UltraSound Lowlevel Toolkit

Revision 2.01
20 May 1993

Advanced Gravis
101-3750 North Fraser Way
Burnaby, British Columbia V5J 5E9
FAX (604)-431-5155

Forte Technologies
1555 East Henrietta Rd.
Rochester, N.Y. 14526
FAX (716)-292-6353

Advanced Gravis and Forte Technologies
Low Level Toolkit/Source for UltraSound

NOTICE

The information contained in this manual is believed to be correct. The manual is subject to change without notice and does not represent a commitment on the part of FORTE or Advanced Gravis.

Neither FORTE nor Advanced Gravis make a warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Neither FORTE nor Advanced Gravis shall be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Table of Contents

Chapter 1 General Information

1.1 Features

- Jumper selectable base port address.
- Software selectable IRQ vectors and DMA channels.
- XT and AT compatibility.
- 8 or 16 bit playback. Stereo and Monophonic.
- 8 bit recording. Stereo or Monophonic.
- Playback and recording rates up to 44.1 kHz.
- 32 voice playback. All voices mixed on board.
- Separate recording channel for simultaneous playback and recording.
- Each voice has its own volume settings, volume enveloping, playback rate and pan (balance) position.
- Both line level and amplified outputs.
- MIDI on-board.
- Gravis Eliminator joystick interface with jumper enable/disable.
- 256K DRAM on board for waveforms. Expandable to 1 MEG
- Right/Left panning (balance) on a per voice basis.
- Stereo microphone input with automatic level control.
- Line level input.
- CD ROM Drive audio input.
- Right/Left mini-jacks for line & amplified outputs and Mic and Line inputs.

1.2 MIDI Interface Functional Description

The MIDI 101 interface consists of standard UART functionality - Motorola MC68C50. An interrupt to the PC is generated for each byte of data received or transmitted. This hardware is independent of any of the other hardware. This circuitry is also included in the GF1. External to the GF1 is an optical isolator that is used on the serial input data and an open collector driver that is used for the serial output. In addition, external logic is included on board to loopback transmit data to the receive data under software control. The serial interface will have a fixed configuration with no programmable options as in the MC6850. A control register is used to enable and disable the interrupt generation logic. A status register is used to determine if the transmit or receive register is interrupting. A read or write to the data register clears the interrupt status. The following are the hardware specifications:

31.25 KHZ +- 1%
asynchronous
1 start bit
8 data bits
1 stop bit

The MIDI signals are available on the 15 pin D connector used for the Joy Stick. An external cable assembly containing the optical isolator and driver is required to use MIDI.

1.3 JoyStick Interface

The joystick interface is an Eliminator Joystick interface designed by Advanced Gravis. The joystick interface consists of an eight bit register. When written to, four flip-flops are reset and

comparator inputs (LM339) begin to charge up based on the position of the joystick. The comparator threshold is setup in the GF1. Crossing the threshold of the comparators cause the flip-flops to be preset and the capacitor to be discharged. Reads of the register return the state of four digital inputs (internally pulled up) and the state of the flip-flops. The rate of discharge of the capacitors has a minimum time constant of 1 us. A jumper on board is used to enable or disable the joystick.

1.4 GF1 - 32 Voice Sound Synthesizer

The method of sound synthesis is Wave Table Synthesis. Either sampled data from actual instruments or other synthesized digital audio is stored in memory on the Ultrasound Board. The GF1 is setup to playback relatively short digital audio samples and produce continuous sound exactly reproducing the original instrument. The 32 voices are independent and can concurrently be producing different sounds which are mixed into either the left or right channel output. Circuitry in the GF1 can be programmed to perform the following audio processing functions independently for each voice:

- Frequency Shifting to produce different notes of the same instrument.
- Amplitude Modulation to produce note enveloping (attack, decay, sustain, release), overall volume control or special effects such as LFO (low frequency amplitude modulation)
- Panning the voice from left to right channel outputs.

This method of Audio synthesis is the same method used in expensive keyboards.

Continuous digital audio recordings can be played through the GF1 by using one voice per digital audio track. The GF1 is compatible with 8 and 16 bit data, stereo or mono, signed or unsigned data, and 8 or 16 bit DMA channels.

1.5 Theory of Operation

This section describes the theory behind the operation of the GF1.

The GF1 is basically a pipeline processor. It constantly loops from voice #0 to the end of the active voices. (How to define then number of active voices is shown later). Every 1.6 microseconds, the GF1 performs a series of operations on a particular voice. The more active voices there are, the longer it takes between each time a particular voice is serviced. This puts a limit on the rate at which playback can occur. 14 active voices will allow a maximum of 44.1 kHz playback. 28 voices will allow 22 kHz. Faster rates can be achieved by making the frequency constant greater than 1. This will cause the GF1 to skip some data bytes to play a sample back at the requested frequency. This is not generally a problem, but could cause some distortion or aliasing.

The formula for calculating the playback rate is:

Frequency divisor	Active voices	Frequency divisor	Active voices
44100	14	41160	15
38587	16	36317	17
34300	18	32494	19
30870	20	29400	21
28063	22	26843	23
25725	24	24696	25
23746	26	22866	27
22050	28	21289	29
20580	30	19916	31
19293	32		

This table is calculated by knowing that 14 active voices will give exactly 44.1 khz playback. Therefore:

1,000,000 / (X * 14) = 44100 X = 1.619695497

Once that is known, the frequency divisor is calculated by:

1,000,000 / (1.619695497 * # of active voices)

The lowlevel code pre-calculates this table (see **vocfreq.c**) so that floating point arithmetic doesn't need to be done. To calculate a FC (frequency counter) for any given frequency with a particular # active voices, run it through this formula:

```
fc = (unsigned int)(((speed_khz<<9L)+(divisor>>1L))/divisor);
fc = fc << 1;
```

(The last left shift is needed because the FC is in bits 15-1. Bit 0 is not used).

This is then put in the frequency control register for that particular voice. If the mantissa portion of the FC is 1, then each time around the loop, the GF1 uses each data point to play. If there is a fractional portion, the GF1 interpolates the actual data to play from the two data points that it is between. This makes the sound much 'smoother', since the GF1 will create points in between the actual data points. For example, assume an 8 bit recording at 22 khz and 14 active voices . The frequency control register is set up to 1/2 (exponent = 256). This means that every time around the loop, that particular voices accumulator is adjusted by 1/2. So the first time the accumulator is 0 and data point 0 is used. The second time around the loop, the accumulator is 0.5. Since there obviously is no DRAM location 0.5, the GF1 interpolates what the data would be by looking at location 0 and location 1 and taking the appropriate ratio from each. In this case, it picks a point half-way between the two. If the recording rate were 11khz, it would take 25% from location 0 and 75% from location 1 the first time thru the loop. The next time it would take 50% from each. The next time it would take 25% from location 0 and 75% from location 1. The forth time thru it uses 100% of location 1.

The interpolation is done to a resolution of 16 bits, even for 8 bit playback. This interpolation has the effect of making an 8 bit recording sound better when played back on the GF1 than on a standard 8 bit card.

Remember that the GF1 works on a voice every 1.6 microseconds. This means that the fewer voices, the faster each voice gets updated. The frequency control register setting for the voice MUST take this into account. The FC must get smaller if the number of active voices gets smaller. This will increase the number of points created between the actual data points so the perceived playback speed remains the same.

NOTE: The volume enveloping uses the same principle when ramping up and down.

1.6 Hardware Volume Ramping

The Ultrasound has built-in volume ramping to facilitate the implementation of the attack decay sustain release envelopes. A section of the envelope can be programmed such that the PC does not need to be burdened with the task of changing each volume at specified intervals. At the end of that particular section, an IRQ can be generated so that the next section can be programmed in. This continues until the entire envelope has been completed. The start and end points as well as the increment rate are fully programmable. The register definitions are:

Current Volume (9,89)	**EEEEMMMMMMMM**	**(Bits 15-4)**
Volume Start (7,87)	**EEEEMMMM**	**(Bits 7-0)**
Volume End (8,88)	**EEEEMMMM**	**(Bits 7-0)**
Volume Incr (6,86)	**RRMMMMMM**	**(Bits 7-0)**

Once the current volume, start and end volumes are programmed, the only thing left is to set up the volume increment register. This register determines how fast the ramp takes place and with

what granularity. The finer the granularity, the smoother (but slower) the ramp. The increment register has 2 fields. The first is the amount added to (or subtracted from) the current volume to get to the next one. These are the low 6 bits and can range from 1 to 63. A 1 is a long, slow ramp compared to a 63. The upper 2 bits determine how often the increment is applied to the current volume. The rate bits are defined as:

Rate Bits	Volume Update Rate
00	FUR (FUR = 1/(1.6*#active voices))
01	FUR / 8
10	FUR / 64
11	FUR / 512

Each rate increment is 8 times longer than the preceding one. This means that the value to store for the fastest possible ramp is **`0x1F`** (63), and the value for the slowest possible ramp is **`0xC1`** (193). The approximate times for a full scale volume ramp (0 - 4095) are:

Rate	Vol Inc	14 Voices	32 Voices
0	63	1.4 ms	3.3 ms
0	1	91.7 ms	209.7 ms
1	63	11.5 ms	26.2 ms
1	1	733.8 ms	1.7 sec
2	63	91.8 ms	209.7 ms
3	1	5.9 sec	13.4 sec
3	63	734.0 ms	1.7 sec
3	1	47.0 sec	107.3 sec

Note that these times are for full scale ramping. Since the volume ramps usually go between points in between the rails, the actual ramp times will be much smaller.

The volume ramping can be very useful for things other than the enveloping. Since there are only 16 pan positions, a balance sweep from right to left may produce clicks since there is such a large jump between pan positions. You can get a very smooth balance sweep using 2 voices and volume ramping. Just set one voice up to the right, one up to the left and ramp one down from volume X to zero at the same rate as you ramp the other from 0 up to volume X.

It can also be used to remove clicks and pops at the beginning and end of digital samples. By setting up fairly fast rates when a sample start or ends and ramping up or down appropriately, any pop created by a sudden change in the DAC value will be summed in at such a low volume, it will never be heard.

1.7 Tips on click removal.

There are several ways that clicks and pops may be produced on the Ultrasound. Depending on what causes it, it can be very tricky to prevent.

1.7.1 During Reset

The simplest click to create and remove is during initialization. When the GF1 is reset, a pop may happen. To prevent this, only reset the GF1 with the output disabled.

1.7.2 During balance sweep

Since there is only 16 pan positions, it is quite likely that you will get pops when changing pan positions. Particularly if it is done at a fairly high rate. To overcome this, use 2 voices and volume ramping. Set the balance positions on both voices, and ramp one down at the same rate the other ramps up. The result is a very smooth balance sweep.

1.7.3 When a voice starts

A click can happen when the output of the DAC changes suddenly. This is the most frequent type of click, since there may not be a way to control the data values the voice is trying to play.

The way to avoid the click is to set up a volume ramp that starts at a low volume when the sample starts and ramps up to an audible volume. That way the click will be summed in at such a low volume that it will not contribute much to the final output DAC value.

1.7.4 When a voice ends

This usually happens because a voice stops abruptly. The solution to this is to set up a fast volume ramp that will take the voice from its current volume down to 0 very quickly.

1.7.5 General

It is necessary to remember that all voices are being summed in to the final output, even if they are not running. This means that whatever data value that the voice is pointing at is contributing to the summation. It is important that a voice be pointed to a known value at a known location after it is stopped so that some control is kept over it. For instance, if a voice were left at where ever the end position was for the last time it played, a pop could occur if new data were either DMA'ed or poked over the top of it. It is recommended that a voice be pointed to a location containing a 0 and that its volume be set to 0. Now that voice will have no contribution to the output.

A possibly useful side-effect of this summation process could be called direct DAC output. This can be accomplished by setting a voice up to point to a specific location with a given volume and poking in the data bytes at a constant rate. It will not sound as good as letting the GF1 play the sound because no interpolation or oversampling will occur. However, it could be useful for some applications.

Chapter 2 - Hardware Information

2.1 I/O Port Map

The following describes I/O address map used on the board. The 'X' is defined by the jumper settings on the UltraSound and should match that specified in the **ULTRASND** environment variable.

