selection    */
                startFrame; /* beginning of signal selection */
  unsigned long
                loopStart,  /* loop start, in samples        */
                loopEnd,    /* loop end, in samples          */
                loopCount;  /* loop count                    */
} wavFt;
```

MEMBERS:

| | |
|---|---|
| fname | host sound filename string with full path. |
| pSignal | pointer to waveform signal data (internal use only). |
| next | pointer to next waveform struct (for application use). |
| synch | pointer to waveform that this wave will synchronize with. |
| diskBased | Boolean; non-zero indicates that file is open. |
| sampleRate | sample rate of the sound signal in samples per second. |
| numChannels | number of signal channels; 1 = monoaural, 2 = stereo. Currently, only supports monoaural. |
| sampleSize | bytes per sample; 1 = 8-bit, 2 = 16-bit. Promotes 8-bit RIFF files to 16-bit playback. Non-RIFF files are assumed 16-bit. |
| frameSize | bytes per frame; frame = sampleSize * numChannels. |
| waveId | unique index assigned for remote client/server packets. |
| sourceId | index of associated source; ¯1 = unassigned. |
| pitchFactor | factor by which wave gets pitch shifted >1 indicates upward shift, <1 indicates lowering of the pitch |
| numFrames | total frames in the signal data. |
| remFrames | unloaded frames remaining on disk. |
| loopStart | starting point for looping in samples. |
| loopEnd | ending point for looping in samples. |
| loopCount | number of times through the loop. |

Currently, the following members are not being used:

| | |
|---|---|
| selFrames | length of selected signal. |
| startFrame | index of beginning frame in selection. |

## cre_init

### Synopsis

```
#include "cre_tron.h"
int cre_init
   (int driver, int head, int sources, int mode);
```

### Description

Computes, detects, and allocates resources (i.e., processors and host memory) to provide the services specified by *driver* to listener *head* for the requested number of *sources*. The *driver* is a legacy CRE term, which specified the DSP binary to load from the host to the resourced hardware. In the AuSIM implementation, all old anechoic CRE drivers are honored, with a competent emulation. New AuSIM drivers initialize a specific set of simulation parameters.

All host objects are initialized with reasonable values. The listener's head is located at the origin. All sound sources are initially positioned at the full RESPONSE_DISTANCE (radius of actual HRTF responses; defined in ATRON.H) directly in front of the listener.

### Parameters

*Driver* driver selected from ATRNdriver enum defined in ATRON.H. Supported drivers are:

Atrn_CMP1

Atrn_BMP1

Atrn_BMP3

Atrn_BMP4

Atrn_ASM1

Atrn_A441

**Note:** Using any outdated driver values will cause **cre_init()** to fail.

*Head* — listener identifier to be initialized. An identifier must be unique for each listener, numbered sequentially from 0 to 63. If the listener is also a sound source to other listeners, the _ORATOR_ flag may be OR'ed with the listener ID. The _ORATOR_ flag forces the source with the same ID as the given listener ID not to be mixed for that listener. If the listener identifier has been previously initialized, the requested sources are additional.

*Sources* — maximum number of sources to be heard by the given listener.

*Mode* — bit field OR'ed from ATRON.H macros and enums:

units—defined in the ATRNunitsDef enum. Select *one* from:

AtrnINCHES

Atrn_FEET_

AtrnMMETER

AtrnCMETER

Atrn_METER

mode flags:

_VERBOSE_ – set verbose messages to be displayed

**Return Value**

| | |
|---|---|
| On success | returns the number of sources allocated. |
| On failure | `Error0` - no sources requested, no hardware available, or Trons already initialized. |
| | `Error1` – memory allocation error or load failure. |
| | `Error3` - acoustic headmap invalid or load failure. |
| | `Error4` - invalid *`driver`* type. |
| | `Error5` - invalid listener identifier. |
| | `Error6` - server response error. |

**Example**

```
if (cre_init(Atrn_ASM1, 0, 2, VERBOSE) < Ok)
  abort();
```

**Remarks**

If all localization resources have been allocated, subsequent calls to **`cre_init`**`()` fail until a **`cre_close`**`()` is called to free resources. Since all listeners share the same sound space, the number of sources requested for each head should be consistent.

**Important:** In the CRE implementation, all gains were initially set by **`cre_init`**`()` to `GAIN_dB_OFF`, so that active analog inputs at full level will not "pop" on without user control. *The AuSIM implementation does not turn the gain off*, but rather initializes each source with no signal channel assignment. To hear sound, the programmer *MUST* assign a signal channel with **`cre_define_source()`**, **`cre_select_source()`**, or **`cre_ctrl_wave()`**. Before connecting a signal channel, the programmer may choose to make use of the **`cre_amplfy_source`**`()` function to control the initial source amplitude.

**Note:** The fixed-point distance resolution within the RS-232 ATRON protocol is in hundreths of units. If the base units are set to `AtrnINCHES`, then the smallest movement would be 0.01".

## cre_update_audio

### Synopsis

```
#include "cre_tron.h"
int cre_update_audio (void);
```

### Description

Synchronizes frames and controls signal processing. This routine checks for any pending updates since the previous call, recomputes signal processing parameters with respect to all affected source-to-listener relationships, and passes the new values to the signal processing engine.

### Parameters

None

### Return Value

| | |
|---|---|
| On success | `Ok`, even if no changes were pending. |
| On failure | `Error0` - no sources have been initialized. |
| | `Error1` - system could not be interrupted to perform an update. |

### Example

```
cre_update_audio();
```

### Remarks

**`cre_update_audio`**`()` should be called *once* every time you want the audio updated. **`cre_update_audio`**`()` need not be called more frequently than the maximum framerate of the GoldServe™ (see **`cre_detect()`** to query framerate). Redundant calls are ignored.

**Synchronization Note:** In order to maintain synchronization of audio processing in a system in which more than one object (listener and/or sources) are moving simultaneously, you should complete all necessary calls to **`cre_locate_head`**`()` and **`cre_locate_source`**`()` *before* calling **`cre_update_audio`**`()`. Because the Tron's internal audio parameters can be updated only as fast as the specific Tron's update rate, the localization process may ignore more frequent calls to this routine.

## cre_close

### Synopsis

```
#include "cre_tron.h"
int cre_close (int driver, int head);
```

### Description

Deallocates host resources for a given driver and listener (which may then be reallocated with another **cre_init**() call). **cre_close**() will gently shut off audio for the specified listener When the last open listener is closed, all open wave files are also closed. Calling **cre_close**(ALL_DRIVERS, ALL_HEADS) will close all audio and wave files. **cre_close**() is required to safely terminate a host application.

### Parameters

| | |
|---|---|
| *driver* | selected from enumeration list in ATRON.H (see **cre_init**()). The macro ALL_DRIVERS shuts down all CRE drivers. |
| *head* | the identifier of an initialized listener to be closed. The macro ALL_HEADS will force all listeners associated with the given driver to be closed. |

### Return Value

| | |
|---|---|
| On success | Ok |
| On failure | Error0 - no Tron sources have been initialized. |
| | Error1 - invalid driver type. |
| | Error2 – uninitialized listener identifier *head*. |

### Example

```
cre_close(ALL_DRIVERS, ALL_HEADS);
```

## cre_end

### Synopsis

```
#include "cre_tron.h"
void cre_end (void);
```

### Description

This is an alias for **cre_close**(ALL_DRIVERS, ALL_HEADS) and is compatible with the ANSI C **atexit**() function.

### Parameters

None

### Return Value

None

### Example

