UltraSound Lowlevel ToolKit

Revision 2.22

21 December 94

Advanced Gravis
101-3750 North Fraser Way
Burnaby, British Columbia V5J 5E9
FAX (604) 451-9358

Part No. 901-0031-02
Printed In Canada

Table of Contents

µ1. Chapter 1 - General Information 1
1.1. Introduction 1
1.2. Features of the UltraSound 1
1.3. Benefits of supporting the UltraSound 2
1.4. The GF1 - 32 Voice Sound Synthesizer 2
1.5. MIDI Interface 4
1.6. Joystick Interface 5
2. Chapter 2 - Hardware Information 6
2.1. I/O Port Map 6
2.2. MIDI Control Port - 3X0 7
2.3. MIDI Status Port - 3X0 7
2.4. MIDI Data Port - 3X1 7
2.5. Page Register - 3X2 8
2.6. Select Register - 3X3 8
2.6.1. Global Registers 8
2.6.1.1. DRAM DMA Control Register - (41) 9
2.6.1.2. DMA Start Address - (42) 9
2.6.1.3. DRAM I/O Address (43,44) 9
2.6.1.4. Timer Control - (45) 10
2.6.1.5. Timer 1 and Timer 2 Count - (46,47) 10
2.6.1.6. Sampling Frequency - (48) 10
2.6.1.7. Sampling Control Register - (49) 10
2.6.1.8. Joystick Trim DAC - (4B) 10
2.6.1.9. Reset Register - (4C) 11
2.6.2. Voice-specific Registers 11
2.6.2.1. Voice Control Register - (0,80) 12
2.6.2.2. Frequency Control Register - (1,81) 12
2.6.2.3. Starting location HIGH - (2,82) 12
2.6.2.4. Starting location LOW - (3,83) 13
2.6.2.5. End Address HIGH - (4,84) 13
2.6.2.6. End Address LOW - (5,85) 13
2.6.2.7. Volume Ramp Rate - (6,86) 13
2.6.2.8. Volume Ramp Start - (7,87) 13
2.6.2.9. Volume Ramp End - (8,88) 13
2.6.2.10. Current Volume - (9,89) 14
2.6.2.11. Current Location HIGH - (A,8A) 14
2.6.2.12. Current Location LOW - (B,8B) 14
2.6.2.13. Pan Position - (C,8C) 14
2.6.2.14. Volume Ramp Control Register - (D,8D) 15
2.6.2.15. Active Voices - (E,8E) 15
2.6.2.16. IRQ Source Register - (F,8F) 16
2.7. Global Data Low - 3X4 16
2.8. Global Data High - 3X5 16
2.9. IRQ Status - 2X6 17
2.10. Timer Control Register - 2X8 17
2.11. Timer Data Register - 2X9 17

2.12. DRAM I/O - 3X7 17
2.13. Mix Control Register - 2X0 18
2.14. IRQ Control Register - 2XB 18
2.15. DMA Control Register - 2XB 19
2.16. Register Control - 2XF 20
2.17. Mixer Control 7X6 20
2.18. Mixer Data - 3X6 20
2.19. Revision Level - 7X6 20
2.20. Codec Registers 20
2.21. UltraMax Control Register - 3X6 21
2.22. Volume ramping description 21
3. Chapter 3 - Programming the UltraSound 23
3.1. Introduction 23
3.2. Sound 23
3.3. The Basics of the UltraSound 24
3.4. Using GUS Memory 24
3.5. What are Samples? 25
3.6. Using Voices 25
3.7. Volumes 26
3.8. Using Looping 27
3.9. Clicks and click removal 27
3.10. Interrupt Handling Functions 28
3.11. Rollover feature 29
3.12. Stereo playback 30
3.13. C-specific information 32
3.14. PASCAL-specific information 33
3.14.1. Available constants and variables 34
3.14.2. Examples 35
3.14.3. Management of GUS RAM. 36
3.15. Technical Support 37
4. Chapter 4 - Reference Guide 38
4.1. UltraCalcRate 38
4.2. UltraClose 38
4.3. UltraDownload 39
4.4. UltraDRAMDMABusy 39
4.5. UltraGoRecord 40
4.6. UltraGoVoice 40
4.7. UltraDisableLineIn 40
4.8. UltraDisableMicIn 41
4.9. UltraDisableOutput 41
4.10. UltraEnableLineIn 41
4.11. UltraEnableMicIn 42
4.12. UltraEnableOutput 42
4.13. UltraGetLineIn 42
4.14. UltraGetOutput 43
4.15. UltraGetMicIn 43
4.16. UltraDRAMTcHandler 44
4.17. UltraMIDIXmitHandler 45
4.18. UltraMIDIRecvHandler 45
4.19. UltraTimer1Handler 46
4.20. UltraTimer2Handler 46
4.21. UltraWaveHandler 47
4.22. UltraVolumeHandler 47
4.23. UltraRecordHandler 48
4.24. UltraAuxHandler 48
4.25. UltraMaxAlloc 48