INTERFACE	I/O,MEM	R,W	ADDRESS	
MIDI Interface	Control I/O	W	3X0	
	Status I/O	R	3X0	
	Transmit Data I/O	W	3X1	
	Receive Data I/O	R	3X1	
Joystick Interface	Trigger Timer I/O	W	201	
	Read Data I/O	R	201	
GF1 Synthesizer:	GF1 Page Register I/O	R/W	3X2	
	GF1/Global Register Select I/O	R/W	3X3	
	GF1/Global Data Low Byte I/O	R,W	3X4	
	GF1/Global Data High Byte I/O	R/W	3X5	
	IRQ Status Register 1=ACTIVE I/O	R	2X6	
	Timer Control Reg I/O	R/W	2X8	
	Timer Data I/O	W	2X9	
	DRAM I/O	R,W	3X7	
	DRAM DMA	R,W	1,3,5,6,7	
	Record Digital Audio DMA	R	1,3,5,6,7	
BOARD ONLY	Mix Control register I/O	W	2X0	
	IRQ control register I/O	W	2XB	(2X0- bit 6 = 1)
	DMA control register I/O	W	2XB	(2X0- bit 6 = 0)

At powerup the board will have operational Joystick and MIDI interfaces. This will allow their direct use with existing software. The GF1 ASIC will power up with all voices disabled not requiring a software initialization. This will help eliminate noise at powerup and allow the Joystick and MIDI interfaces to be used by existing applications. The IRQ control register MUST be set up before the MIDI interface can generate an IRQ. This is done in **ultrinit.exe** and when an application sets up the latches to the **ULTRASND** parameters.

2.2 MIDI Control Port - 3X0

Here are the bit definitions for the MIDI control byte. It is located at the 3X0 hex and is write only.

Bit 7 (0x80) **1 = Receive IRQ Enable**

Bit 6 (0x40) **0**

Bit 5 (0x20) **1 = Transmit IRQ Enable**

Bit 4 (0x10) **Reserved**

Bit 3 (0x08) **Reserved**

Bit 2 (0x04) **Reserved**

Bit 1 (0x02) **1**

Bit 0 (0x01) **Master Reset (when set)**

Bit 0 & 1 will cause a master reset when toggled high and then low. They must be left low when using port. This will normally cause a transmit buffer empty IRQ.

2.3 MIDI Status Port - 3X0

Here are the bit definitions for the MIDI status byte It is located at the 3X0 hex and is read only.

Bit 7 (0x80) **Interrupt Pending**

Bit 6 (0x40) **Reserved**

Bit 5 (0x20) **Overrun Error**

Bit 4 (0x10) **Framing Error**

Bit 3 (0x08) **Reserved**

Bit 2 (0x04) **Reserved**

Bit 1 (0x02) **Transmit Register Empty**

Bit 0 (0x01) **Receive Register Full**

The MIDI control behaves identically to a 6850 UART.

2.4 MIDI Data Port - 3X1

The transmit and receive data registers are at 3X1 hex and are 8 bits wide.

2.5 Page Register - 3X2

This could also be called the voice select register. This register is used to specify which voice's registers you want to read/write. This value can range from 0 to the number of active voices specified (13-31). Once this has been specified, you may select the specific register within that voice. Be careful that IRQs are off during the time that the Page and Select registers are being modified. This will prevent the foreground from selecting a voice and having the background change it in the background.

2.6 Select Register - 3X3

2.6.1 Global Registers

These are the global registers. These registers are not voice-specific.

Address	Mode	Width	Description
41	R/W	8	DRAM DMA Control
42	W	16	DMA Start Address
43	W	16	DRAM I/O Address (LOW)
44	W	8	DRAM I/O Address (HIGH)
45	R/W	8	Timer Control
46	W	8	Timer 1 Count
47	W	8	Timer 2 Count
48	W	8	Sampling Frequency
49	R/W	8	Sampling Control
4B	W	8	Joystick trim DAC
4C	R/W	8	RESET

2.6.1.1 DRAM DMA Control Register - (41)

Bit 7 (0x80) **Invert MSB (Write only):** Invert High bit to flip data to twos complement form. Note: This flips bit 7 for 8 bit data and bit 15 for 16 bit data.

Bit 6 (0x40) **DMA IRQ Pending (Read), 0/1 = 8/16-bit data (Write):** Note: Data size is independent of channel size.

Bit 5 (0x20) **DMA IRQ Enable**

Bits 3,4 (0x18) **DMA Rate Divider:** The Maximum rate is approx 650 khz. 00/01/10/11 = divide by 1/2/3/4.

Bit 2 (0x04) **DMA Channel Width:** 0/1 = DMA channel is an 8 bit channel (0-3)/16 bit channel (4-7) Note: This is independent of the data size.

Bit 1 (0x02) **1/0 = DMA Read/Write:** Read is taking data OUT of the UltraSound. Write sends data to it.

Bit 0 (0x01) **DMA Enable:** Enable the DMA channel. The GF1 will begin sending DMA ACK protocol. If PC DMA controller is programmed, data will begin being transferred. If not, data will move as soon as it is programmed.

2.6.1.2 DMA Start Address - (42)

Bits 15-0 are Address lines 19-4.

This register defines where the DMA will transfer data to/from. Since only the upper 16 address bits are used and the lower 4 bits are set to 0, a DMA transfer MUST begin on an 16 byte boundary for an 8 bit DMA channel (0-3). If a 16 bit DMA channel is being used, the transfer MUST being on a 32 byte boundary. An additional address translation is necessary if a 16 bit DMA channel is used. Here is the C code to do the translation. (See function **`convert_to_16()`**):

```
unsigned long address;
unsigned long hold_address;

hold_address = address; /* Convert to 16 translated address */
address = address >> 1; /* Zero out bit 17 */
address &= 0x0001FFFFL; /* Reset bits 18 and 19 */
address |= (hold_address & 0x000c0000L);
```

2.6.1.3 DRAM I/O Address (43,44)

These 2 registers allow you to specify an address to peek and poke directly into UltraSound DRAM. Register 43 is the lower 16 address lines. Register 44 is the upper 4 address lines. (bits 0-3). Read or write to register 3X7 to get at the address location.

2.6.1.4 Timer Control - (45)

Bit 7 (0x80) **Unused**

Bit 6 (0x40) **Unused**

Bit 5 (0x20) **Unused**

Bit 4 (0x10) **Unused**

Bit 3 (0x08) **Enable Timer 2 IRQ**

Bit 2 (0x04) **Enable Timer 1 IRQ**

Bit 1 (0x02) **Unused**

Bit 0 (0x01) **Unused**

2.6.1.5 Timer 1&2 Count - (46,47)

These counts are loaded by the application and then they will count up to 0xFF and generate an IRQ. Timer 1 has a granularity of 80 microsecs and Timer 2 has a granularity of 320 microsecs.

2.6.1.6 Sampling Frequency - (48)

The formula for calculating this value is:

rate = 9878400/(16*(FREQ+2))

2.6.1.7 Sampling Control Register - (49)

Bit 7 (0x80) **Invert MSB (Write only):** Invert High bit to flip data to twos complement form.

Bit 6 (0x40) **DMA IRQ Pending (Read), 0/1 = 8/16-bit data (Write):** Note: Data size is independent of channel size.

Bit 5 (0x20) **DMA IRQ Enable**

Bits 3,4 (0x18) **DMA Rate Divider:** The Maximum rate is approx 650 khz. 00/01/10/11 = divide by 1/2/3/4.

Bit 2 (0x04) **DMA Channel Width:** 0/1 = DMA channel is an 8 bit channel (0-3)/16 bit channel (4-7) Note: This is independent of the data size.

Bit 1 (0x02) **0/1 = Mono/Stereo:** In stereo mode, the order of the data bytes is left is first, and right is second. If a 16 bit data channel is used, the left is in the lower byte.

Bit 0 (0x01) **DMA Enable:** If PC DMA controller is programmed, it will begin sampling as soon as this is enabled.

2.6.1.8 Joystick Trim DAC - (4B)

This register is initialized to 4.3 volts (value = 29). It only needs to be modified to account for faster/slower machines. A utility is provided (**`ultrajoy.exe`**) that sets this up. There should be no reason for your application to modify this register.

2.6.1.9 Reset Register - (4C)

Bit 7 (0x80) **Unused**

Bit 6 (0x40) **Unused**

Bit 5 (0x20) **Unused**

Bit 4 (0x10) **Unused**

Bit 3 (0x08) **Unused**

Bit 2 (0x04) **GF1 Master IRQ Enable**

Bit 1 (0x02) **DAC Enable**

Bit 0 (0x01) **Master Reset**

As long as bit 0 is a 0, it will be held in a reset state. DAC's will not run unless bit 1 is set. Bit 2 MUST be set to get ANY of the GF1-generated IRQs (wavetable,volume etc). This register will normally contain a 0x07 when your application is running.

2.6.2 Voice Specific Registers

These are the voice-specific registers. Each voice has its own bank of read and write registers that alter its behavior. The write registers range from 0 to F and the corresponding read registers range from 80 to 8F. To convert from the write to the read, just add 80 hex.

Write	Read	Width	Description
0	80	8	Voice Control
1	81	16	Frequency Control
2	82	16	Starting Address (HIGH)
3	83	16	Starting Address (LOW)
4	84	16	End Address (HIGH)
5	85	16	End Address (LOW)
6	86	8	Volume Ramp Rate
7	87	8	Volume Ramp Start

8	**88**	**8**	**Volume Ramp End**
9	**89**	**16**	**Current Volume**
A	**8A**	**16**	**Current Address (HIGH)**
B	**8B**	**16**	**Current Address (LOW)**
C	**8C**	**8**	**Pan Position**
D	**8D**	**8**	**Volume Control**
E	**8E**	**8**	**Active Voices (Voice independent)**
F	**8F**	**8**	**IRQ Status (Voice independent)**

There are several 'self-modifying' bits defined in the following registers. This means that the GF1 may change them at anytime on its own. The software must accommodate this phenomena. Because of this, it's possible that the GF1 may change something immediately after your application has set/reset one of the bits. This is due to the GF1's pipeline processor type of architecture. It does a read-modify-write cycle, and if your application modifies one of these bits AFTER its done the read portion and BEFORE it does the write portion, it's possible for the chip to perform incorrectly. To overcome this, you need to do a double write (with a delay in between) when those particular bits are involved. This delay must be at least 3 times the length of time necessary to process a voice. (3*1.6 microsecs). In the lowlevel source code, this is done with a function called **`gf1_delay()`**. The self-modifying bits are designated with an (*) after the particular bit definition.

2.6.2.1 Voice Control Register - (0,80)

Bit 7 (0x80) **IRQ Pending:** If IRQ's are enabled and looping is NOT enabled, an IRQ will be constantly generated until voice is stopped. This means that you may get more than 1 IRQ if it isn't handled properly.

Bit 6 (0x40) **Direction of Movement*:** 0/1 = increasing/decreasing addresses. It is self-modifying because it might shift directions when it hits one of the loop boundaries and looping is enabled.

Bit 5 (0x20) **Wave Table IRQ:** Generate an IRQ when the voice hits the end address.

Bit 4 (0x10) **Bidirectional Loop Enable**

Bit 3 (0x08) **Loop Enable:** Loop to begin address when it hits the end address.

Bit 2 (0x04) **0/1 = 8/16-bit Sample Width**

Bit 1 (0x02) **Stop Voice:** Manually force voice to stop.

Bit 0 (0x01) **Voice Stopped:** This gets set by hitting the end address (not looping) or by setting bit 1 in this reg.

2.6.2.2 Frequency Control Register - (1,81)

Bit 15-10 (0xFC00) **Integer portion**

Bit 9-1 (0x03FE) **Fractional portion**

Bit 0 (0x01) **Unused**

This register determines the amount added to (or subtracted from) the current position of the voice to determine where the next position will be. This is how the interpolated data points are determined. If the FC register is less than 0, the GF1 will interpolate the data point in between the two actual data points. Note that the FC can be greater than 1. This allows for skipping over data bytes. The actual frequency that it will play back is directly related to the number of active voice specified (reg 8E).

2.6.2.3 Starting location HIGH - (2,82)

Bits 12-0 are the HIGH 13 bits of the address of the starting location of the waveform (address lines 19-7). Bits 15-13 are not used.

2.6.2.4 Starting location LOW - (3,83)

Bits 15-9 are the low 7 bits of the address of the starting location of the waveform (address lines 6-0). Bits 8-5 are the fractional part of the starting address. Bits 4-0 are not used.

2.6.2.5 End Address HIGH - (4,84)

Bits 12-0 are the high 13 bits of the address of the ending location of the waveform (address lines 19-7). Bits 15-13 are not used.

2.6.2.6 End Address LOW - (5,85)

Bits 15-9 are the low 7 bits of the address of the ending location of the waveform. (address lines 6-0). Bits 8-5 are the fractional part of the ending address. Bits 4-0 are not used.

2.6.2.7 Volume Ramp Rate - (6,86)

Bits 5-0 is the amount added to (or subtracted from) the current volume to get the next volume. The range is from 1 to 63. The larger the number, the greater the volume step. Bits 7-6 defines the rate at which the increment is applied.

See section 1.6 for a more complete explanation of how this register works.

2.6.2.8 Volume Ramp Start - (7,87)

Bits 7-4 Exponent, Bits 3-0 Mantissa

This register specifies the starting position of a volume ramp. See section 1.6 for a more complete explanation of how this register works.

2.6.2.9 Volume Ramp End - (8,88)

Bits 7-4 Exponent, Bits 3-0 Mantissa

This register specifies the ending position of a volume ramp. See section 1.6 for a more complete explanation of how this register works.