```
atexit(cre_end);
```

## cre_detect

### Synopsis

```
#include "cre_tron.h"
int cre_detect (int prm);
```

### Description

This is an undocumented CRE function that has been formalized and extended in the AuSIM implementation to retrieve system parameter values of interest to the application programmer.

### Parameters

| | |
|---|---|
| *prm* | one of the pre-defined parameter values from `ATRNdetectDef` enum. Values can be one of the following list: |

`AtrnLSTNRinit`

`AtrnASRCinit`

`AtrnFRAMErate`

`AtrnSYSunits`

`AtrnWAVBUFavail`

### Return Value

| | |
|---|---|
| On success | non-negative value detected. |
| On failure | `Error1` - invalid parameter type. |
| | `Error2` - unable to detect type. |

### Example

```
cre_detect(AtrnQRYframerate);
```

## cre_define_head

### Synopsis

```
#include "cre_tron.h"
int cre_define_head
   (int id, int prm, int pts, const void *data);
```

### Description

Allows the user to specify parameters defining the listener model (head size and pinnae characteristics) and the reference frame for location coordinates in subsequent calls to **`cre_locate_head`**`()`.

### Parameters

*`id`* — the identifier of an initialized listener to be defined. The macro `ALL_HEADS` is *not* supported.

*`prm`* — one of the pre-defined parameter values from `ATRNheadDef` enum. Values can be one of the following list:

`AtrnAURALocular`

`AtrnAURALpinnae`

`AtrnAURALcrown`

`AtrnAURALoffsets`

`AtrnINTERAURAL`

`AtrnHRTFfile`

`AtrnAHMname`

`AtrnDELAYtable`

`AtrnDELAYscale`

`AtrnHRTFmodel`

`AtrnHEADgain`

`AtrnDISPLAYtype`

`AtrnEQleft`

`AtrnEQright`

| | |
|---|---|
| *pts* | number of data points given in ***data***. A negative value may be used by the particular parameter. |
| *data* | pointer to data of type specific to each parameter. |

`AtrnAURALocular`

AURAL OFFSET along X axis. Typically zero, or an offset from ocular axis (eye) coordinates. An offset from the ocular axis, which is in front of the aural axis, would be negative.

`AtrnAURALpinnae`

AURAL OFFSET along Y axis. One-half the positive distance between ear canal openings.

`AtrnAURALcrown`

AURAL OFFSET along Z axis. Typically, either the vertical separation of ocular and aural axes, or the vertical offset to the head tracking sensor, which is often placed on top of the head. An offset from the head crown, which is above the aural axis, would be negative.

The single value AURAL OFFSET parameter settings *must* have *either* a positive value ***`pts`*** ( = 1) and one float pointed to by ***`data`*** to set a new value, *or* ***`pts`*** = 0 to reset the default value in current units (***`data`*** will be ignored). These parameters locate the pinnae with respect to the head location given in each call to **`cre_locate_head`**`()` along the right-handed head coordinate axes, of which the positive X-axis extends out in front of the listener. The values must be in the current units, as set by **`cre_init`**`()`.

Setting to `AtrnAURALpinnae` affects only the spatial relationship of sound sources and receivers and *not* the temporal delays. (See `AtrnINTERAURAL` below for temporal control.) The head location to the head center vector direction determines the sign of the `AtrnAURALocular` and `AtrnAURALcrown` offsets. Negative values for `AtrnAURALpinnae` may yield unpredictable results.

Default offsets are zero, except for `AtrnAURALpinnae`, which is one-half the interaural separation (in current units) specific to the loaded listener map.

`AtrnAURALoffsets`

The set of ordered AURAL OFFSETs. With a single call to **`cre_define_head`**`()`, this parameter can update one to all AURAL OFFSETs as an ordered array of floats pointed to by ***`data`***, of ***`pts`*** items. The ordered sextuple is specified by the enumeration

`ATRNspaceDef` (*x,y,z,yaw,pitch,roll*). Currently, only Cartesian translations (*x,y,z*) are supported. All of the single value AURAL OFFSET parameter rules stated above apply to `AtrnAURALoffsets`, including ***`pts`*** = 0 to reset the default values.

`AtrnAHMname`

Specify the name of an AHM subject to load. If it is not one of the pre-loaded subjects, the system will search the AHM files (located in the directory defined by the HRTF environment variable) for the specified AHM subject. An error is returned if the AHM subject is not found.

`AtrnINTERAURAL`

The `AtrnAURALpinnae` OFFSET doubled and time scaled. This parameter combines the spatial control of the `AtrnAURALpinnae` parameter above with its associated interaural delay scaling. `AtrnINTERAURAL` is the full ear to ear width (measured to ear canal opening) and thus is twice the `AtrnAURALpinnae` value. ***`pts`*** = 1 will set the pinnae offset to one-half the absolute float value pointed to by ***`data`***. If ***`pts`*** = 0, ***`data`*** is ignored and the parameter is reset to the default value. In both cases, interaural delay values are not scaled to the ratio of the given interaural size with the default size. If ***`pts`*** = ¯1, the parameter is set according to ***`data`***, but the delay scaling is *not* altered.

`AtrnHRTFfile`

Specify HRTF filename to be loaded. If ***`pts`*** = 0, ***`data`*** is ignored and the default HRTF map is reloaded. Otherwise the filename pointed to by ***`data`*** is loaded from the directory given by the HRTF environment variable. The ***`pts`*** value should be the string length of the filename for consistency.

`AtrnDELAYtable`

Redefine the interaural delay table. This parameter requires a properly formatted interaural delay table. This parameter is provided for psychoacoustic research, and is otherwise undocumented.

`AtrnDELAYscale`

Scale the current delay table. When ***`pts`*** = 1, this will scale all interaural delays by the float value pointed to by ***`data`***. If ***`pts`*** = 0, ***`data`*** is ignored and the interaural delays are reset to their

default values.

`AtrnHRTFresolve`

Sets the filter order trim.

`AtrnHRTFmodel`

Sets the model to use. This will be one of: `NearField`, `FarField`, or `MixedField`.

`AtrnHEADgain`

`AtrnHEADgain` sets a single floating point dB level for the final conversion gain in preparing a listener's signal pair for output. The *`pts`* parameter is ignored.

`AtrnDISPLAYtype`

`AtrnDISPLAYtype` sets the output type for filtering per display device. The enum describing the particular device is given in the *`pts`* parameter. Choose one of the following: `eqGenericHeadphone`, `eqGenericNearphone`, `eqSennheiserHD250` or `eqSennheiserHD570`. *`data`* is ignored.

`AtrnEQleft`

Downloads the display type for left EQ coefficients.

`AtrnEQright`

Downloads the display type for right EQ coefficients.

**Return Value**

| | | |
|---|---|---|
| On success | `Ok` | |
| On failure | `Error0` - | no Trons have been initialized. |
| | `Error1` - | invalid parameter *`prm`*. |
| | `Error2` - | *`pts`* > 0, but *`data`* is NULL. |
| | `Error3` - | AHM is invalid, or load failure. |
| | `Error4` - | uninitialized listener identifier *`id`*. |

**Example**

```
/* setting offset for a tracking device on top of
the head */
const float offsets[3] = { 0.0, INTERAURAL,
CROWN_OFFSET };
cre_define_head(2, AtrnAURALoffsets, 3, offsets);
```

## cre_locate_head

### Synopsis

```
#include "cre_tron.h"
int cre_locate_head (int id, const float
*headLoc);
```

### Description

Locates the head of a listener six dimensionally in world coordinates. It only updates changes from previous state, recalculating pinnae locations as needed. *This function does not affect processing until a synchronization call to* **`cre_update_audio`**`()` *is successful.*

### Parameters

*`id`* the identifier of a listener to be defined.

*`HeadLoc`* a pointer to an ordered array of six floats as follows:

- `AtrnX` world *x*-axis coordinate.
- `AtrnY` world *y*-axis coordinate.
- `AtrnZ` world *z*-axis coordinate.
- `AtrnYAW` angle of -π to π from the world *x*-axis about the world *z*-axis of the projection of the head's *x*-axis onto the world *x*-*y* plane. Looking down at the *x*-*y* plane a counter-clockwise rotation is positive.
- `AtrnPTC` angle of -π/2 to π/2 from the world *x*-*y* plane of the head's *x*-axis about the world *y*-axis. *Remember* that with *x* forward and *z* up, a positive pitch is down.
- `AtrnROL` angle of -π to π from the world *y*-axis about the world *x*-axis of the head's *y*-axis. From the listener's point of view, a clockwise roll of the head, rolls *y* into *z* and is

therefore positive.

**Return Value**

| | | |
|---|---|---|
| On success | `Ok` | |
| On failure | `Error0` - | no Trons have been initialized. |
| | `Error1` - | *`headLoc`* is NULL. |
| | `Error2` - | uninitialized listener identifier *`id`*. |

**Example**