4.26. UltraMaxAvail 49
4.27. UltraMemAvail 49
4.28. UltraMemAlloc 50
4.29. UltraMemFree 50
4.30. UltraMemInit 51
4.31. UltraMidiDisableRecv 51
4.32. UltraDisableMIDIXmit 51
4.33. UltraMIDIEnableRecv 52
4.34. UltraEnableMIDIXmit 52
4.35. UltraMIDIRecv 52
4.36. UltraMIDIReset 53
4.37. UltraMIDIStatus 53
4.38. UltraMIDIXmit 53
4.39. UltraOpen 54
4.40. UltraPeekData 54
4.41. UltraPing 55
4.42. UltraPokeData 55
4.43. UltraPrimeRecord 56
4.44. UltraPrimeVoice 56
4.45. UltraProbe 57
4.46. UltraRampVolume 57
4.47. UltraReadRecordPosition 58
4.48. UltraReadVoice 58
4.49. UltraReadVolume 58
4.50. UltraRecordData 59
4.51. UltraRecordDMABusy 59
4.52. UltraReset 60
4.53. UltraSetBalance 60
4.54. UltraSetFrequency 60
4.55. UltraSetLoopMode 61
4.56. UltraSetRecordFrequency 61
4.57. UltraSetVoice 61
4.58. UltraSetVoiceEnd 62
4.59. UltraSetVolume 62
4.60. UltraSizeDRAM 62
4.61. UltraStartTimer 63
4.62. UltraStartVoice 63
4.63. UltraStopTimer 64
4.64. UltraStopVoice 64
4.65. UltraStopVolume 64
4.66. UltraTimerStopped 65
4.67. UltraTrimJoystick 65
4.68. UltraUpload 66
4.69. UltraVectorVolume 66
4.70. UltraVersion 67
4.71. UltraVersionStr 67
4.72. UltraVoiceStopped 67
4.73. UltraVolumeStopped 68
4.74. UltraWaitDRAMDMA 68
4.75. UltraWaitRecordDMA 68
4.76. UltraAllocVoice 69
4.77. UltraClearVoices 69
4.78. UltraFreeVoice 69
4.79. UltraVoiceOff 70
4.80. UltraVoiceOn 70
4.81. UltraSetLinearVolume 71

4.82. UltraReadLinearVolume 71
4.83. UltraRampLinearVolume 71
4.84. UltraVectorLinearVolume 72
5. Chapter 5 - Mixer 73
5.1. Introduction 73
5.2. Block Diagram 73
5.3. Notes 74
5.4. Mixer Functions 74
5.5. UltraMixProbe 74
5.6. UltraMixAttn 75
5.7. UltraMixXparent 75
6. Chapter 6 - 16 Bit Codec Functions 76
6.1. Introduction 76
6.2. Features: 76
6.3. Block Diagram 77
6.4. Daughter card Specific Info 77
6.5. UltraMax Codec Specific Info 78
6.6. UltraMax DMA Channel block diagram 78
6.7. Codec Functions 78
6.7.1. ULTRA16PROBE 79
6.7.2. ULTRA16OPEN 80
6.7.3. ULTRA16CLOSE 80
6.7.4. ULTRA16DISABLEIRQS 80
6.7.5. ULTRA16ENABLEIRQS 80
6.7.6. IRQ CALLBACK HANDLERS 81
6.7.7. ULTRA16REVISION 81
6.7.8. ULTRA16VERSION 81
6.7.9. ULTRA16XPARENT 81
6.7.10. CD INPUT LEVELS 82
6.7.11. GF1 (SYNTH) INPUT LEVELS 82
6.7.12. LINE INPUT LEVELS 82
6.7.13. MIC INPUT LEVEL 83
6.7.14. INPUT MUTES 83
6.7.15. MIC GAIN 83
6.7.16. RECORDING SOURCE 84
6.7.17. RECORDING GAIN 84
6.7.18. OUTPUT LEVELS 84
6.7.19. OUTPUT MUTES 84
6.7.20. ULTRA16SETFREQ 85
6.7.21. ULTRA16GOPLAY 85
6.7.22. ULTRA16PLAYDATA 86
6.7.23. ULTRA16PLAYFORMAT 86
6.7.24. ULTRA16PROGPLAYCNT 86
6.7.25. ULTRA16READPLAYPOSITION 87
6.7.26. ULTRA16STARTPLAYDMA 87
6.7.27. ULTRA16PLAYDMABUSY 87
6.7.28. ULTRA16WAITPLAYDMA 87
6.7.29. ULTRA16STOPPLAYDMA 88
6.7.30. START_PLAY 88
6.7.31. ULTRA16READRECORDPOSITION 88
6.7.32. ULTRA16PROGRECCNT 88
6.7.33. ULTRA16RECFORMAT 89
6.7.34. ULTRA16GORECORD 89
6.7.35. ULTRA16RECORDDATA 89
6.7.36. ULTRA16STARTRECORDDMA 90
6.7.37. ULTRA16RECORDDMABUSY 90