Note: The starting volume must always be less than the ending volume. If you want the volume to ramp down, turn on the decreasing volume bit in the Volume Control Register.

2.6.2.10 Current Volume - (9,89)

Bits 15-12 Exponent*, Bits 11-4 Mantissa*, Bits 3-0 Unused

Note: This register has 4 extra bits of precision that is necessary for finer granularity of volume placement. The extra bits are used during a volume ramp. Note: This is a self-modifying value. The GF1 will update this register as it ramps. Note: You should always set this register equal to the value of the beginning of the volume ramp (start OR end)

2.6.2.11 Current Location HIGH - (A,8A)

Bits 15-13 Unused, Bits 12-0 High 13 bits of address (address lines 19-7)

2.6.2.12 Current Location LOW - (B,8B)

Bits 15-9 Low 7 bits of address (address lines 6-0). Bits 8-0 9-bit fractional position.

2.6.2.13 Pan Position - (C,8C)

Bits 8-4 Unused, Bits 3-0 Pan position (0=full left, 15=full right).

2.6.2.14 Volume Ramp Control Register - (D,8D)

Bit 7 (0x80) **IRQ Pending***

Bit 6 (0x40) **Direction of Movement*:** 0/1 = increasing/decreasing addresses. It is self-modifying because it might shift directions when it hits one of the loop boundaries and looping is enabled.

Bit 5 (0x20) **Volume Ramp IRQ:** Generate an IRQ when the ramp hits the end value.

Bit 4 (0x10) **Bidirectional Loop Enable**

Bit 3 (0x08) **Loop Enable:** Loop to begin value when it hits the end value.

Bit 2 (0x04) **Rollover:** This bit pertains more towards the location of the voice rather than its volume. Its purpose is to generate an IRQ and NOT stop (or loop). It will generate an IRQ and the voice's address will continue to move thru DRAM in the same direction. This can be a very powerful feature. It allows the application to get an interrupt without having the sound stop. This can be easily used to implement a ping-pong buffer algorithm so an application can keep feeding it data and there will be no pops. Even if looping is enabled, it will not loop.

Bit 1 (0x02) **Stop Ramp:** Manually stop the ramp.

Bit 0 (0x01) **Ramp Stopped***

2.6.2.15 Active Voices - (E,8E)

Bits 7-6 Must be set to a 1, Bits 5-0 # of voices to enable - 1.

The range is from 14 - 32. Any value less than 14 will be forced to 14.

2.6.2.16 IRQ Source Register - (F,8F)

Bit 7 (0x80) **0 = Wave Table IRQ Pending**

Bit 6 (0x40) **0 = Volume Ramp IRQ Pending**

Bit 5 (0x20) **Always 1**

Bit 4-0 (0x1F) **Interrupting Voice**

Note: This is a global read only register. There is only 1 for ALL voices. You MUST service any indicated IRQ's since a read of this port will clear the associated IRQ bits in the particular voice's control and/or volume control registers. Note: It is possible that multiple voices could interrupt at virtually the same time. In this case, this register will behave like a fifo. When in your IRQ handler, keep reading (and servicing) this register until you do a read with both IRQ bits set to a 1. This means there are no voice IRQs left to deal with. Note: Since it is possible to get ANOTHER IRQ from the same voice for the SAME reason, you must ignore any subsequent IRQ from that voice while in the IRQ handler. For example, when a voice hits its end position and generates an IRQ back to your application, it will continue to generate IRQ's until either the voice is stopped, the IRQ enable is turned off, or the end location is moved.

2.7 Global Data Low - 3X4

This register can be used to do either a 16 bit transfer for one of the 16 bit wide GF1 registers (Start addr high etc) when using a 16 bit I/O instruction or the low part of a 16 bit wide register when using an 8 bit I/O instruction.

2.8 Global Data High - 3X5

This register is used to do either an 8 bit transfer for one of the GF1 8 bit registers or to do the high part of a 16 bit wide register.

2.9 IRQ Status - 2X6

Bit 7 (0x80) **DMA TC IRQ (DRAM or sample)**

Bit 6 (0x40) **Volume Ramp IRQ (any voice)**

Bit 5 (0x20) **Wave Table IRQ (any voice)**

Bit 4 (0x10) **Unused**

Bit 3 (0x08) **Timer 2 IRQ**

Bit 2 (0x04) **Timer 1 IRQ**

Bit 1 (0x02) **MIDI Receive IRQ**

Bit 0 (0x01) **MIDI Transmit IRQ**

CAUTION: Note that this is at 2X6 **NOT** 3X6!.

2.10 Timer Control Register - 2X8

This register maps to the same location as the ADLIB board's control register. Writing a 4 here selects the timer stuff. Bit 6 will be set if timer #1 has expired. Bit 5 will be set it timer #2 has expired. See **`timer.c`** for an example of programming the timers.

2.11 Timer Data Register - 2X9

Bit 7 (0x80) **Clear Timer IRQ**

Bit 6 (0x40) **Mask Timer 1**

Bit 5 (0x20) **Mask Timer 2**

Bit 4 (0x10) **Unused**

Bit 3 (0x08) **Unused**

Bit 2 (0x04) **Unused**

Bit 1 (0x02) **Timer 2 Start**

Bit 0 (0x01) **Timer 1 Start**

2.12 DRAM I/O - 3X7

This register is used to read or write data at the location pointed at by registers 43 and 44. This is used to peek and poke directly to DRAM.

2.13 Mix Control Register - 2X0

Bit 7 (0x80) **Unused**

Bit 6 (0x40) **Control Register Select:** When this is set to a 1, the next IO write to 2XB will be to the IRQ control latches. When this is set to a 0, the next IO write to 2XB will be to the DMA channel latches. The write to 2XB for either of these MUST occur as the NEXT IOW or else the write to 2XB will be locked out and not occur. This is to prevent an application that is probing for cards to accidentaly corrupt the latches.

Bit 5 (0x20) **Enable MIDI Loopback (TxD to RxD)**

Bit 4 (0x10) **Combine Channel 1 IRQ with Channel 2 (MIDI)**

Bit 3 (0x08) **Enable Latches:** This provides power to the DMA/IRQ latches. Once these are enabled, NEVER disable them. Disabling them will cause random IRQ's in the PC since the DMA and IRQ lines are not being driven any more.

Bit 2 (0x04) **1 = Enable Mic In**

Bit 1 (0x02) **0 = Enable Line Out**

Bit 0 (0x01) **0 = Enable Line In**

2.14 IRQ Control Register - 2XB

IRQ control register I/O W 2XB (2X0- bit 6 = 1)

Bits 2-0 Channel 1 (GF1) IRQ Selector 0=No Interrupt 1=IRQ2 2=IRQ5 3=IRQ3 4=IRQ7 5=IRQ11 6=IRQ12 7=IRQ15

Bits 5-3 Channel 2 (MIDI) IRQ selector 0=No Interrupt 1=IRQ2 2=IRQ5 3=IRQ3 4=IRQ7 5=IRQ11 6=IRQ12 7=IRQ15

Bit 6 1 = Combine Both IRQS using Channel 1's IRQ Bit 7 Unused

Note: If the channels are sharing an IRQ, channel 2's IRQ must be set to 0 and turn on bit 6. A bus conflict will occur if both latches are programmed with the same IRQ #.

DMA control register I/O W 2XB (2X0- bit 6 = 0)

Bits 2-0 DMA Select Register 1 0=NO DMA 1=DMA1 2=DMA3 3=DMA5 4=DMA6 5=DMA7

Bits 5-3 DMA Select Register 2 0=NO DMA 1=DMA1 2=DMA3 3=DMA5

4=DMA6 5=DMA7

Bit 6 - Combine Both on the same DMA channel. Bit 7 - Unused.

Note: If the channels are sharing an DMA, channel 2's DMA must be set to 0 and turn on bit 6. A bus conflict will occur if both latches are programmed with the same DMA #.

Please refer to UltraSetInterface() in init.c of the lowlevel source code for the proper sequence for programming these latches. If the order is not right, unpredictable things may happen.

Changing the IRQ settings will usually cause an IRQ on the OLD IRQ because it is no longer being driven low by the latches and it will tend to float up. That low to high transition causes and IRQ on the PC. Normally, this is not a problem, but it is something to be aware of.

Chapter 3 - Software Information

3.1 Lowlevel Introduction

This low level toolkit will allow developers to write code at a much more basic level than the other toolkit. Its purpose is to allow the programmer direct access to the functionality of the board. Also the code has been highly optimized to make your resulting code as small and fast as possible. Currently, 9 different memory models are supplied. They are: large, medium, small, and tiny for both Borland and Microsoft and flat model with the Watcom compiler. There are also 2 different levels of libraries being shipped. Level 0 contains the lowest level function calls that talk directly to the hardware. The Level 1 library is a little higher and contains functions that call level 0 functions. The 3D libraries contain the functions necessary for implementing the Focal Point 3D sounds. Borland C++ 2.0, Microsoft 6.0 and Watcom C9.0/386 were used respectively.

3.2 Library naming conventions

Library name	Level	Model	Compiler
ultra0lb.lib	0	Large	Borland C
ultra1lb.lib	1	Large	Borland C
ult3d_lb.lib	3D	Large	Borland C
ultra0mb.lib	0	Medium	Borland C
ultra1mb.lib	1	Medium	Borland C
ult3d_mb.lib	3D	Medium	Borland C
ultra0sb.lib	0	Small	Borland C
ultra1sb.lib	1	Small	Borland C
ult3d_sb.lib	3D	Small	Borland C
ultra0tb.lib	0	Tiny	Borland C
ultra1tb.lib	1	Tiny	Borland C
ult3d_tb.lib	3D	Tiny	Borland C
ultra0lm.lib	0	Large	Microsoft C
ultra1lm.lib	1	Large	Microsoft C
ult3d_lm.lib	3D	Large	Microsoft C
ultra0mm.lib	0	Medium	Microsoft C
ultra1mm.lib	1	Medium	Microsoft C
ult3d_mm.lib	3D	Medium	Microsoft C
ultra0sm.lib	0	Small	Microsoft C
ultra1sm.lib	1	Small	Microsoft C
ult3d_sm.lib	3D	Small	Microsoft C
ultra0tm.lib	0	Tiny	Microsoft C
ultra1tm.lib	1	Tiny	Microsoft C
ult3d_tm.lib	3D	Tiny	Microsoft C
ultra0wc.lib	0	Flat	Watcom C
ultra1wc.lib	1	Flat	Watcom C
ult3d_wc.lib	3D	Flat	Watcom C

Several example applications are supplied on the toolkit disks to show you how to interface to the libraries. Please look them over carefully. They are the best way to get a handle on the way the card operates.

The next few sections will attempt to give you an overview of the features the UltraSound has and how to take advantage of them.

3.3 Main Features

32 voices 8 bit and 16 bit playback (mono and stereo)

Signed or Unsigned data

High speed DMA to/from DRAM

8 bit DMA record (mono and stereo)

1 Meg on-board DRAM

Digital volume control

Hardware volume enveloping

Line level input

Microphone input (with auto gain)

Line level output

Amplified output

Speed compensating joystick

3.4 Caveats

There are several things that a low level programmer needs to be aware of to successfully program the UltraSound. The low level code protects against ALL of these things happening, but it is possible for a programmer to bypass these protections.

The DMA can only begin on a 16 or 32 byte boundary. An 8 bit DMA channel (0-3) must start on a 16 byte boundary while a 16 bit DMA channel (4-7) must start on a 32 byte boundary. If an improper address is supplied, the address will be truncated down when the DMA occurs. The lowlevel memory allocation routines will ONLY return 32 byte aligned addresses, so if your application uses them, this will not be a problem. If you chose not to use the allocation routines provided, be sure and follow these rules. (Note: The 3D routines require that the memory allocation routines be used).

A 16 bit sample cannot be played across a 256K boundary. An 8 bit sample can, but a 16 bit sample cannot. This means that all 16 bit samples must be less that 256K. To enforce this restriction, the memory allocation routines split each 256K bank into its own DRAM pool. No memory can be allocated that will cross the boundary. Again, if you chose not to use the memory allocation routines provided, you need to be aware of this.

A DMA to or from the card cannot cross a 256K boundary. Since the lowlevel code doesn't allow you to allocate DRAM across a 256k boundary either, this is not usually a problem (if you use the allocation routines). The functions that do DMA to and from the card (**`UltraUpload()`** and **`UltraDownload()`**), will allow you to try and DMA across a 256K boundary, but will split it up to 2 transfers.

Allowing to let the volume ramps to go to the rails (0 - 4095) can cause a random oscillation of the volume when it reaches the limits. This is caused by an overshoot past the limit due to a large step size. The low level code protects against this by limiting how close to the rails you can get.

3.5 Level 0 Interface Functions

3.5.1 UltraCalcRate

Function: Calculate a rate for a volume ramp.