```
const float headLoc[6] = {10.0, 20.0, 30.0, 0.0,
0.0, 0.0};
cre_locate_head(2, headLoc);
```

## cre_define_source

### Synopsis

```
#include "cre_tron.h"
int cre_define_source
      (int id, int prm, int pts, const void
      *data);
```

### Description

Allows the user to specify parameters defining the source rendering model (directional radiation pattern, localization ON/OFF, and listener linkage). The function is a generic dispatcher that may be extended in future releases. See parameter descriptions below for specific behavior.

### Parameters

*id* the zero-based index of the audio source in reference. The macro `ALL_SOURCES` is supported.

*prm* one of the pre-defined parameter values from the `ATRNsrcDef` enumeration. This can be one of the following values:

`AtrnRADfields`

`AtrnRADprofile`

`AtrnPROFILEpts`

`AtrnSPATIALoff`

`AtrnSPATIALon`

`AtrnHEADlink`

`AtrnHEADunlink`

`AtrnSPRDrolloff`

`AtrnGAINdist`

`AtrnCHNLinput`

`AtrnCHNLmidi`

`AtrnDPLRfactor`

| | |
|---|---|
| *pts* | the number of points to be read from the ***data*** pointer. |
| *data* | base pointer of an array that has at least ***pts*** elements. An undetectable error will occur if ***pts*** is larger than the number of elements defined in ***data[]***. The pointer can be NULL, in which case the first ***pts*** points of the existing pattern table will be used. |

`AtrnRADfields`

Allows the user to control the directivity of the sound. *`data[]`* will contain two parameters (both in radians) describing a field of radiation and a field of intensity. These fields are cones centered on the source's boresight direction (principal direction of aural emission) in which ~90% (for the field of radiation) or ~45% (for the field of intensity) of the sound energy is dissipated.

`AtrnRADprofile`

Defines an audio source's radiation pattern about its boresight axis. The radiation profile of a sound source *`id`* is specified in *`data`* with an array of relative sound pressure levels in decibels, sampled equiangularly at *`pts`* points from the boresight direction to anti-boresight direction, inclusive. The boresight direction is coincident with the source's positive roll axis (the axis parallel to the world coordinate X-axis when source yaw and pitch are zero). All definable radiation patterns are symmetric about the roll axis (i.e., no rectangular horn speakers). For off-axis angles from source to listener that fall between sample points, the profile is linearly interpolated. Designed for empirically sampled data, `AtrnRADprofile` also provides effective profile definition with very few artificial points. The macro `MAX_RADPROFILE` specifies the maximum *`data`* array size supported. See Figure 2 for examples of using `AtrnRADprofile`.

**Note:** Uniformly radiating sources save considerable computation and are specially defined by *`pts`* = 0. In the special case of *`pts`* = 1, the boresight is defined by `(*data + GAIN_dB_OFF)/2`. If a NULL *`data`* pointer is given, the profile is set to the first *`pts`* points of the existing table. With this feature you can pre-load the profile array, and then toggle directional radiation on and off with *`pts`*.

`AtrnPROFILEpts`

Specifies the number of profile points to use.

`AtrnSPATIALoff`

Disables localization for this source. The monaural sound is patched directly through and mixed with the left and right outputs with a gain (in dB) defined by *`data`*. This allows the user to implement simple panning. If *`pts`* = 0, the previous panning mixture values are used. If *`pts`* = 1, the given value is applied to a

balanced mixture. If *pts* > 2, an error is returned. If *pts* < 0, localization is re-enabled, which is the default. The source is passed through a flat filter, so the latency remains the same as if it were localized.

`AtrnSPATIALon`

Enables source localization. This is the default.

`AtrnHEADlink`

By setting this parameter, the source maintains its relative position and orientation to the listener's head for all head positions and attitudes given. When `AtrnHEADlink` is enabled, all calls to **`cre_locate_source`**`()` establish a new relative position as a difference of the global coordinates of head and source locations given. *`data`* is ignored. The *`pts`* argument is interpreted as the head index. *`pts`* < 0 unlinks the sound to all listener head positions.

`AtrnHEADunlink`

Disables source to head linkage. This is the default.

`AtrnSPRDrolloff`

`AtrnSPRDrolloff` defines a single float value multiplier for that particular source of the global spreading-loss roll-off exponent defined in **`cre_define_medium`**`()`. The value must be positive and "reasonable". Reasonableness depends on the global rolloff value.

`AtrnGAINdist`

`AtrnGAINdist` defines the distance at which the gain is specified.

`AtrnCHNLinput`

`AtrnCHNLinput` allows zero, one, or more physical live audio input channels to be mapped to a particular source. *`pts`* specifies how many and *`data`* is the array of integers specifying which channels to map.

`AtrnCHNLmidi`

`AtrnCHNLmidi` allows zero, one, or more MIDI audio input channels to be mapped to a particular source. *`pts`* specifies how many and *`data`* is the array of integers specifying which MIDI channels to map.

`AtrnDPLRfactor`

`AtrnDPLRfactor` sets the Doppler exaggeration. *`pts`* always specifies the listener *`id`*, which may be `ALL_HEADS`. If *`data`* is NULL, the Doppler factor is reset to its default of 1.0. Setting Doppler factor to 0.0 disables it completely.

### Return Value

On success `Ok`

On failure `Error0` - audio source *`id`* is out of range, or no Tron sources have been initialized.

`Error1` - invalid parameter *`prm`*.

`Error2` - *`pts`* is out of range.

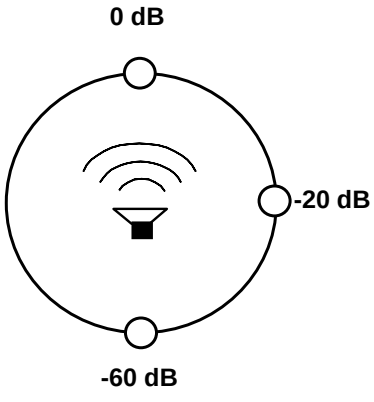
`Error3` - parameter specific error, see parameter descriptions

### Example

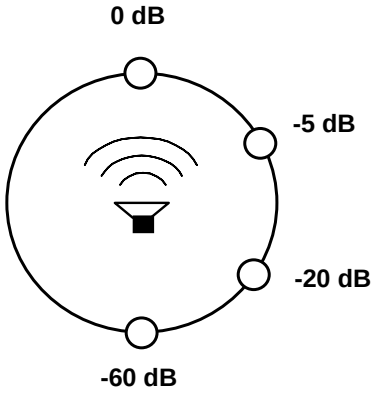
```
const float rad_table[4] =
      { 0.0, -5.0, -20.0, -60.0 };
cre_define_source (mysource, AtrnRADprofile, 4,
rad_table);
```

**Figure 4: `cre_define_source()` examples for two different radiation tables.**

```
rad_table[3] =
{0.0, -20.0, -60.0}
```




```
rad_table[4] =
{0.0, -5.0, -20.0, -60.0}
```



## cre_locate_source

### Synopsis

```
#include "cre_tron.h"
int cre_locate_source
     (int id, const float *sourceLoc);
```

### Description

Locates an audio source `id` in (*x, y, z, yaw, pitch, roll*) world coordinates. ***This function does not affect processing until a synchronization call to*** **`cre_update_audio`**`()` ***is successful.***

### Parameters

*id* — the zero-based index of the audio source to be located. The macro `ALL_SOURCES` is *not* supported. To locate multiple audio sources, multiples calls **`cre_locate_source`**`()` are necessary.

*SourceLoc* — a pointer to a sextuple of floats:

- `AtrnX` — world *x*-axis coordinate.
- `AtrnY` — world *y*-axis coordinate.
- `AtrnZ` — world *z*-axis coordinate.
- `AtrnYAW` — angle of -π to π from the world *x*-axis about the world *z*-axis of the projection of the source's *x*-axis onto the world *x*-*y* plane. Looking down at the *x*-*y* plane, a counter-clockwise rotation is positive.
- `AtrnPTC` — angle of -π/2 to π/2 from the world *x*-*y* plane of the source's *x*-axis about the world *y*-axis. *Remember* that with *x* forward and *z* up, a positive pitch is down.
- `AtrnROL` — angle of -π to π from the world *y*-axis about the world *x*-axis of the source's *y*-axis. From the source's point of view, a clockwise roll of the sound rolls *y* into *z* and is therefore positive.

### Return Value

On success — `Ok`

On failure — `Error0` - audio source *id* is out of range, or no Tron sources have been initialized.

`Error1` – *sourceLoc* is NULL.

**Example**

```
const float srcloc[6] =
      {-10.0, -20.0, 0.0, 0.0, 0.0, 0.0};
cre_locate_source (4, srcloc);
```

**Remarks**

Only the first three floats are used when the audio source is in uniform radiation mode. However, as a safe programming practice, you should always maintain pointers to six-element data structures. `AtrnROL` is not presently used since non-uniform radiation is symmetric about the boresight roll axis, but it should be maintained for future compatibility.

## cre_amplfy_source

### Synopsis

```
#include "cre_tron.h"
int cre_amplfy_source (int id, float dB);
```

### Description

Sets the loudness of a source to 0 dB at $2^{1/rollofexp}$ units from the listener. This distance is ~1.96 inches from the head if units of inches (default) are being used. To alter this distance, scale the base 2 by the macro `GAIN_RATIO`. Distant sound sources may need to be set much higher (as much as +30 dB), in order to be audible at the listener's position.

### Parameters

| | |
|---|---|
| *id* | the zero-based index of the sound source to be amplified. The macro `ALL_SOURCES` is supported. |
| *dB* | the amplification level in decibels. The macro `GAIN_dB_OFF` is provided to definitively turn off a sound source. Any value less than -120 dB is interpreted as off. |

### Return Value

| | |
|---|---|
| On success | `Ok` |
| On failure | `Error0` - audio source *id* is out of range, or no Tron sources have been initialized. |
| | `Error1` - *dB* is unreasonable. To prevent floating point overflows, *dB* should not exceed `20 * EXPONENT_LIMIT`, defined in ATRON.H. |

### Example

```
cre_amplfy_source(ALL_SOURCES, 1.0);
```

### Remarks

This is the most misunderstood function in the CRE_TRON API. Attenuation over distance is a very important 3D cue, over which the system must have dynamic range to apply. As a sound source gets closer to a receiver, its sound pressure level must increase exponentially (nominally 6 dB for every half of the distance), but there is a maximum volume that audio hardware can (and, for

safety reasons, should) reach. We have set the library so that the maximum volume is reached for a 0 dB at 2.5 inches from the receiver. If a source is within this range, our software can provide very little distance cue. If the source is mostly far-field (never comes near the receiver), you can optimize the dynamic range by setting the gain to a higher value. A table relating source amplitude setting to clipping distance may be found in ATRON.H.

If you need to adjust the relative amplitude of the source, it should be done at the synthesis of the sound. **`cre_amplfy_source`**`()` will provide such relative amplitude service, but you run the risk of ruining the 3D effect for near field sounds.

## cre_select_source

### Synopsis

```
#include "cre_tron.h"
int cre_select_source (int id, int channel);
```

### Description

Selects from among the available hardware analog input channels for a given source *id* for all listeners. The implementation of this function is hardware specific and the command may not be desirable globally. This function is not defined for all drivers, such as `Atrn_CMP1`, which uses all inputs available. This function may produce unexpected results when used with different drivers rendering the same audio source to different listeners.

This has the same effect as **`cre_define_source`**`(`*`id`*`, AtrnCHNLinput, 1, `*`channel`*`)`

### Parameters

| | |
|---|---|
| *id* | the zero-based index of the audio source to be patched. The macro `ALL_SOURCES` is supported. |
| *Channel* | the zero-based index of the analog input channel on the target hardware. |

### Return Value

| | |
|---|---|
| On success | `Ok` |
| On failure | `Error0` - audio source *id* is out of range or no Tron sources have been initialized.<br>`Error1` - invalid *channel*.<br>`Error2` - failure to execute patch. |

### Example

```
cre_select_source(0, 3);
```

## cre_pmeter_source

### Synopsis

```
#include "cre_tron.h"
int cre_pmeter_source (int id, float *power);
```

### Description

Measures the instantaneous power of the sound source. The measured value is written to the float that *power* points to.

### Parameters

| | |
|---|---|
| *id* | the zero-based index of the sound source to be amplified. The macro ALL_SOURCES is supported. |
| *power* | pointer to which the calculated power is written. |

### Return Value

| | |
|---|---|
| On success | Ok |
| On failure | Error0 - audio source *id* is out of range, or no Tron sources have been initialized. |

### Example

```
cre_pmeter_source(id, &power);
```

## cre_define_medium

### Synopsis

```
#include "cre_tron.h"
int cre_define_medium
        (int prm, int pts, const void *data);
```

### Description

Allows the user to specify parameters to model the medium through which the sound propagates (absorption filter distance and spreading roll-off exponent). The function is a generic dispatcher that may be extended in future releases. See parameter descriptions below for specific behavior.

### Parameters

*rm* — one of the pre-defined parameter values from the ATRNmedDef enumeration. This can be one of the following list:

AtrnROLLOFF

AtrnABSORBdist

AtrnMEDDEFlast

*pts* — the number of points to be read from the *data* pointer.

*data* — a pointer to at least *pts* data points. An undetectable error will occur if *pts* is larger than the number of points available to read. The pointer can be NULL, in which case the first *pts* points of the existing pattern table will be used.

### PRM Types

AtrnABSORBdist

Atmospheric absorption control distance. The absorption distance controls the amount of extra high frequency fall-off over distance that is applied to simulate atmospheric absorption. Currently, this parameter can only be set to apply to all sources. The *pts* argument must be 1 in order to have *data* set the distance in current units. *pts* = 0 or *pts* < 0 will reset the absorption

distance to its default value, defined by the macro ABSORPTION_DISTANCE (in ATRON.H). The effect of the absorption filter can be minimized by setting this value to an arbitrarily large distance. However, it can be disabled entirely by passing a value of 0.0 or less.

**Remarks**: The given distance affects the amount of atmospheric absorption filtering at a given source-to-receiver range by a factor of the range divided by the sum of the range and the absorption distance. Hence, the absorption filter will be applied at 50% when the range is equal to the absorption distance.

AtrnROLLOFF

The roll-off exponent due to spreading power loss. The spreading roll-off exponent parameter sets the rate at which sound amplitude is attenuated over distance to yield cues in the third dimension. Currently, this parameter can only be set to apply to all sources. The *pts* argument must be set to one to have the float value pointed to by *data* set the spreading roll-off exponent, or may be zero or negative to reset the default value, defined by the macro SPREADING_ROLLOFF in ATRON.H. An out of range exponent value will return an error. The exponent must be greater than zero and less than the value defined by the macro EXPONENT_LIMIT in ATRON.H.

**Remarks:** In a free sound field, spreading loss is -6 dB for every doubling of the distance (i.e., gain is proportional to 1.0 / R). However, there are few free sound fields in the real world, so the apparent spreading loss depends on the acoustic impedance of the propagation medium and elements in the sound field. Since the Tron is simulating a virtual anechoic environment, a nominal roll-off exponent of 1.0 sounds steep. Typically, roll-off exponents in between 0.5 and 1.2 are of interest.

**Return Value**

| | |
|---|---|
| On success | `Ok.` |
| On failure | `Error0` – no Tron sources have been initialized. |
| | `Error1` – invalid parameter *`prm`*. |
| | `Error2` – *`pts`* is non-zero, but *`data`* is NULL. |

**Example**