Syntax:
```
unsigned char UltraCalcRate(start,end,mil_secs);
    unsigned int start;
    unsigned int end;
    unsigned long mil_secs;
```

Prototype: **gf1proto.h**

Remarks: This function calculates the rate necessary to ramp the volume from the start volume to the end volume in a desired # of milliseconds. This value should be passed to **UltraRampVolume()**. This is only an approximation. The longer the time span, the less precise the result is likely to be.

Returns: The value to pass to **UltraRampVolume()**

See also: **UltraRampVolume**, **UltraRampLinearVolume**

3.5.2 UltraClose

Function: Close out the UltraSound card

Syntax:
```
int UltraClose(void);
```

Prototype: **gf1proto.h**

Remarks: This function should be called before your application exits. It shuts down all audio and puts the card in a stable state. It also puts the PC back to the state prior to running your application. (reset IRQ vectors etc).

Returns: **ULTRA_OK** == No problem

See also: **UltraOpen**, **UltraReset**

3.5.3 UltraDownload

Function: Download a chunk of data into UltraSound DRAM.

Syntax:
```
int UltraDownload(dataptr,control,dram_loc,len,wait)
    void *dataptr;          /* ptr to buffer in pc RAM */
    unsigned char control;  /* control bits */
    unsigned long dram_loc; /* location in UltraSound */
    unsigned int len;       /* # of bytes to xfer.  */
    int wait;               /* wait for completion */
```

Prototype: **gf1proto.h**

Remarks: This function will send a chunk of data down to the UltraSound' DRAM. It will transfer ***len*** # of bytes from ***data_ptr*** (in PC) to ***dram_loc*** (in UltraSound). If ***wait*** is **TRUE**, then it will wait until the transfer is complete. If ***wait*** is **FALSE**, it will return as soon as transfer is started. In most cases, waiting is usually OK, but if you need to get output quickly, you can start the download and then immediately start a voice playing the data. The DMA transfer is MUCH faster than the voice playback, so it will be able to download data ahead of the playback. For obvious reasons, this will not work if you want to play the data backwards. See Appendix D for a definition of the control bits. They specify the type of data being sent down.

Returns: **ULTRA_OK** == No problem
DMA_BUSY == Dma channel busy. It may be busy doing a download or upload or a record (if play and record channels are the same).

See also: **UltraUpload**, **UltraDramDmaWait**

3.5.4 UltraDramDmaBusy

Function: Test to see if the DMA channel is busy

Syntax:
```
int UltraDramDmaBusy(void)
```

Prototype: **gf1proto.h**

Remarks: This function will check to see if the DMA to/from DRAM channel is busy. It might be useful so your application doesn't hang around while waiting for a DMA transfer to complete.

Returns: Non-zero == Channel is still busy
0 == Channel is free

See also: **UltraWaitDramDma**, **UltraDownload**, **UltraUpload**

3.5.5 UltraGoRecord

Function: Start up a preset record

Syntax:
```
int UltraGoRecord(control);
    unsigned char control;
```

Prototype: **gf1proto.h**

Remarks: This function will start up a pre-primed record. **UltraPrimeRecord()** can be used to do this. It can also be used to restart a indefinite (autoinit) record from the record handler callback.

Returns: **ULTRA_OK** == No error
DMA_BUSY == This channel is busy

See also: **UltraRecordData**, **UltraPrimeRecord**, **UltraRecordHandler**

3.5.6 UltraGoVoice

Function: Start a voice that has been primed

Syntax:
```
void UltraGoVoice(voice,mode)
    int voice;
    unsigned char mode;
```

Prototype: **gf1proto.h**

Remarks: This function will start up a voice that has already been primed with setup values by **UltraPrimeVoice()**. This can be useful if you need to start multiple voices as close together as possible. See Appendix C for the mode bit definitions.

Returns: None

See also: **UltraPrimeVoice**, **UltraStartVoice**

3.5.7 Input/Output Source Functions

3.5.7.1 UltraDisableLineIn

Function: Disable line level input.

Syntax: **void UltraDisableLineIn(void);**

Prototype: **gf1proto.h**

Remarks: If line level input is enabled and output is enabled, the input is routed directly to the output and audio will be heard. If this is not desired, use this to disable line in.

Returns: None

3.5.7.2 UltraDisableMicIn

Function: Disable microphone input

Syntax: **void UltraDisableMicIn(void);**

Prototype: **gf1proto.h**

Remarks: If microphone input is enabled and output is enabled, the input is routed directly to the output and audio will be heard. If this is not desired, use this to disable microphone in.

Returns: None

3.5.7.3 UltraDisableOutput

Function: Turn off all output from UltraSound

Syntax: **void UltraDisableOutput(void);**

Prototype: **gf1proto.h**

Remarks: This function will disable all output from the UltraSound. This can be used during recording so that the input will not be looped back to the output. It is also useful to disable output during resets etc. It will help eliminate 'pops' during initialization.

Returns: None

3.5.7.4 UltraEnableLineIn

Function: Enable line level input

Syntax: **`void UltraEnableLineIn(void);`**

Prototype: **`gf1proto.h`**

Remarks: Turn on the line level input. If you are not recording from the line input, it is probably not desirable to have this enabled since it will be looped back to the output (if output is enabled).

Returns: None

3.5.7.5 UltraEnableMicIn

Function: Turn on the microphone input

Syntax: **`void UltraEnableMicIn(void);`**

Prototype: **`gf1proto.h`**

Remarks: This function should be called when you want to record from the microphone. It is possible to have both the microphone input enabled and line level input enabled. If you are not recording from the microphone, it is recommended that it be disabled. This will reduce noise on the output.

Returns: None

3.5.7.6 UltraEnableOutput

Function: Enable any output from UltraSound

Syntax: **`void UltraEnableOutput(void);`**

Prototype: **`gf1proto.h`**

Remarks: This function must be called to enable any output from the UltraSound. If output is not enabled, you will get no audio output. It can be used to mute output at a particular time (e.g. reset).

Returns: None

3.5.7.7 UltraGetLineIn

Function: Return the current state of line level input

Syntax: **`int UltraGetLineIn(void);`**

Prototype: **`gf1proto.h`**

Remarks: This can be useful if you want to change the state of the line level input and then restore it back to the original state.

Returns: Current line level input state (**`ON`** or **`OFF`**)

3.5.7.8 UltraGetOutput

Function: Return the current output enable state.

Syntax: **`int UltraGetOutput(void);`**

Prototype: **`gf1proto.h`**

Remarks: This can be useful if you want to change the state of the output and then restore it back to the original state.

Returns: None

3.5.7.9 UltraGetMicIn

Function: Return the current state of the microphone input.

Syntax: `int UltraGetMicIn(void);`

Prototype: `gf1proto.h`

Remarks: This can be useful if you want to change the state of the microphone input and then restore it back to the original state.

Returns: None

3.5.8 Interrupt Handling Functions

The following 8 functions can be used to define a handler for events that happen under interrupt on the UltraSound. These are NOT interrupt handlers but are callbacks from the interrupt handler. This means that they should not be defined as interrupt type functions but must adhere to the general rules of interrupt code. No DOS calls should be made and care must be taken not to cause problems with code running in the foreground. All these functions return the old callback address so chaining could be done if desired. This not usually necessary. It is also not necessary to restore the handler back to the old one when exiting your application. However, **`UltraClose()`** MUST be called to restore the actual interrupt handler. These will only be called if the interrupt for the particular function is enabled in the mode parameter. (See the appropriate appendix for the bit definitions.) It is also not necessary to set up a callback since it has a default associated with it, but then why bother enabling the interrupt if you don't want to know when it finishes?

You should be able to make any Level 0 or Level 1 UltraSound library calls from within the interrupt handler. All the library functions protect themselves from being interrupted while doing critical operations.

3.5.8.1 UltraDramTcHandler

Function: Define a callback for DMA complete

Syntax:
```
PFV *(UltraDramTcHandler(handler));
    PFV *handler;
```

Prototype: `gf1proto.h`

Remarks: This function will be called when a DMA transfer to/from the UltraSound DRAM is complete. This can be called to start up another transfer.

Nothing is passed to this handler.

Returns: Old handler

3.5.8.2 UltraMidiXmitHandler

Function: Define a callback for MIDI xmit IRQ

Syntax:
```
PFV *(UltraMidiXmitHandler(handler));
    PFV *handler;
```

Prototype: `gf1proto.h`

Remarks: This function will be called on a MIDI Transmit Empty IRQ. This can be used to send out MIDI data under interrupt.

The MIDI status byte is passed to the handler. This is used to determine if there were an errors in transmission. The status bits are defined in Chapter 2.

Returns: Old handler

3.5.8.3 UltraMidiRecvHandler

Function: Define a callback for a MIDI recv IRQ

Syntax:
```
PFV *(UltraMidiRecvHandler(handler));
    PFV *handler;
```

Prototype: **gf1proto.h**

Remarks: This function will be called when a character is received in the MIDI input port. This can be used to get data from the MIDI port under interrupt. The MIDI port status and MIDI data are passed to your handler. The status bits are defined in Appendix D. Your handler should be defined like this:

```
PFV handler(status,data);
    unsigned char status;
    unsigned char data;
```

Returns: Old handler

3.5.8.4 UltraTimer1Handler

Function: Define a callback from Timer #1

Syntax:
```
PFV *(UltraTimer1Handler(handler));
    PFV *handler;
```

Prototype: **gf1proto.h**

Remarks: There are 2 general purpose timers available on the UltraSound. They are functionally identical to the Adlib timers. Nothing is currently passed to this handler.

Returns: Old handler

3.5.8.5 UltraTimer2Handler

Function: Define a callback for Timer #2

Syntax:
```
PFV *(UltraTimer2Handler(handler));
    PFV *handler;
```

Prototype: **gf1proto.h**

Remarks: There are 2 general purpose timers available on the UltraSound. They are functionally identical to the Adlib timers. Nothing is currently passed to this handler.

Returns: Old handler

3.5.8.6 UltraWaveHandler

Function: Define a callback for wavetable IRQ's

Syntax:
```
PFV *(UltraWaveHandler(handler));
    PFV *handler;
```

Prototype: **gf1proto.h**

Remarks: This function will be called when a voice generates a wavetable IRQ. This happens when a voice hits the end of its wave and interrupts are enabled. It will happen even if looping is on (i.e. the voice keeps playing). Normally, it would be used to signify that a voice is done playing. This IRQ can be useful for starting another voice or counting the # of times that a voice goes thru a loop before it is stopped. The voice # that generated the IRQ is passed back to your handler.

Returns: Old handler

3.5.8.7 UltraVolumeHandler

Function: Define a callback for volume ramp complete

Syntax:
```
PFV *(UltraVolumeHandler(handler));
    PFV *handler;
```

Prototype: **gf1proto.h**

Remarks: This function can be used to generate a volume envelope (attack, decay, sustain, release). This is done by changing to the appropriate volume ramps in the handler to handle the next part of the envelope. The voice # causing the IRQ will be passed back to your handler.

Returns: Old handler

3.5.8.8 UltraRecordHandler

Function: Define a callback for a DMA record complete

Syntax:
```
PFV *(UltraUltraRecordHandler(handler));
    PFV *handler;
```

Prototype: **gf1proto.h**

Remarks: This function is called when a buffer that was being recorded is full. Normally, this would be used to let the application start up another record. A double buffering scheme could be used to record data continuously.

Nothing is passed to this handler.

As long as the DMA channels for recording and playback are different, the UltraSound is capable of simultaneously recording and playback. At high data rates, throughput may be a problem that your application will have to deal with.

Returns: Old handler

3.5.9 UltraMaxAlloc

Function: Find the size of the largest allocatable block left

Syntax:
```
unsigned long UltraMaxAlloc(void);
```

Prototype: **gf1proto.h**

Remarks: This function will return the largest block of DRAM (in bytes) that can be allocated. This can be useful for determining whether or not a patch can be loaded. The maximum size of a block is 256K.

Returns: # of bytes in largest available block.

See also: **UltraMemAlloc**, **UltraMemFree**

3.5.10 UltraMemAlloc

Function: Allocate a chunk of UltraSound DRAM

Syntax:
```
int UltraMemAlloc(size,location);
    unsigned long size;
    unsigned long *location;
```

Prototype: **gf1proto.h**

Remarks: This function allocates a chunk of DRAM of ***size*** bytes from the UltraSound. The memory allocation structures are set up in **UltraOpen()**. A first-fit algorithm is used so it is up to the programmer to avoid too much memory fragmentation. ***location*** is filled in with the DRAM location of the start of the chunk of memory. The memory returned will ALWAYS be aligned on a 32 byte boundary so that the DRAM can be DMA'ed into. Also, the size will be rounded UP to the next 32 byte boundary.