```
/* set distance in current units */
float absorb_dist = 100.0;
cre_define_medium
      (AtrnABSORBdist, 1, &absorb_dist);
```

## cre_open_wave

### Synopsis

```
#include "cre_wave.h"
wavFt *cre_open_wave
        (const char *wavefile, int mode);
```

### Description

Opens a sound file referred to by the filename *wavefile* from the GoldServe™'s disk, returning a pointer to the allocated wavefile structure `wavFt`. Sound file control, such as playback through a particular source, is effected through **`cre_ctrl_wave`**`()`. Currently, the only formally recognized sound file format is RIFF (MS Windows .WAV format). Note that, independent of which driver is being used, both 22.05 kHz and 44.1 kHz wave files can be opened and played back.

**Note:** All calls to **`cre_open_wave`**`()` should precede any call to **`cre_init`**`()` and every **`cre_open_wave`**`()` should be paired with a corresponding **`cre_close_wave`**`()`. Wave file open-close pairs are independent of but must enclose **`cre_init`**`()`-**`cre_close`**`()` pairs. The maximum number of concurrently open wave files is determined by the amount of system memory available (see remarks below).

### Parameters

| | |
|---|---|
| *Wavefile* | a string which specifies the filename to be loaded from disk. If the filename extension is '`.WAV`', the file must have a valid RIFF format header. |
| *mode* | ***This parameter is no longer needed and is ignored.*** |

### Return Value

| | |
|---|---|
| On success | `wavFt*` - the returned structure will be empty. See **`cre_ctrl_wave`**`(s, wave, WaveCTRL FSTAT, NULL)` command to filled the `wavFt` structure. |
| On failure | `NULL` - filename *wavefile* not found, could not be opened, was invalid, or system out of memory. |

### Example

```
char *fname = "test.wav";
wavFt *wave = cre_open_wave(fname, NULL);
```

```
if (wave == NULL)
  printf("%s failed to open.\n",fname);
```

### Remarks

All wave files are loaded into memory. The maximum number of concurrently open wave files is determined by the amount of memory available in the GoldServe™ system. 16-bit wave files will require twice the amount of system memory as disk space. Because loading the data from the hard disk into memory takes some time (about 50 milliseconds per 100 kB of file space), all **cre_open_wave**() calls must take place *before* any calls to **cre_init**().

## cre_ctrl_wave

### Synopsis

```
#include "cre_wave.h"
int cre_ctrl_wave
      (int src, wavFt *wave, int cmd, void *data);
```

### Description

Requests the host to control the waveform *wave* according to the command *cmd,* which may be related to source *src*. This function is a generic dispatcher that may be extended in future releases. See command descriptions below for specific behavior.

### Parameters

*src* — the zero-based index of the audio source in reference. The macro ALL_SOURCES is *not* supported.

*wave* — a pointer to the waveform structure affected by all commands except WaveCTRL_STOP.

*cmd* — one of the pre-defined command values from the wave_ctrl enumeration. This must be one of the following:

WaveCTRL_RFRS

WaveCTRL_PSET

WaveCTRL_RWND

WaveCTRL_NOLP

WaveCTRL_STRT

WaveCTRL_PLAY

WaveCTRL_LOOP

WaveCTRL_STOP

WaveCTRL_STAT

WaveCTRL_SYNC

WaveCTRL_PTCH

WaveCTRL_LPST

| | |
|---|---|
| *data* | NULL, except for `WaveCTRL_PTCH` and `WaveCTRL_LPST` |

**CMD Types** `WaveCTRL_RFRS`

Refreshes the wave file image from disk.

`WaveCTRL_PSET`

Sets the signal pointer to the specified sample. *`data`* will hold the zero-based index of the sample relative to the base sample.

`WaveCTRL_RWND`

Rewinds the current frame position of waveform to its first frame. This will work even while playing. Rewind is useful if the waveform was stopped before finishing its full selection.

`WaveCTRL_NOLP`

Unsets the loop flag, allowing the source to play to its end and stop.

`WaveCTRL_STRT`

Plays the waveform from its beginning (rewinding if necessary), patching it through to source *`src`*.

`WaveCTRL_PLAY`

Plays the waveform starting at the current frame position, patching it through to source *`src`*. By default, playing does not loop.

`WaveCTRL_LOOP`

Turns on looping and starts playing the waveform from its current frame position. Turning on looping means that when playback reaches the end of the sound file, the signal is automatically rewound to its beginning. Looping continues, until the playback is either stopped (`WaveCTRL_STOP`), or the loop flag becomes disabled (`WaveCTRL_NOLP`). See `WaveCTRL_LPST` for information on setting loop points.

`WaveCTRL_STOP`

Stops playing any waveform attached to source *`src`*, maintaining that waveform's current frame position.

`WaveCTRL_STAT`

Tests the current waveform wave status with respect to given source *`src`*. Alternatively, this command can check wave

status for any source with *src* = ¯1, or check *src* for any waveform with *wave* = NULL. Returns a 4-bit value, with each bit representing a state of either *wave* or *src*:

bit 0 - *wave* is playing on source *src*

bit 1 - *wave* playing on some other source.

bit 2 - *src* is playing the *wave*.

bit 3 - *src* is playing some other waveform.

A return of zero means that neither *wave* nor *src* are busy.

WaveCTRL_SYNC

Specifies an already playing wavefile with which to synchronize. This synchronization will take effect on a subsequent call to **cre_update_audio**(). Setting *data* = NULL will turn off synchronization.

Example of WaveCTRL_SYNC usage:

```
cre_ctrl_wave(src1, wave1, waveCTRL_SYNC, wave0);
cre_ctrl_wave(src0, wave0, waveCTRL_PLAY, NULL);
cre_ctrl_wave(src1, wave1, waveCTRL_PLAY, NULL);
```

WaveCTRL_PTCH

Sets the pitch shift factor pointed to by *data* (float *) for wavefile *wave* (BMP2 and BMP3 drivers only). A value of 1.0 (default) results in no pitch shifting, a value of 2.0 (maximum) will double the pitch of the wavefile, a value of 0.5 (minimum) will half the pitch of the wavefile. *data* = NULL will disable pitch shifting.

WaveCTRL_LPST

Sets loop start, loop end, and loop count parameters. *src* is ignored, *data* is a pointer to an array of three longs:

*data*[0] = loop start (unsigned long, in samples)

*data*[1] = loop end (unsigned long, in samples)

*data*[2] = loop count (signed long, ¯1 = infinte looping)

**Return Value**

| | |
|---|---|
| On success | `Ok`, or bit code (see `WaveCTRL_STAT`) |
| On failure | `Error0` - audio source *`src`* is out of range, or no Tron sources have been initialized. |
| | `Error1` - waveform structure pointed to by *`wave`* is invalid. |
| | `Error2` - command *`cmd`* is unsupported. |
| | `Error3` - *`wave`* already playing on another source |
| | `Error4` - failed attempt to perform |

**Example**