Returns: **ULTRA_OK** == No Error
NO_MEMORY == No chunk of DRAM large enough

See also: **UltraMemInit**, **UltraMemFree**

3.5.11 UltraMemFree

Function: Free a chunk of UltraSound DRAM

Syntax:
```
int UltraMemFree(size,location)
    unsigned long size;
    unsigned long location;
```

Prototype: **gf1proto.h**

Remarks: Return a chunk of DRAM back to the memory pool. Be sure that the proper size chunk is returned so the memory structures are not corrupted. The size will automatically be rounded UP to the next 32 byte boundary.

Returns: **ULTRA_OK** == No problem
CORRUPT_MEM == Memory structs have been corrupted

See also: **UltraMemInit**, **UltraMemAlloc**

3.5.12 UltraMemInit

Function: Initialize memory free pool structures.

Syntax: `unsigned long UltraMemInit(void);`

Prototype: `gf1proto.h`

Remarks: This function sets up the UltraSound DRAM so that `UltraMemAlloc()` and `UltraMemFree()` will work. It is called in `UltraOpen()`. It can be called at any time if an application wants to clean up all its allocated or corrupted memory structures.

Returns: Number of K of DRAM found on the UltraSound. If an application wishes to reserve a chunk of DRAM outside of the memory pool, a variable called _ultra_reserved_dram must be set up with the # of bytes to reserve BEFORE `UltraMemInit()` is called. This reserved chunk will start at 0. The reserved chunk must be less than 256K.

See also: **UltraMemAlloc**, **UltraMemFree**

3.5.13 UltraMidiDisableRecv

Function: Disable midi receive data interrupt

Syntax: `void UltraMidiDisableRecv(void);`

Prototype: `gf1proto.h`

Remarks: This function will disable the receive data interrupts from the MIDI. This should be disabled before leaving your application.

Returns: None

See also: **UltraMidiEnableRecv**, **UltraMidiRecvhandler**

3.5.14 UltraDisableMidiXmit

Function: Disable midi transmit IRQs

Syntax: `void UltraDisableMidiXmit(void);`

Prototype: `gf1proto.h`

Remarks: This function will turn off MIDI transmit interrupts. It MUST be called when you are through sending data.

Returns: None

See also: **UltraMidiXmitHandler**, **UltraMidiEnableXmit**

3.5.15 UltraMidiEnableRecv

Function: Enable receive data interrupts for midi port

Syntax: `void UltraMidiEnableRecv(void);`

Prototype: `gf1proto.h`

Remarks: This function will enable receive data interrupts from the MIDI port. It is usually necessary to set up a callback function so your application can process the data.

Returns: None

See also: **UltraMidiRecvhandler**, **UltraMidiDisableRecv**

3.5.16 UltraEnableMidiXmit

Function: Enable transmit interrupts from MIDI

Syntax: `void UltraEnableMidiXmit(void);`

Prototype: `gf1proto.h`

Remarks: This function will enable transmit data interrupts to be generated as each byte is transmitted out the MIDI port. Note that a transmit IRQ will be generated as soon as the IRQ is enabled unless a character is sent out immediately prior to enabling it. This is because the xmit buffer is initially empty (unless primed) so it will pop an IRQ. Also note that you MUST disable this IRQ when you are not sending any more data or else you will be hung up getting transmit ready IRQs.

Returns: None

See also: **UltraMidiDisableXmit**

3.5.17 UltraMidiRecv

Function: Read a character from the midi port.

Syntax: `unsigned char UltraMidiRecv(void);`

Prototype: `gf1proto.h`

Remarks: This function is used to read a character from the MIDI port. It assumes that the character is waiting. The character is there if it got to the MIDI receive interrupt callback function or if you have polled the status and determined the receive buffer is full.

Returns: MIDI data

See also: **UltraMidiRecvHandler**, **UltraMidiStatus**

3.5.18 UltraMidiReset

Function: Reset the MIDI port.

Syntax: `void UltraMidiReset(void);`

Prototype: `gf1proto.h`

Remarks: This function should be used to ensure that the MIDI port is ready to use. All MIDI IRQs will be disabled by this call.

Returns: None

See also: **UltraEnableMidiXmit**, **UltraEnableMidiRecv**

3.5.19 UltraMidiStatus

Function: Read the MIDI status bits

Syntax: `unsigned char UltraMidiStatus(void)`

Prototype: `gf1proto.h`

Remarks: This function returns the current MIDI port status bits. See Chapter 2 for the definition of all the bits. This can be used to determine if an error has occurred or if the port is ready to be read or written.

Returns: The status bits. See Chapter 2.

See also: **UltraMidiXmit**, **UltraMidiRecv**

3.5.20 UltraMidiXmit

Function: Send a byte out the MIDI port.

Syntax:
```
void UltraMidiXmit(data)
    unsigned char data;
```

Prototype: **gf1proto.h**

Remarks: This function will send a byte out the MIDI data port. If IRQs are enabled, an interrupt will be generated when the character has been transmitted.

Returns: None

See also: **UltraMidiXmitHandler**

3.5.21 UltraOpen

Function: Open and initialize the lowlevel code.

Syntax:
```
#include "extern.h"
int UltraOpen(config,voices);
    ULTRA_CFG *config;
    int voices;        /* # of active voices (14-32) */
```

Prototype: **gf1proto.h**

Remarks: This function should ALWAYS be called to initialize the UltraSound. It will probe for the card and program the IRQ and dma latches. It will then disable line and microphone input and enable output. It also initializes the memory structures. The # of active voices is an important parameter to the UltraSound. The fewer the # of voices, the more oversampling that occurs on playback. That will make the sound much cleaner. However, if you need more voices to make more sounds, allocate them as necessary. In general, only make the minimum number active as possible (14 is the minimum).

Returns: **ULTRA_OK** == No problem
NO_ULTRA == No UltraSound card found
BAD_NUM_OF_VOICES == # of active voices not in range

See also: **UltraClose**, **UltraProbe**, **UltraMemInit**

3.5.22 UltraPeekData

Function: Examine any DRAM location on UltraSound

Syntax:
```
unsigned char UltraPeekData(baseport,location);
    unsigned int baseport;
    unsigned long location;
```

Prototype: **gf1proto.h**

Remarks: This function is used to allow an application to look at any location in UltraSound's DRAM whenever it wishes. This can be handy for obtaining VU information or any other time it is nice to know what is in DRAM. Be aware that the data will probably be in twos compliment form. The data MUST be in twos compliment form for the UltraSound to play it. If your original data was in one's compliment, it was probably converted when it was DMA'ed down to the UltraSound.

If the data that you want is 16 bit data, be sure and peek both locations and do any appropriate conversions. The data will be in low/high format. That means that the low byte of the data will be in the even byte and the high byte will be in the odd byte.

Returns: Data byte at ***location***.

See also: **UltraPokeData, UltraDownload**

3.5.23 UltraPing

Function: Quick check to see if UltraSound is present

Syntax:
```
int UltraPing(baseport)
    int baseport;
```

Prototype: **gf1proto.h**

Remarks: This function will determine if an UltraSound is present by attempting to read and write to its DRAM. This function assumes that at least a simple reset has been done since powerup so that the board is no longer in a reset state. If it is on a reset state, this function will ALWAYS fail. **`UltraProbe()`** will pull a quick reset and then call **`UltraPing()`**.

Returns: **`ULTRA_OK`** == Found a board.
`NO_ULTRA` == No board at this I/O location

See also: **UltraProbe**

3.5.24 UltraPokeData

Function: Poke a byte into UltraSound DRAM

Syntax:
```
void UltraPokeData(baseport,location,data);
    unsigned int baseport;
    unsigned long location;
    unsigned char data;
```

Prototype: **gf1proto.h**

Remarks: Poke an 8 bit value directly into UltraSound DRAM. This can be useful to set the value of the location that a voice is pointing to. It is often desirable to point a voice to a known value since its output is ALWAYS summed in to the output even if the voice is not running. Be aware that there is no conversion of the data that is being poked into DRAM. Since the UltraSound only plays two's compliment data, make sure that the data you are poking is in that format. Also be careful with 16 bit data.

Returns: None

See also: **UltraPeekData**, **UltraDownload**

3.5.25 UltraPrimeRecord

Function: Prime the record DMA channel

Syntax:
```
int UltraPrimeRecord(pc_ptr,size,repeat);
    void far *pc_ptr;
    unsigned int size;
    int repeat;
```

Prototype: **gf1proto.h**

Remarks: This function will setup the DMA channel to do a record, but do not start it. This can be used to help synchronize events. Programming the DMA channel can take enough time so that a few samples may be lost. (Depending on sample rate). This function will help alleviate this problem by doing the programming ahead of time.

Returns: **ULTRA_OK** == No problem
DMA_BUSY == Dma channel busy.

It may be busy doing a download or upload or a record (if play and record channels are the same).

3.5.26 UltraPrimeVoice

Function: This function primes a voice with the pertinent values but does NOT start it.

Syntax:
```
unsigned char UltraPrimeVoice(voice,begin,start,end,mode)
    int voice;           /* voice to prime */
    unsigned long begin; /* begin loc in DRAM */
    unsigned long start; /* start loop loc in DRAM */
    unsigned long end;   /* end location in DRAM */
    unsigned char mode;  /* voice mode (loop etc) */
```

Prototype: **gf1proto.h**

Remarks: This function is used to do all the setup necessary to start a voice but does NOT start it up. This can be useful if you want to start more than 1 voice at the same time. Use **UltraPrimeVoice()** to do all the necessary setup and then use **UltraGoVoice()** to start the voices. **UltraStartVoice()** calls **UltraPrimeVoice()** and the **UltraGoVoice()**.

Returns: An updated mode value. The mode may be modified on the basis of the location parameters. This altered value should be the one passed to **UltraGoVoice()**.

See also: **UltraStartVoice**, **UltraGoVoice**

3.5.27 UltraProbe

Function: Probe for the existence of UltraSound

Syntax:
```
int UltraProbe(baseport);
    unsigned int baseport;
```

Prototype: **gf1proto.h**

Remarks: This function probes for the existence of an UltraSound card at the specified base port. An application could call this before calling UltraOpen() to see if a card is present. **UltraOpen()** calls this routine also. The difference between **UltraProbe()** and **UltraPing()** is that **UltraProbe()** will pull a reset to make sure the board is running. **UltraPing(void)** assumes this has already been done.

Returns: **ULTRA_OK** == No problem
NO_ULTRA == No card found at this base port

See also: **UltraOpen**, **UltraPing**

3.5.28 UltraRampVolume

Function: Ramp a voice's volume up or down.

Syntax:
```
void UltraRampVolume(voice,start,end,rate,mode)
    int voice;          /* voice to use */
    unsigned int start; /* start value (0-511) */
    unsigned int end;   /* end value (0-511) */
    unsigned char rate; /* range/rate value to stuff */
    unsigned char mode; /* volume ramp mode (loop etc) */
```

Prototype: **gf1proto.h**

Remarks: Start a volume ramp from a starting volume to the ending volume. The rate at which the ramp occurs can be calculated using **UltraCalcRate()**. The mode determines the looping and interrupting characteristics of the ramp. If you choose to have it interrupt at the end of the ramp, you should set up an function to call for a volume interrupt. This is done with the function **UltraVolumeHandler()**. The mode bits are defined in Appendix B.

Returns: None

See also: **UltraCalcRate**, **UltraVolumeHandler**

3.5.29 UltraReadRecordPosition

Function: Return the # of bytes recorded so far.

Syntax:
```
unsigned int UltraReadRecordPosition(void);
```

Prototype: **gf1proto.h**

Remarks: This function can be used to monitor the amount data recorded so far.

Returns: Number of bytes recorded so far.

See also: **UltraRecordData**

3.5.30 UltraReadVoice

Function: Read a voice's current location.

Syntax:
```
unsigned long UltraReadVoice(voice)
    int voice;
```

Prototype: **gf1proto.h**

Remarks: This routine returns the absolute position of this voice in DRAM

Returns: Current position in DRAM

See also: **UltraSetVoice**

3.5.31 UltraReadVolume

Function: Read a voice's current volume.

Syntax:
```
unsigned int UltraReadVolume(voice)
    int voice;
```

Prototype: **gf1proto.h**

Remarks: This routine returns the current volume of this voice. This can be useful when used in conjunction with volume ramps. The value returned is NOT in linear units (0-511).

Returns: Current volume value (0-4095)

See also: **UltraSetVolume**, **UltraRampVolume**

3.5.32 UltraRecordData

Function: Record some data with the UltraSound

Syntax:
```
int UltraRecordData(pc_ptr,control,size,wait,repeat)
    void far *pc_ptr;
    unsigned char control;
    unsigned int size;
    int wait;
    int repeat;
```

Prototype: **gf1proto.h**

Remarks: This function will record a buffer of data from UltraSound. It can be in either 8-bit mono or stereo. In stereo, the left byte is first then the right. If mono is being used, the left channel is the one that is sampled. See Appendix E for the a description of the recording control bits. If wait is set to a non-zero value, then this function will not return until the buffer has been filled. If repeat is set to a non-zero value, then the DMA channel will be set up in autoinit mode so that the recording is done indefinitely. If this is done, then the buffer MUST reside completely in 1 64K page of PC RAM. Also, it is probably necessary that your application hooks to the record handler so that the control register on the UltraSound can be hit to restart the recording process (**UltraGoRecord()**). This will be very quick since the PC DMA controller will not be re-programmed.