```
wavFt *wavep = cre_open_wave("TEST.WAV", 4);
cre_ctrl_wave(0, wavep, WaveCTRL_LOOP, NULL);
…       /* do other things while sound plays */
cre_update_audio(); /* tend to playback buffers */
…       /* do other things while sound plays */
cre_ctrl_wave(0, wavep, WaveCTRL_STOP, NULL);
cre_close_wave(wavep);
```

## cre_close_wave

### Synopsis

```
#include "cre_wave.h"
int cre_close_wave (wavFt *wave);
```

### Description

Closes the wavefile, if open, and frees the signal and the wave structure. If *wave* is attached to a sound source and is playing, it will be stopped before the wave file is closed. In order to properly deallocate resources, each (successful) call to **cre_open_wave**() must be balanced with a call to this routine.

### Parameters

| | |
|---|---|
| *wave* | a pointer to the waveform structure to be closed. |

### Return Value

| | |
|---|---|
| On success | Ok |
| On failure | Error1 - invalid *wave* structure. |

### Example

```
wavFt *wavep = cre_open_wave("TEST.WAV",4);
…            /* listen to the music */
cre_close_wave(wavep);
```

## cre_send_midi

### Synopsis

```
#include "cre_midi.h"
int cre_send_midi
      (int src, const unsigned char *midistr);
```

### Description

Sends a string of MIDI commands *midistr* (terminated by MIDI_MsgTerm* as defined in CRE_MIDI.H) to the MIDI port on the Tron card that is responsible for source *src*. A well constructed MIDI string is assumed. See CRE_MIDI.H for some pre-defined MIDI strings.

### Parameters

| | |
|---|---|
| *src* | the zero-based index of the audio source in reference. The macro ALL_SOURCES is supported (same message is sent once to each port, i.e., per *pair* of sources). |
| *midistr* | a character string (of any length) of MIDI commands, terminated by MIDI_MsgTerm. |

### Return Value

| | |
|---|---|
| On success | Ok |
| On failure | Error0 - audio source *src* is out of range or no Tron sources have been initialized. |
| | Error1 - MIDI string *midistr* is invalid. |

### Example

```
MIDIbyte MyFavoriteNote[] =
      {0x90, 0x3C, 0x40, MIDI_MsgTerm };
cre_send_midi(1, MyFavoriteNote);
```

* The message terminator MIDI_MsgTerm is *not* sent to the port.

## cre_set_midi

### Synopsis

```
#include "cre_midi.h"
int cre_set_midi
      (int src, int cmd, void *data);
```

### Description

Provides necessary services for the Proteus synthesizer on a Beachtron card, which is associated with source *src*. The function is a generic dispatcher that may be extended in future releases. See command descriptions below for specific behavior.

### Parameters

*src* — the zero-based index of the audio source in reference. The macro ALL_SOURCES is supported. In this mode, the TRON MIDI messages are received.

*cmd* — an integer which specifies one of four Proteus commands:

PROTEUS_RESET

to reset the Proteus synthesizer to boot-up condition.

PROTEUS_INTERNAL

to select internal MIDI string input. In this mode, the TRON MIDI messages are received.

PROTEUS_EXTERNAL

to select external MIDI string input. In this mode, the synthesizer only receives messages through its external port and ignores messages passed by TRON MIDI functions.

PROTEUS_LOAD_PRESET_FILE

to map a preset file specified by the NULL terminated character string filename pointed to by *data*. Returns Error3 if the preset file

does not exist or is invalid, or if the load fails.

| | |
|---|---|
| *data* | (optional) Set to NULL, except when used with PROTEUS_LOAD_PRESET_FILE, where it is set to a null-terminated character string. |

### Return Value

| | |
|---|---|
| On success | Ok |
| On failure | Error0 - audio source *src* is out of range or no Tron sources have been initialized. |
| | Error1 - invalid command *cmd*. |
| | Error2 - command *cmd* requires data, but *data* is NULL. |
| | Error3 - command specific error. See descriptions above. |

### Example

```
cre_set_midi(2, PROTEUS_RESET, NULL);
```

## cre_msg_midi

### Synopsis

```
#include "cre_midi.h"
int cre_msg_midi
      (int src, int msg, int chnl, int data1,
       int data2);
```

### Description

Constructs and sends a MIDI channel voice message to the proper Proteus synthesizer channel or MIDI port associated with source *src*, described by message type *msg*, on Tron MIDI channel *chnl*, and with data arguments *data1* and *data2*.

A MIDI channel voice message consists of an 8-bit status byte including the message type and target MIDI channel, followed by one or two 7-bit data bytes. **cre_msg_midi**() forms common MIDI strings for targeting synthesized sounds to localized Tron sources. The function properly bit-clips all parameters, and then assembles a MIDI status byte via the following formula:

```
status_byte = (0x80 | ((msg&0x07) << 4)
   | ((chnl&0x07) + 8*(id%BMP1_SRCS)))
```

Finally, it appends the correct number of 7-bit clipped data bytes, according to the table below, and sends the resulting message string to the Proteus.

| **Message Macro** *msg* | **Data Byte 1** *data1* | **Data Byte 2** *data2* |
|---|---|---|
| MIDI_NOTE_OFF | *key number* | *velocity* |
| MIDI_NOTE_ON | *key number* | *velocity* |
| MIDI_AFTERTOUCH | *key number* | *pressure amount* |
| MIDI_CONTROL | *control number* | *control value* |
| MIDI_PATCH | *program number* | *(none)* |

| | | |
|---|---|---|
| MIDI_CHNLPRESS | *pressure value* | *(none)* |
| MIDI_PITCHBEND | *pitch bend* | *pitch change* |

Key numbers can be easily derived using the `OCTAVE` and `NOTE` enumeration lists defined in CRE_MIDI.H and the following formula:

```
key_number = (octave * 12) + note
```

---

**Parameters**

| | |
|---|---|
| *src* | the zero-based index of the audio source in reference. The macro `ALL_SOURCES` is supported. |
| *msg* | an integer that specifies a MIDI channel voice message (see above table for valid messages). |
| *chnl* | the MIDI channel to which the command is to be applied |
| *data1* | first data argument for command |
| *data2* | second data argument (if necessary) for command. |

**Return Value**

| | |
|---|---|
| On success | `Ok` |
| On failure | `Error0` - audio source *src* is out of range or no Tron sources have been initialized. |
| | `Error1` - invalid MIDI message. |
| | `Error2` - *chnl* out of range. |

**Example**

```
cre_msg_midi(mysrc, MIDI_NOTE_ON, 0,
   (octave*12)+note, velocity);
```

**Remarks** While this function is designed to assist assembling simple MIDI messages without a MIDI sequencer, we recommend that the inexperienced MIDI programmer invest in a good MIDI reference to fully understand the appropriate use of the interface.

## cre_set_rel_pos

### Synopsis

```
#include "cre_tron.h"
int cre_set_rel_pos
      (int head, int src, float azim, float elev,
       float gain);
```

### Description

Locates sound source *src* relative to listener *head*. If the sound source is imagable (i.e., it is processed relative to its environment as in the Acoustic Room Simulation) and the rendering model has been set to be near-field, the relative coordinates will be converted to world coordinates (*x,y,z*). Otherwise, the sound will be processed directly in polar coordinates, short circuiting much processing overhead.

**Note:** This function does not affect a sound source's directional boresight or its current status with respect to other listeners.

### Parameters

| | |
|---|---|
| *head* | the identifier of an initialized listener from which the source is relatively located. The macro ALL_HEADS is supported. |
| *src* | the zero-based index of the audio source to be amplified. The macro ALL_SOURCES is *not* supported. To locate multiple audio sources, this function must be applied for each sound source. |
| *azim* | counter-clockwise rotation (in radians) from straight ahead of the source's projection onto the head's median plane. |
| *elev* | angular elevation (in radians) from the head's median plane to the source. Note that, *as opposed to pitch*, an elevation is positive above the plane and negative below it. |
| *gain* | linear amplitude from 0.0 to 1.0, for the mixed-field and far-field models. No range is computed, hence acoustic rendering effects such as reflected source |

images, Doppler pitch shift, and atmospheric absorption will be interpreted as though the source were located at the head's center. For near-field models, *gain* is interpreted as a range greater than 1.0 in current units with the rendering effects including *gain* computed normally.

**Return Value**

| | |
|---|---|
| On success | `Ok` |
| On failure | `Error0` - audio source *src* is out of range |
| | `Error1` - *gain* argument out of range for current rendering model. |
| | `Error2` - uninitialized listener identifier *head* |

**Example**

```
cre_set_rel_pos(0, src, 0.0, 0.0, 1.0);
```

## cre_get_polar

### Synopsis

```
#include "cre_tron.h"
struct {float azim, elev, rang; } polar;
int cre_get_polar
          (int head, int src, float *polar);
```

### Description

Retrieves the relative position in polar coordinates of sound source *src* with respect to listener *head*. If the sound source was relatively positioned, either the rendering system is imaging reflections or the rendering model is far-field. The information should have been computed in the most recent update but, if not, it will be computed from global coordinates. The retrieved polar coordinates are written to the *polar* structure.

### Parameters

| | |
|---|---|
| *head* | the identifier of an initialized listener from which the source is relatively located. The macro ALL_HEADS is *not* supported. |
| *src* | the zero-based index of the audio source to which the bearing is requested. The macro ALL_SOURCES is *not* applicable. |
| *polar* | a pointer to storage for three floats. Upon success, three floats will be copied into the storage pointed to by *polar*: azimuth, elevation (in radians), and range (in current units). |

### Return Value

| | | |
|---|---|---|
| On success | Ok | |
| On failure | Error0 - | audio source *src* is out of range |
| | Error1 - | *polar* is NULL |
| | Error2 - | uninitialized listener identifier *head* |

### Example

```
struct { float azim, elev, rang; } polar;
cre_get_polar(0, src, &polar);
```

**Remarks**

s function is provided for utility only.

## cre_test_atron

### Synopsis

```
#include "atron.h"
int cre_test_atron (int verbose);
```

### Description

Tests the connection to the localization server via the ATRON protocol and gathers a text string from the server. If *verbose* is non-zero, it will print this string to stdout.

**Important:** This function runs verbosely from a console window.

### Parameters

| | |
|---|---|
| *verbose* | verbose flag |

### Return Value

| | |
|---|---|
| On success | Returns the alphabetical index of the first letter of this string, which is one of:<br>1 = Acoustetron<br>2 = Beachtron Server<br>3 = Convolvotron Server<br>4 = GoldServe Server<br>13 = Mtron. |
| On failure | `Error1` - a connection could not be made.<br>`Error2` – response from client was incomplete |

### Example

```
cre_test_atron(1)
```

## cre_fetch_message

```
synopsis      #include “cre_tron.h”
              int cre_fetch_message
                (int msg, char *buffer, int bufsize)
```

Description

Retrieves messages from the output buffer. Can be used to assist with debugging and to provide a means of checking for problems. The messages are copied into a buffer so that they can be read, printed to the screen etc.

Parameters

msg

If msg >=0 then the message which is retrieved is the one at the index msg from the END of the list of messages. The smaller the value of msg, the more recent the message. If msg < 0 then this function will retrieve as many messages as possible beginning with the most recent until the buffer is full.

buffer

A pointer to the location where the retrieved messages will be stored as a string.

bufsize

The maximum number of characters which can be stored in the buffer. Once the end of the buffer is reached, no more characters will be stored.

Return value

If successful – returns the number of characters that have been copied into buffer.

If unsuccessful – returns 0 if there is no message to retrieve or if the message can not be stored in buffer.

Example

```
char previous_msgs[256];
cre_fetch_message(-1, previous_msgs, 256);
```

## cre_fetch_error

```
synopsis      #include “cre_tron.h”
              int cre_fetch_error
                (int msg, char *buffer, int bufsize)
```

Description

Retrieves error messages from the error buffer. Can be used to assist with de-bugging and to provide a means of checking for problems. The error messages are copied into a buffer so that they can be read, printed to the screen etc.

Parameters

msg

If msg >=0 then the error which is retrieved is the one at the index msg from the END of the list of error messages. The smaller the value of msg, the more recent the message. If msg < 0 then this function will retrieve as many error messages as possible beginning with the most recent until the buffer is full.

buffer

A pointer to the location where the retrieved messages will be stored as a string.

bufsize

The maximum number of characters which can be stored in the buffer. Once the end of the buffer is reached, no more characters will be stored.

Return value

If successful – returns the number of characters that have been copied into buffer.

If unsuccessful – returns 0 if there is no error message to retrieve or if the message can not be stored in buffer.

Example