Returns: **ULTRA_OK** == No error
DMA_BUSY == Sampling DMA channel is busy.
BAD_DMA_ADDR == Autoinit Buffer crosses 64K page.

See also: **UltraWaitRecordDma**, **UltraSetRecordFrequency**, **UltraGoRecord**

3.5.33 UltraRecordDmaBusy

Function: See if record DMA channel is busy

Syntax:
```
int UltraRecordDmaBusy(void);
```

Prototype: **gf1proto.h**

Remarks: This function checks to see if the record DMA channel is busy. It might be busy doing a record or playback. (If the record & playback channels are the same).

Returns: **ULTRA_OK** == No error
DMA_BUSY == This channel is not free to use yet.

See also: **UltraRecordData**

3.5.34 UltraReset

Function: Pull a full reset on the UltraSound

Syntax:
```
int UltraReset(voices)
    int voices;
```

Prototype: **gf1proto.h**

Remarks: This function is called by **UltraOpen()** to make sure the card is in a known state. **UltraClose()** also calls this function.

Returns: **ULTRA_OK** == No problem
BAD_NUM_OF_VOICES == # of voices not in range

See also: UltraOpen, UltraClose

3.5.35 UltraSetBalance

Function: Set a voice's pan position

Syntax:
```
void UltraSetBalance(voice,data)
    int voice;
    int data;  /* (0-15) */
```

Prototype: **gf1proto.h**

Remarks: This function sets the voice's position somewhere between right and left. A 0 will place the audio all the way to the left. A 15 will put the sound all the way to the right.

Returns: None

3.5.36 UltraSetFrequency

Function: Set a voice's playback frequency

Syntax:
```
void UltraSetFrequency(voice,speed_hz)
    int voice;
    unsigned long speed_hz;
```

Prototype: **gf1proto.h**

Remarks: This function sets the voice's playback rate to the specified rate. The number of active voices is taken into account when making the appropriate calculations.

Returns: None

3.5.37 UltraSetLoopMode

Function: Set a voice's loop mode

Syntax:
```
void UltraSetLoopMode(voice,mode)
    int voice;
    unsigned char mode;
```

Prototype: **gf1proto.h**

Remarks: This function will set this voice's looping mode to the specified mode. See Appendix C for the definition of these bits.

Returns: None

3.5.38 UltraSetRecordFrequency

Function: Set the recording rate.

Syntax:
```
void UltraSetRecordFrequency(rate)
    unsigned long rate; /* in hertz */
```

Prototype: **gf1proto.h**

Remarks: This function sets the record rate to the specified rate. Since the UltraSound uses the PC DMA channel to do the sampling directly into PC RAM, no voice is specified.

Returns: None

3.5.39 UltraSetVoice

Function: Set a voice to an absolute position

Syntax:
```
void UltraSetVoice(voice,location)
    int voice;              /* voice to set */
    unsigned long location; /* location in ultra DRAM */
```

Prototype: **gf1proto.h**

Remarks: This function sets a voice's current position to an absolute location. This can be useful to set a voice to a location with a known value since all voices current locations are summed in to the output even if the voice is not running. 'Pops' in the audio may result if a voice is set to a location that is not proper.

Returns: None

3.5.40 UltraSetVoiceEnd

Function: Set a voice's end position.

Syntax:
```
void UltraSetVoiceEnd(voice,end)
    int voice;          /* voice to start */
    unsigned long end; /* end location in ultra DRAM */
```

Prototype: **gf1proto.h**

Remarks: This function sets a new endpoint for the specified voice. Used in conjunction with **UltraSetLoopMode()** to turn looping off, a sampled decay can be implemented.

Returns: None

See also: **UltraSetLoopMode**

3.5.41 UltraSetVolume

Function: Set a voice's current volume.

Syntax:
```
void UltraSetVolume(voice,volume)
    int voice;
    unsigned int volume;  /* (0-4095) */
```

Prototype: **gf1proto.h**

Remarks: This function sets the volume of the voice to a specific value. The range is from 0 to 4095 and is logarithmic, not linear. **UltraSetLinearVolume()** can do that.

Returns: None

See also: **UltraSetLinearVolume**

3.5.42 UltraSizeDram

Function: Find the amount of DRAM on UltraSound

Syntax:
```
int UltraSizeDram(void)
```

Prototype: **gf1proto.h**

Remarks: This function could be used by an application to determine how much it can load into the UltraSound. **UltraMemInit()** calls this to determine how much can be used for its memory pool.

Returns: # of K found on the UltraSound.

See also: **UltraMemInit**

3.5.43 UltraStartTimer

Function: Start up a Timer

Syntax:
```
void UltraStartTimer(timer,duration)
    int timer;              /* 1 or 2 */
    unsigned char duration; /* 0 - 255 ticks */
```

Prototype: **gf1proto.h**

Remarks: This function can be used to start up one of two available hardware timers. Timer #1 is an 80 microsecond timer. Timer #2 is a 320 microsecond timer. You supply the # of ticks before the timer stops. When the timer stops, it ALWAYS calls the callback function defined for it. If you don't supply one, a default is used. This can be used to trigger various real time events. Please remember that the callback routine is called directly from the interrupt handler, so you must be careful what you do in the callback routine. Note that the timer is automatically restarted after it ticks. Your application must explicitly call **UltraStopTimer()** to shut it off.

Returns: None

See also: **UltraStopTimer**, **UltraTimerStopped**, **UltraTimer1Handler**, **UltraTimer2Handler**.

3.5.44 UltraStartVoice

Function: Start a voice playback.

Syntax:
```
void UltraStartVoice(voice,begin,start,end,mode)
    int voice;           /* voice to start */
    unsigned long begin; /* start loc in DRAM */
    unsigned long start; /* start loop loc in DRAM */
    unsigned long end;   /* end location in DRAM */
    unsigned char mode;  /* mode to run voice (loop etc) */
```

Prototype: **gf1proto.h**

Remarks: This function will start up a voice. The voice will begin playback at the ***begin*** and continue to ***end***; if looping is on, then it will loop back to ***start*** (or play backwards to ***start***) and then continue to ***end***. The method of looping is determined by ***mode***. See Appendix C for the definition of these bits.

NOTE: To play a sample backwards, ***begin*** should be set to address that you want it to start at, ***start*** must be a lower address than the ***end***. Now the sample will begin playing at ***begin***, with decreasing addresses thru the end location until it hits the ***start*** location. It will then loop back to the ***end*** position if looping is enabled.

Returns: None

See also: **UltraSetEnd**,**UltraLoopMode**

3.5.45 UltraStopTimer

Function: Stop one of the 2 timers

Syntax:
```
void UltraStopTimer(timer)
    int timer;  /* 1 or 2 */
```

Prototype: **gf1proto.h**

Remarks: This function shuts off a particular timer.

Returns: None

See also: **UltraStartTimer**

3.5.46 UltraStopVoice

Function: Stop a voice.

Syntax:
```
void UltraStopVoice(voice)
    int voice;
```

Prototype: **gf1proto.h**

Remarks: This function is used to stop a currently playing voice.

Returns: None

See also: **UltraStartVoice**,**UltraSetEnd**

3.5.47 UltraStopVolume

Function: Stop a volume ramp.

Syntax:
```
void UltraStopVolume(voice)
    int voice;
```

Prototype: **gf1proto.h**

Remarks: This function is used to stop a currently active volume ramp.

Returns: None

See also: **UltraRampVolume**, **UltraRampLinearVolume**

3.5.48 UltraTimerStopped

Function: Check to see if a timer is not running any more.

Syntax:
```
int UltraTimerStopped(timer)
    int timer;
```

Prototype: **gf1proto.h**

Remarks: This function will return non-zero if the timer is stopped.

Returns: 0 == still running.
non-zero == stopped.

See also: **UltraStartTimer**, **UltraStopTimer**

3.5.49 UltraTrimJoystick

Function: Set the trim voltage on the joystick port

Syntax:
```
void UltraTrimJoystick(value);
    unsigned char value;
```

Prototype: **gf1proto.h**

Remarks: This function is used to set the speed compensation value on the joystick port on the UltraSound. The faster the computer, the smaller this value should be. This allows all software that reads the joystick to return consistent joystick positions regardless of the speed of the machine. This is normally not needed and probably should never be used in your application. The utility **Ultrajoy.exe** is used to set this up in your **autoexec.bat**.

Returns: None

3.5.50 UltraUpload

Function: Dma a section of UltraSound DRAM into PC Ram

Syntax:
```
int UltraUpload(dataptr,control,dram_loc,len,wait)
    void *dataptr;          /* ptr to buffer in PC RAM */
    unsigned char control;  /* control bits */
    unsigned long dram_loc; /* location in UltraSound */
    unsigned int len;       /* # of bytes to read.  */
    int wait;               /* wait for completion */
```

Prototype: **gf1proto.h**

Remarks: This function will retrieve a chunk of data from the UltraSound's DRAM. It will transfer ***len*** # of bytes to ***dataptr*** (in PC) from ***dram_loc*** (in UltraSound). If ***wait*** is **TRUE**, then it will wait until the transfer is complete. If ***wait*** is **FALSE**, it will return as soon as transfer is started. See Appendix D for a definition of the control bits. They specify the type of data being retrieved.

Returns: **ULTRA_OK** == No problem
DMA_BUSY == Dma channel busy.

It may be busy doing a download or upload or a record (if play and record channels are the same).

See also: **UltraDownload**, **UltraDramDmaWait**

3.5.51 UltraVectorVolume

Function: Ramp a volume from here to new end point

Syntax:
```
void UltraVectorVolume(voice,end,rate,mode)
    int voice;
    unsigned int end; /* end volume to go to */
    unsigned char rate;
    unsigned char mode; /* mode to run the vol ramp in */
```

Prototype: **gf1proto.h**

Remarks: This function can be used to ramp from an unknown place to a new place. It is useful if you are doing volume envelopes and need to restart the attack/decay sequence at any time.

Returns: None

See also: **UltraRampVolume**, **UltraStopVolume**

3.5.52 UltraVersion

Function: Return the version of the low level code

Syntax:
```
void UltraVersion(major,minor);
    int *major;
    int *minor;
```

Prototype: **gf1proto.h**

Remarks: This function can be used to track the version of the low level code being used.

Returns: Major and minor version of low level code.

3.5.53 UltraVoiceStopped

Function: Return the state of a voice.

Syntax:
```
int UltraVoiceStopped(voice);
    int voice;
```

Prototype: **gf1proto.h**

Remarks: This function is used to determine the current state of the voice. It can be used to see if a wave has finished yet.

Returns: 0 == Voice is running
non-zero == Voice is stopped

3.5.54 UltraVolumeStopped

Function: Determine if volume ramp is running

Syntax:
```
int UltraVolumeStopped(voice);
    int voice;
```

Prototype: **gf1proto.h**

Remarks: This function is used to determine the current state of the volume of the voice. It can be used to see if a volume ramp has finished yet.

Returns: 0 == Volume ramp is running
non-zero == Volume ramp is stopped

See also: **UltraRampVolume**, **UltraRampLinearVolume**

3.5.55 UltraWaitDramDma

Function: Wait for a DRAM DMA transfer to complete.

Syntax:
```
void UltraWaitDramDma(void)
```

Prototype: **gf1proto.h**

Remarks: If a DMA transfer to/from DRAM is started but told not to wait for it to complete, this function can be used to let the application wait for it to complete.

Returns: None

See also: **UltraDownload**, **UltraUpload**

3.5.56 UltraWaitRecordDma

Function: Wait for a Recording DMA transfer to complete.

Syntax: **`void UltraWaitRecordDma(void)`**

Prototype: **`gf1proto.h`**

Remarks: This can be used to let an application wait until a complete sample is finished being acquired.

Returns: None

See also: **UltraRecordData**

3.6 Level 1 Interface Functions

3.6.1 UltraAllocVoice

Function: Allocate a voice to use.

Syntax:
```
int UltraAllocVoice(voice_num,new_num)
    int voice_num;
    int *new_num;
```

Prototype: **`gf1proto.h`**

Remarks: This function will return a voice for your application to use. If you supply a voice number, it will attempt to allocate that particular voice. If you pass a -1 for the voice number, it will return the next free voice. This function will only allocate voices up to the # of active voices specified in the **`UltraOpen(void)`** function. The voice number will be returned in new_num.

Returns: **`ULTRA_OK`** == Got a voice
`VOICE_NOT_FREE` == Can't get the voice or none left.
`VOICE_OUT_OF_RANGE` == Specified voice out of range.