```
char last_error[256];
cre_fetch_message(0, last_error, 256);
```

# Appendix 1 Glossary

• **3D sound**: refers to the fact that sounds in the real world are three-dimensional. Human beings have the ability to perceive sound spatially, meaning that they can hear where a sound is coming from, and where different sounds are in relation to their surroundings and in relation to each other. There are three main pieces of information that are essential for the human brain to perform these functions:

• ITD, or Interaural Time Difference, means that, unless a sound is located at exactly the same distance from each ear (e.g. directly in front), it will arrive at one ear before it arrives at the other. For example, if the sound arrives at the right ear before the left ear, the brain knows that the sound is coming from somewhere to the right.

• IID, or Interaural Intensity Difference, is similar to ITD. It says that if a sound is closer to one ear, the sound's intensity at that ear will be higher than the intensity at the other ear, which is not only further away, but usually receives a signal that has been shadowed by the listener's head.

• Finally, the trickiest part of localization is the fact that a sound bounces off a listener's shoulders, face, and outer ear, before it reaches the eardrum. The pattern that is created by those reflections is unique for each location in space relative to the listener. A human brain can therefore learn to associate a given pattern with a location in space.

Since 3D sound consists of two signals (left and right ear) it can be rendered on conventional stereo equipment, preferably headphones (because of the clean separation of the two signals). The 3D sound produced by a direct path AuSIM3D system is combined with sound reflections (wavetracing) to create a very high level of realism and immersion in a sound space.

• **ambient channel**: a way of displaying sounds as coming from everywhere - all around the listener. This is useful for background music or ambient sounds such as rain.

• **atmospheric absorption**: the attenuation of sounds as they propagate through a medium. For example, in air the high frequency components of sound attenuate faster than the lower frequency components.

• **AuSIM3D**: binaural, immersive, interactive, real-time 3D audio technology by AuSIM, Inc.

• **auralization**: the process of rendering audio by physically or mathematically modeling the sound field of a source in space in such a way as to simulate the binaural listening experience at any given position in the modeled space.

• **binaural**: two audio tracks, one for each ear (as opposed to stereo, which is one for each speaker). Binaural sounds are what we hear in everyday life.

• **boresight axis**: axis defined by the boresight direction (see boresight direction)

• **boresight direction**: principal direction of aural emission of a sound source

• **Chapin, William**: the founder of AuSIM Inc. and inventor of the Crystal River Engineering Acoustetron, WaveTracing, and AuSIM3D GoldServe Audio Localization Server. Often confused with Harry Chapin, his distant cousin and legendary performing artist, Harry is deceased.

• **Convolvotron**: the world's first multi-source, real-time, digital localization system built by Crystal River Engineering for NASA in 1987.

• **direct path**: the direct path from a sound source to a listener's ears (as opposed to the path that includes reflections off surfaces). The direct path allows a listener to tell where each sound is coming from, 360 degrees both in azimuth and elevation. This is the main concept of any 3D sound system.

• **Doppler effect**: the change in frequency of a sound wave due to the relative motion between a sound source and listener. For example, when a car moves past you while sounding its horn, you will hear a drop in pitch as the car passes.

• **extended stereo**: a term that summarizes a number of techniques that involve processing of traditional stereo sounds with the goal of making them appear to originate from a range which extends beyond the physical speaker locations. The effect is often limited to a planar arc in front of the listener with everything at the same elevation. Extended stereo effects tend to be incompatible with headphone listening and to only have the intended effect if the listener is located at a particular spot in relation to the speakers (see "sweet spot").

• **Fisher, Scott**: the founder of Telepresence Research and original director of the Virtual Interactive Environment Workstation (VIEW) project at NASA's Ames Research Center. Often confused with Scott Foster, his friend and the founder of Crystal River Engineering and inventor of the Convolvotron, both hail from MIT in the 1970's and the Atari Research Labs in the early 1980's. The VIEW project, started in 1984, determined the need for virtual interactive aural imaging and initiated the development of the Convolvotron.

• **Foster, Scott**: the founder of Crystal River Engineering and inventor of the Convolvotron. Often confused with Scott Fisher, his friend and founder of Telepresence Research, both hail from MIT in the 1970's and the Atari Research Labs in the early 1980's. Scott Foster is considered by many to be the founder of interactive localized audio, since his Convolvotron was the first device to bring the technology to auralization.

• **gain**: the amplification or attenuation of a sound source, usually measured in dB (decibels). 0 dB means no amplification and no attenuation. A positive value amplifies a source, a negative value attenuates it.

• **HRTF**: HRTFs, or Head Related Transfer Functions, are sets of mathematical transformations that can be applied to a mono sound signal. The resulting left and right signals are the same as the signals that someone perceives when listening to a sound that is coming from a location in real-life 3D space. HRTFs are the core concept behind AuSIM3D, since they contain the information that is necessary to simulate a realistic sound space (see localization). Once the HRTF of a generic person is captured, it can be used to create AuSIM3D sound for a large percentage of the population (most people's heads and ears, and therefore their HRTFs, are similar enough for the filters to be interchangeable).

• **IID**: Interaural Intensity Difference, see "3D sound".

• **ITD**: Interaural Time Difference, see "3D sound".

• **listener**: an object in a sound space that is sampling ("listening to") sound, usually a head with associated HRTF characteristics.

• **materials**: by absorbing sound energy at different frequencies, the material of which an object is made effects the way the sound reflects off and transmits through the object. A carpeted room sounds very different from a glass room. An object's material characteristics can be measured empirically by recording known sounds as they bounce off materials.

• **medium**: see "atmospheric absorption" and "transmission loss".

• **MIDI**: MIDI, or Musical Instrument Digital Interface, is a standard control language that is used for communication between electronic music and effects devices.

• **mono/monophonic**: refers to a single audio signal, usually rendered on a single speaker. Mono sounds appear to originate from the speaker, or from the center of a listener's head in the case of headphones.

• **psychoacoustics**: an area of psychology that studies the structure and performance of human auditory perception.

• **quadraphonic sound**: refers to four audio signals, usually rendered on four separate speakers. Quadraphonic sounds appear to originate from somewhere in-between the four speakers. The inconvenience associated with the amount of equipment necessary to produce quadraphonic sound, coupled with the fact that it is not compatible with conventional stereo equipment (and therefore headphones), makes quadraphonic sound an unpopular choice.

• **radiation pattern**: each sound-emitting object can optionally radiate sound in a certain pattern (rather than uniformly all around it). For example, a head should emit sounds in the direction that its nose is pointing.

• **reflection**: a sound reflection off a surface. It gives a listener information about the listening environment and the location and motion of sound sources. See "surfaces".

• **refraction**: sounds waves refract or bend as they travel around the edges and through openings of objects.

• **reverberation**: or reverb, refers to the sum of all sound reflections in a listening environment.

• **sample rate**: the number of samples per second at which a sound is processed (usually ranges from 8kHz to 50kHz (CD quality is 44.1kHz, or 44,100 samples per second).

• **source**: refers to an object in 3D space that emits sound. The actual sound signal that it sends out can be a live signal, a wave file, or any other audio signal. A 3D sound device is often rated on how many different sources it can independently position at any one time. Realistic sound spaces can be created with as few as four concurrent sources; very complex spaces can have dozens of separate sounds at a time.

• **speaker arrays**: an installation of multiple speakers in a certain pattern, usually designed to create a sound field within the space defined by the speakers. Examples are stereo speakers, or quadraphonic speakers.

• **stereo/stereophonic**: refers to two audio signals, usually rendered on two separate speakers. Stereo sounds appear to originate from somewhere between the two speakers, or between the ears of a listener in the case of headphones.

• **surfaces**: sounds not only travel to a pair of ears on a direct path, but they also bounce off objects in the world. Most natural listening environments contain at least a sound reflecting ground plane, such as a floor. Therefore, reflecting objects are necessary to make virtual environments sound natural and realistic. They help listeners navigate and enhance the overall effect of immersion in a virtual environment. Almost as important as reflections, is the absence of a reflection. For example, the brain can tell the change in a sound space when A reflection is removed by opening a door or a window.

• **sweet spot:** the location where a listener has to be placed to get the optimal effect when listening to a specific speaker setup.

• **transmission loss**: sounds get absorbed as they travel through objects such as walls (similar to atmospheric absorption in the case of traveling through a medium). Transmission loss models are needed to realistically simulate sounds outside a window or in the next room.

• **update rate**: the number of times that a specific instance of a sound space is re-computed and updated per second. Each time any object moves (most often the listener), the space needs to be updated. The higher the update rate, the faster objects can move without creating audio artifacts, such as clicking. Audio update rates generally range from a minimum of 20Hz to 100Hz. Video update rates are usually in the same range (TV signals are updated at 30Hz).

• **wave file**: a digital sound file stored in the Microsoft RIFF file format.

• **wavetracing**: the idea of tracing sound waves as they emit from a source and bounce around an environment (walls, objects, openings). The resulting sound reflections are rendered to a listener to create a more convincing 3D effect, as well as a more immersive, familiar, and realistic sound space.

## FCC Notice

WARNING: This equipment generates, uses, and can radiate radio frequency energy, and if not installed and used in accordance with this instruction manual, may cause interference to radio communications. It has been tested and found to comply with the limits of a class A computing device pursuant to Subpart J of Part 15 of FCC rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of this equipment in a residential area is likely to cause interference in which case the user, at her own expense, will be required to take whatever measures necessary to correct the interference.

# Appendix 2 Errata

• **Chapter 1, Real-time signal input**: Real time signal input is not yet supported at this time. However, it is at the top of the list of functionality to be completed, and it is expected to be available within a matter of days.

• **Chapter 2, Software**: Although the sample UNIX GUI application audioClient in theory should work as described, it has not yet actually been tested by AuSIM with the current line of GoldServe servers. Therefore, the reliability of this application should be considered less solid than other applications this document mentions.

• **Chapter 2, System components and specifications**: System library and demos not yet available on CD; GoldServe User Manual (this document) available only online, not in hardcopy.

• **Chapter 2, Hardware:** Step 3) The double beep when the server is initialized has not yet been activated. Step 5 and Startup problems) Audio service through ethernet is not supported at this time. However, the GoldServe System does come with ethernet networking software installed, so it can be connected to a LAN for purposes of transferring files, etc.

• **Chapter 3, Run-time usage:** The monitor and keyboard are required for runtime usage at this time. This is because the operator must press the CNTL-ALT-DELETE keys and enter the username "Golddigger" and the password "nugget" in order to login to the Windows NT desktop. With practice, this may be done without a monitor, but a keyboard is still required at boot time. Also, the double beep has yet to be added.

• **Code Defects:**
D1001 – cre_open_wave, wavefiles < 1Mb fail to return success flag