See also: **UltraClearVoices**, **UltraFreeVoices**

3.6.2 UltraClearVoices

Function: Reset all voice to un-allocated.

Syntax: **`void UltraClearVoices(void)`**

Prototype: **`gf1proto.h`**

Remarks: This function will un-allocate all previously allocated voices. It would be advisable to call this before using either **`UltraAllocVoice()`** or **`UltraFreeVoice()`**.

Returns: None

See also: **UltraAllocVoice**, **UltraFreeVoice**

3.6.3 UltraFreeVoice

Function: Free up an allocated voice.

Syntax:
```
void UltraFreeVoice(voice_num)
    int voice_num;
```

Prototype: **gf1proto.h**

Remarks: This function will free up a previously allocated voice. This should be used when your application no longer needs the voice so it can be re-allocated at another time.

Returns: None

See also: **UltraClearVoices**, **UltraAllocVoice**

3.6.4 UltraVoiceOff

Function: Turn a voice off & let it decay.

Syntax:
```
void UltraVoiceOff(voice,end)
    int voice;
    int end;
```

Prototype: **gf1proto.h**

Remarks: This function will either stop a voice immediately or let it finish its current loop. if ***end*** is equal to 0 then it will stop abruptly, else it will finish the current loop. **UltraReadVoice()** could be called after if you need to know when the loop finished. If used with **UltraSetVoiceEnd()**, you could have a sampled decay on the end of your sample. This would occur if your loop point was not at the end of your data, and you changed the end point to the real end point of your data and then called this function with ***end*** not equal 0.

Returns: Nothing

See also: **UltraReadVoice**, **UltraSetVoiceEnd**

3.6.5 UltraVoiceOn

Function: Turn a voice on at a given frequency.

Syntax:
```
void UltraVoiceOn(voice,begin,s_loop,e_loop,control,freq)
    int voice;
    unsigned long begin; /* initial start location */
    unsigned long s_loop; /* start of loop */
    unsigned long e_loop; /* end of loop */
    unsigned char control;
    unsigned long freq;
```

Prototype: **gf1proto.h**

Remarks: This function just sets the frequency and calls **UltraStartVoice()**.

Returns: None

See also: **UltraSetFrequency**, **UltraStartVoice**

3.6.6 UltraSetLinearVolume

Function: Set the volume to a linearized value.

Syntax:
```
void UltraSetLinearVolume(voice,index)
    int voice;
    int index;
```

Prototype: **gf1proto.h**

Remarks: This function indexes into a table to translate a linear volume (0-511) to a logarithmic volume (0-4095).

Returns: None

See also: **UltraSetVolume**

3.6.7 UltraRampLinearVolume

Function: Ramp between linear volume settings

Syntax:
```
void UltraRampLinearVolume(voice,start,end,msecs,mode)
    int voice;
    unsigned int start;  /* start linear vol (0-511) */
    unsigned int end;    /* end linear vol (0-511) */
    unsigned long msecs; /* # of millisecs to run ramp */
    unsigned char mode;  /* mode to run the vol ramp in */
```

Prototype: **gf1proto.h**

Remarks: This function is used to ramp between 2 linear volume settings. It uses the same method as **UltraSetLinearVolume()** to determine the actual volume settings to use.

Returns: None

See also: **UltraRampVolume**, **UltraSetLinearVolume**

3.6.8 UltraVectorLinearVolume

Function: Ramp a linear volume from here to end point.

Syntax:
```
void UltraVectorLinearVolume(voice,end,rate,mode)
    int voice;
    unsigned int end;   /* end ramp volume */
    unsigned char rate;
    unsigned char mode; /* mode to run the vol ramp in */
```

Prototype: **gf1proto.h**

Remarks: This function can be used to ramp from an unknown place to a new place. It is useful if you are doing volume envelopes and need to restart the attack/decay sequence at any time. This uses linear volume entries rather than logarithmic. It can also produce a smoother ramp from one volume to another. Arbitrarily setting a volume that is far away from the current volume can cause 'pops'.

Returns: None

See also: **UltraSetLinearVolume**.

Chapter 4 - Focal Point 3D Sound

Three dimensional audio on the Ultrasound is achieved by a technique called binaural representation. Basically, this means that a mono sound is 'shaped' in such a way that when it is presented to the right and left ears properly, the sound seems to come from the proper place in space. The technique of shaping the sound is called convolution. This is done thru algorithms developed by Focal Point(tm) 3D Audio. Focal Point has provided a utility (called fp3d.exe) to convert a mono sound file to a 3D file capable of being played on an UltraSound. The basic concept is that the mono sound is processed in such a way that the output file contains up to 6 tracks of sound. When the sound is played back, the volumes and balances are adjusted to make the sound appear to originate from anywhere in the 3D space. It is possible to create files that only have 4 tracks and can therefore only be positioned in 2 dimensions. This may be adequate for many applications and it will make the resultant file smaller.

There are 2 types of 3D sound that an application might want to use. The first is a sound effect. This is a sound that can be completely loaded into DRAM and played and positioned at any time. It is used for things like gunshots, cars etc. These can be looped sounds or 'one shots'. The other type of sound is a sound track. This is for a very long sound that cannot be loaded in DRAM all at once. The implementation of each of these methods is very different. A sound effect is implemented using a blocked data format. This means that each track's data is in a block of its own. A sound track uses interleaved data. This means that all the track's data is interleaved together. For example, a blocked file would look like this:

HEADER FFFFFFFFFFRRRRRRRRRRBBBBBBBBBBLLLLLLLLLL

An interleaved file would look like this:

HEADER FRBLFRBLFRBLFRBLFRBLFRBLFRBLFRBLFRBLFRBL

where:

F - Front Track
R - Right Track
B - Behind Track
L - Left Track
U - Up Track
D - Down Track

Each of these methods has its advantages and disadvantages. First, the interleaved method makes it very easy to read in the data in a continuous stream since it will look the same all the way through the file. It is also faster to read 60K of data once rather than 6 10K reads. This makes it very useful for a sound track. However, one draw back is that it has a very limited number of frequencies that it will run at. The reason for this is rather difficult to explain, but it pertains to getting the UltraSound's voices to play every 4th (or 5th, etc) sample. The UltraSound normally would interpolate between data points but it can't do that here because the adjacent data points are not in the same track. If you want to run at 22050 Hz, then you need to have only 28 active voices to accomplish this. If you want to play a track at 44100 Hz, you must have only 14 active voices. Blocked data doesn't have this problem. Since the data is NOT interleaved, but in a block of contiguous DRAM, its frequency can be adjusted to any value. This is usually very useful for sound effects (rev'ing engines etc). Also, since the data is blocked, each track's data can be allocated separately and can therefore be up to 256K. An interleaved file must fit into 1 256K bank since we cannot play 16bit data across a 256K bank. See the two example 3D programs provided (**`play3d.c`**, **`play3di.c`**) for ideas on how to implement 3D sound into you application.

4.1 Creating A 3D File

Please read the **`readme.3d`** file for this info.

4.2 3D Sound Functions

These are the functions currently available to playback a 3D sound file.

4.2.1 UltraAbsPosition

Function: Place a sound in 3D space (Cartesian).

Syntax:
```
void UltraAbsPosition3d(sound,xpos,ypos,zpos)
    SOUND_3D *sound; /* Handle for the 3D sound */
    int xpos;        /* -511 (left) to +511 (right) */
    int ypos;        /* -511 (below) to +511 (above) */
    int zpos;        /* -511 (behind) to +511 (ahead) */
```

Prototype: **threed.h**

Remarks: This function will position a sound using standard cartesian coordinates. The X position is right/left. The Y position is up/down. The Z position is ahead/behind. X and Z determine the azimuth position and Y determines the elevation. The distance away from you is determined using trigonometry . Once these are determined, **UltraAngPosition3d()** is called. Be careful when using this function, since the trig functions will be needed from the C libraries. If the distance is calculated to be greater than 511, it is clipped to 511. Also, if the distance is calculated to be 0, no positioning is done since the origin is undefined. (i.e. A sound cannot be generated INSIDE your head (origin)).

Returns: None

See also: **UltraAngPosition3d**

4.2.2 UltraAngPosition3d

Function: Position a sound using polar coordinates

Syntax:
```
void UltraAngPosition3d(sound,azimuth,elevation,volume)
    SOUND_3D *sound; /* Handle */
    int azimuth;     /* Horiz plane (-180 to +180 deg) */
    int elevation;   /* Vert plane (-90 to +90 deg) */
    int volume;      /* volume level at this position */
```

Prototype: **threed.h**

Remarks: This function will position a 3D sound using polar coordinates. The azimuth and elevation is specified in degrees. ***azimuth*** is the angle in the horizontal plane. Straight ahead is 0 degrees, 90 degrees is to the right, -90 degrees is to the left and 180 (or -180) is directly behind you. If an angle larger than 180 or smaller that -180 is specified, it is converted to its -180 to 180 equivalent. For example, 270 degrees is equivalent to -90 degrees. ***elevation*** is the angle of elevation above or below the horizontal plane. 0 degrees is no elevation, 90 degrees is straight up, and -90 is straight down. Any angle larger than 90 or smaller that -90 is ignored.

If more precision is needed, use **UltraAngFltPosition3D()** so that floating point arithmetic is used. Since that function uses floating point functions, your application may not want to use it.

Returns: None

See also: **UltraAbsPosition3d**

4.2.3 UltraAngFltPosition3d

Function: Position a sound using polar coordinates.

Syntax:
```
void UltraAngFltPosition3d(sound,azimuth,elevation,volume)
    SOUND_3D *sound;  /* Handle */
    double azimuth;   /* Horiz plane (+/-180.0 deg) */
    double elevation; /* Vert plane (+/-90.0 deg) */
    int volume;       /* volume level at this position */
```

Prototype: **threed.h**

Remarks: This function will position a 3D sound using polar coordinates. The azimuth and elevation is specified in as a floating point number in degrees. For example, 45 and one half degrees would be specified as 45.5. ***azimuth*** is the angle in the horizontal plane. Straight ahead is 0 degrees, 90 degrees is to the right, -90 degrees is to the left and 180 (or -180) is directly behind you. If an angle larger than 180 or smaller that -180 is specified, it is converted to its -180 to 180 equivalent. For example, 270 degrees is equivalent to -90 degrees. ***elevation*** is the angle of elevation above or below the horizontal plane. 0 degrees is no elevation, 90 degrees is straight up, and -90 is straight down. Any angle larger than 90 or smaller that -90 is ignored.

If your application does not wish (or need) to use the floating point function, use **UltraAngPosition()**.

4.2.4 UltraLoad3dEffect

Function: Load a 3D effect into DRAM

Syntax:
```
int UltraLoad3dEffect(sound,filename,pc_buffer,size)
    SOUND_3D *sound;
    char *filename;
    void far *pc_buffer;
    unsigned int size;
```

Prototype: **threed.h**

Remarks: This function is used to load a 3D sound into the DRAM on the UltraSound. It will allocate all necessary resources so that **UltraStart3D()** can be called later. As many buffers of DRAM as needed will be allocated along with the proper number of voices. These resources will be freed up when you call **UltraUnLoad3dEffect(void)**. The ***filename*** MUST be a properly formatted file. This means that it must have the proper header and 3D data in it. The ***pc_buffer*** is a buffer supplied by your application for this functions use to download data into the UltraSound. The ***size*** parameter is the size of the ***pc_buffer***. The larger this buffer is, the faster it will download.

Returns:
ULTRA_OK == no error
NO_3D_FILE == 3D file not found.
BAD_3D_HDR == 3D header is bad.
NO_3D_HDR == File doesn't have a valid header.
NOT_BLOCK_DATA == File not in blocked format. (interleaved)
NO_FREE_VOICES == Not enough free voices
BAD_FILE_DATA == Not enough data. (Hdr wrong or bad file.)
CORRUPT_MEM == Memory structs have been corrupted
DMA_BUSY == Dma channel busy.

See also: **UltraMemAlloc**, **UltraVoiceAlloc**, **UltraDownload**

4.2.5 UltraSetFreq3D

Function: Set Frequency of 3D sound

Syntax:
```
void UltraSetFreq3D(sound,frequency)
    SOUND_3D *sound;
    unsigned long frequency;
```

Prototype: **threed.h**

Remarks: This function will allow you to alter the frequency that the 3D sound is using. This allows you to do pitch shifting to get a doppler shift type effect. This function can only be done on blocked data, not interleaved.

Returns:
ULTRA_OK == no problem
NOT_BLOCKED_DATA == Can't change freq of interleaved data.

4.2.6 UltraRelease3dInterleave

Function: Release Interleaved 3D resources

Syntax:
```
void UltraRelease3dInterleave(sound)
    SOUND_3D *sound;
```

Prototype: **threed.h**

Remarks: This functions will release the DRAM and voices allocated for this interleaved sound track.

Returns: None

See also: **UltraSetup3dInterleave**

4.2.7 UltraSetup3dInterleave

Function: Setup for an interleaved 3D sound

Syntax:
```
int UltraSetup3dInterleave(sound,filename,size)
    SOUND_3D *sound;
    char *filename;
    unsigned long size;
```

Prototype: **threed.h**

Remarks: This function will allocate the voices and memory necessary to playback an interleaved 3D sound. It will NOT load any of the file data into DRAM. Your application is responsible for that. It will open the file and read the header and allocate the appropriate resources and setup the begin,start and end loop points for each voice. Since they are interleaved, the loop points are staggared appropriately.

Returns:
ULTRA_OK == no error
NO_3D_FILE == 3D file not found.
BAD_3D_HDR == 3D header is bad.
NO_3D_HDR == File doesn't have a valid header.
NOT_INTERLEAVED_DATA == File in blocked format.
NO_FREE_VOICES == Not enough free voices
BAD_FILE_DATA == Not enough data. (Hdr wrong or bad file)
CORRUPT_MEM == Memory structs have been corrupted

See also: **UltraLoad3dEffect**

4.2.8 UltraStart3d

Function: Start a 3D sound.

Syntax:
```
void UltraStart3d(sound)
    SOUND_3D *sound;
```

Prototype: **threed.h**

Remarks: This function will start a 3D sound. If you want it to begin at a specific point in space, be sure that you position it first. It is not necessary to stop the sound before starting it again.

Returns: None

See also: **UltraStart3d**

4.2.9 UltraStop3d

Function: Stop a 3d sound.

Syntax:
```
void UltraStop3D(sound,abruptly)
    SOUND_3D *sound;
    int abruptly;
```

Prototype: **threed.h**

Remarks: This function will stop a 3D sound. There are two ways to stop a sound. If the ***abruptly*** flag is **TRUE**, then the sound will shut off immediately. If it is **FALSE**, then the sound will be ramped down very quickly. The second method will give a smoother transition.

Returns: None

4.2.10 UltraUnLoad3dEffect

Function: Free up a 3D effect resources.

Syntax:
```
void UltraUnLoad3dEffect(sound)
    SOUND_3D *sound;
```

Prototype: **threed.h**

Remarks: This function is used to free up all the resources (voices & DRAM) that a 3D sound uses. Since a 3D sound can use a lot of the voices & DRAM, you should free up the resources whenever you can.

Returns: None

See also: **UltraLoad3dEffect**

Appendix A - Error Codes

These are defined in **ultraerr.h**

#define ULTRA_OK	**1**	No error
#define BAD_NUM_OF_VOICES	**2**	must be 14-32
#define NO_MEMORY	**3**	Not enough free DRAM left
#define CORRUPT_MEM	**4**	memory structures are corrupt
#define NO_ULTRA	**5**	Can't find an UltraSound
#define DMA_BUSY	**6**	This DMA channel is still busy
#define BAD_DMA_ADDR	**7**	auto init across page boundaries
#define VOICE_OUT_OF_RANGE	**8**	allocate a voice past # active
#define VOICE_NOT_FREE	**9**	voice has already been allocated
#define NO_FREE_VOICES	**10**	not any voices free

Error codes for 3D functions. These are defined in **threed.h**.

#define NO_3D_FILE	**101**	Can't open file
#define BAD_3D_HDR	**102**	Header for file is corrupt/non-existent
#define NO_DRAM_3D	**103**	Not enough room for this sound
#define NO_3D_HDR	**104**	No header on this file
#define NOT_BLOCK_DATA	**105**	data not in block format
#define BAD_FILE_DATA	**106**	not enough data as header says
#define NOT_INTERLEAVED_DATA	**107**	data not in interleaved format
#define BAD_3D_FREQ	**108**	nvalid freq. for interleaved 3D data

Appendix B - Volume Control Bits

Here are the volume ramp control bit definitions:

Bit 7 (0x80) **Reserved**

Bit 6 (0x40) **Reserved**

Bit 5 (0x20) **1 = Volume Ramp IRQ Enable**

Bit 4 (0x10) **1 = Bidirectional Enable**

Bit 3 (0x08) **1 = Loop Enable**

Bit 2 (0x04) **Reserved**

Bit 1 (0x02) **Reserved**

Bit 0 (0x01) **Reserved**

This specifies the bits that should be set by the application to get a particular type of volume ramp. This should be supplied to **`UltraRampVolume()`** and **`UltraRampLinearVolume()`**. If IRQ's are enabled, make sure that you have set up a volume IRQ handler. See **`UltraVolumeHandler()`**. This can be used to create your own multi-point volume envelopes.

Bi-directional looping can be used to create a vibrato effect.

Appendix C - Voice Control Bits

Here are the voice control bit definitions:

Bit 7 (0x80) **Reserved**

Bit 6 (0x40) **Reserved**

Bit 5 (0x20) **1 = Wavetable IRQ Enable**

Bit 4 (0x10) **1 = Bidirectional Enable**

Bit 3 (0x08) **1 = Loop Enable**

Bit 2 (0x04) **0/1 = 8-bit/16-bit data**

Bit 1 (0x02) **Reserved**

Bit 0 (0x01) **Reserved**

The UltraSound is capable of playing back 8 or 16 bit data. (It can only record 8 bit). Stereo is handled by using 2 voices. It can loop on the data in either a unidirectional or bidirectional mode. If you have asked that this voice generate a wavetable IRQ when it hits the end of the data (or loop point, if looping is specified), be sure you have specified a wavetable IRQ hander. (**`UltraWaveHander()`**). These mode bits would be passed to **`UltraStartVoice()`**, **`UltraSetLoopMode()`**, and **`UltraNoteOn()`**.

Appendix D - DMA Control Bits

Here are the dma to/from DRAM control bit definitions:

Bit 7 (0x80) **1 = Convert to 2's-complement**

Bit 6 (0x40) **0/1 = 8-bit/16-bit data**

Bit 5 (0x20) **Reserved**

Bit 4 (0x10) **Reserved**

Bit 3 (0x08) **Reserved**

Bit 2 (0x04) **Reserved**

Bit 1 (0x02) **0/1 = Read/Write**

Bit 0 (0x01) **Reserved**

Note: The UltraSound DRAM location MUST be on a 32-byte boundary. **`UltraMemAlloc()`** enforces this stipulation.

This is the definition of the bits to be passed to **`UltraUpload()`** and **`UltraDownload()`**. To DMA the data out of DRAM, use **`UltraUpload()`** and to send data to the DRAM, use **`UltraDownload()`**. Be sure to specify the sample size and whether or not the data is in two compliment form. The UltraSound can only play data back that is in two's compliment form. If your sample is in one's compliment form, turn on bit 7. The data will be translated to two's compliment as it is being DMA'ed.

Note: If you are poking data into DRAM, you MUST put the data in twos compliment yourself. This is accomplished by exclusive OR-ing the high bit with a 1.

Appendix E - Recording Control Bits

Here are the recording control bit definitions:

Bit 7 (0x80) **Reserved**

Bit 6 (0x40) **Reserved**

Bit 5 (0x20) **Reserved**

Bit 4 (0x10) **Reserved**

Bit 3 (0x08) **Reserved**

Bit 2 (0x04) **Reserved**

Bit 1 (0x02) **0/1 = Mono/Stereo**

Bit 0 (0x01) **Reserved**

Appendix F - Patch Header

```
/*****************************************************************
* NAME: PATCH.H                                                  *
* COPYRIGHT:                                                      *
* "Copyright (c) 1991,1992, by FORTE                             *
*                                                                *
* "This software is furnished under a license and may be used,   *
* copied, or disclosed only in accordance with the terms of such *
* license and with the inclusion of the above copyright notice.  *
* This software or any other copies thereof may not be provided or *
* otherwise made available to any other person.  No title to and  *
* ownership of the software is hereby transferred."              *
*****************************************************************
* CREATION DATE: 07/01/92                                        *
* VERSION   DATE   NAME DESCRIPTION                              *
*   1.0   07/01/92 Original                                      *
*****************************************************************/

#define ENVELOPES 6

/* This is the definition for what FORTE's patch format is. */
/* All .PAT files will have this format.  */

#define HEADER_SIZE                 12
#define ID_SIZE                     10
#define DESC_SIZE                   60
#define RESERVED_SIZE               40
#define PATCH_HEADER_RESERVED_SIZE  36
#define LAYER_RESERVED_SIZE         40
#define PATCH_DATA_RESERVED_SIZE    36
#define GF1_HEADER_TEXT             "GF1PATCH110"

typedef struct
{
    char header[ HEADER_SIZE ];
    char gravis_id[ ID_SIZE ];       /* Id = "ID#000002" */
    char description[ DESC_SIZE ];
    unsigned char instruments;
    char voices;
    char channels;
    unsigned int wave_forms;
    unsigned int master_volume;
    unsigned long data_size;
    char reserved[ PATCH_HEADER_RESERVED_SIZE ];
} PATCHHEADER;

typedef struct
{
    unsigned int instrument;
    char instrument_name[ 16 ];
    long instrument_size;
    char layers;
```

```
    char reserved[ RESERVED_SIZE ];
} INSTRUMENTDATA;

typedef struct
{
    char layer_duplicate;
    char layer;
    long layer_size;
    char samples;
    char reserved[ LAYER_RESERVED_SIZE ];
} LAYERDATA;

typedef struct
{
    char wave_name[7];

    unsigned char fractions;
    long wave_size;
    long start_loop;
    long end_loop;

    unsigned int sample_rate;
    long low_frequency;
    long high_frequency;
    long root_frequency;
    int tune;

    unsigned char balance;

    unsigned char envelope_rate[ ENVELOPES ];
    unsigned char envelope_offset[ ENVELOPES ];

    unsigned char tremolo_sweep;
    unsigned char tremolo_rate;
    unsigned char tremolo_depth;

    unsigned char vibrato_sweep;
    unsigned char vibrato_rate;
    unsigned char vibrato_depth;

    /* bit 0 = 8 or 16 bit wave data.  */
    /* bit 1 = Signed - Unsigned data.  */
    /* bit 2 = looping enabled-1.  */
    /* bit 3 = Set is bidirectional looping.  */
    /* bit 4 = Set is looping backward.  */
    /* bit 5 = Turn sustaining on.  (Env.  pts.  3)*/
    /* bit 6 = Enable envelopes - 1 */
    char modes;

    int scale_frequency;
    unsigned int scale_factor;
    /* from 0 to 2048 or 0 to 2 */
    char reserved[ PATCH_DATA_RESERVED_SIZE ];
} PATCHDATA;
```

Appendix G - 3D File Header

```
/*****************************************************************
* NAME: HDR3D.H                                                  *
* COPYRIGHT:                                                     *
* "Copyright (c) 1992, by FORTE                                  *
*                                                                *
* "This software is furnished under a license and may be used,   *
* copied, or disclosed only in accordance with the terms of such *
* license and with the inclusion of the above copyright notice.  *
* This software or any other copies thereof may not be provided or *
* otherwise made available to any other person.  No title to and  *
* ownership of the software is hereby transferred."              *
*****************************************************************
* CREATION DATE: 02/01/93                                        *
* VERSION   DATE   NAME DESCRIPTION                              *
*   1.0   02/01/93 Original                                      *
*****************************************************************/
/* Bit definitions for tracks that are in a 3D file  */
#define FRONT_TRACK  0x01
#define RIGHT_TRACK  0x02
#define REAR_TRACK   0x04
#define LEFT_TRACK   0x08
#define ABOVE_TRACK  0x10
#define BELOW_TRACK  0x20
#define FBLOCK_3D    0x01 /* File data is blocked, not interleaved */
#define F16BIT_3D    0x02 /* 16 bit data */
#define FTWOS_CMP_3D 0x04 /* sound is in twos complement form */
#define FLOOPED_3D   0x08 /* sound is looped , not one shot*/
#define FBI_LOOP_3D  0x10 /* sound is bi-directional */
#define F3D_TYPE     0x60 /* Type of 3D sound */

/* types (F3D_TYPE) of 3D sound (only binaural currently supported) */
#define F_BINAURAL   0x20 /* Binaural representation */
#define F_SURROUND   0x40 /* Surround sound (???) */
#define F_QSOUND     0x60 /* Q sound (???) */

/* this structure is exactly 256 bytes long ...  */
typedef struct
{
    char id[10];                /* 3D FILE tag */
    int major;                  /* major version # */
    int minor;                  /* minor version # */
    char description[80];
    unsigned int type;          /* See above ....  */
    int tracks;                 /* tracks included in this file  */
    int reserve1[24];           /* for expansion */
    int maxvol;                 /* volume ranges from 0 to maxvol */
    int reserve2[10];           /* for expansion */
    unsigned long blocksize;   /* # of bytes in block data */
    unsigned long loop_offset; /* byte offset to where loop begins  */
    unsigned long reserve3[9]; /* for expansion */
```

```
    unsigned long frequency;     /* initial playback frequency */
    unsigned long reserve4[10]; * for expansion */
} FILEHDR_3D;
```