6.7.38. ULTRA16WAITRECORDDMA 90
6.7.39. ULTRA16STOPRECORDDMA 90
6.7.40. START_RECORDING 91
6.7.41. ULTRA16STARTTIMER 91
6.7.42. ULTRA16STOPTIMER 91
6.7.43. ULTRA16SETTIMER 91
7. Chapter 7 - Utility Functions 92
7.1. Introduction 92
7.2. UltraGetCfg 92
7.3. GetUltra16Cfg 92
7.4. MallocAlignedBuff 93
8. Appendix A - Error Codes 94
9. Appendix B - Volume Control 95
10. Appendix C - Voice Control 96
11. Appendix D - DMA Control 97
12. Appendix E - Recording Control 98
13. Appendix F - Patch Files 99
13.1. File Header 104
13.2. Instrument Header 105
13.3. Layer Header 105
13.4. Wave Header 105
14. Appendix G - Modulation Tables 108
14.1. RATE table for TREMOLO and VIBRATO 108
14.2. DEPTH table for TREMOLO 110
14.3. DEPTH table for VIBRATO 112

1. Chapter 1 - General Information

1.1. Introduction

This is version 2.22 of the Software Development Kit (SDK) for the Advanced Gravis UltraSound card, and is the combined work of Advanced Gravis, FORTE Technologies, and Ingenuity Software. This kit is intended for use by programmers of IBM-compatible PCs. After becoming familiar with this material, you should be able to use C or Pascal to program the UltraSound under MS-DOS.

This is the first version which includes a full Borland Pascal (version 7) and Turbo-Pascal (versions 6 and 7) interface. This was written by Kurt Kennett of Ingenuity Software for Advanced Gravis, and is the direct translation of the routines in the C interface. For specific information about the Pascal interface, see section 3.14.

For specific information about the revised and updated C interface, including compilers and memory models supported, see section 3.13.

Some of the discussions of the hardware interface to the card (Chapter 2) are quite technical, and go beyond the level of technical sophistication needed to write software. Therefore, a separate section (Chapter 3) has been included which is a general overview of how to write software for the card. What follows in this chapter is a general description of the UltraSound card and its features, the MIDI functionality, the Joystick interface, and the GF1 32-voice Sound Synthesizer. Throughout this documentation, the words 'lowlevel tool kit', 'lowlevel routines' and 'lowlevel code' are used. This refers to any and all of the routines described and included in the C and/or Pascal interfaces.

1.2. Features of the UltraSound

- 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 wavetable synthesis: all voices mixed on board.
- Simultaneous playback and recording is possible.
- Each voice has its own volume settings, volume enveloping, playback rate and balance.
- Both line level and amplified outputs.
- 6850 compatible MIDI UART on-board.
- Gravis Eliminator joystick interface with jumper enable/disable.
- 256Kb DRAM on board for waveforms. Expandable to 1Mb.
- Stereo microphone input with automatic level control.
- Line level input.
- CD ROM Drive audio input.
- Stereo mini-jacks for line & amplified outputs and Mic and Line inputs.

1.3. Benefits of supporting the UltraSound

The UltraSound has 32 separate voices, each of which can be used to play back any 8- or 16-bit digital data that is loaded into its DRAM. This data can consist of individual sound effects, a whole sound track, or even single musical notes of a digitized instrument. The UltraSound is completely controlled by the PC, thus giving maximum flexibility to applications which use it. The main chip which runs the card (the GF1) is directly addressable from the PC's CPU via basic I/O instructions.

Here are some of the features that the UltraSound offers:

1) WaveTable synthesis has a much higher sound quality that FM-based cards can ever achieve. This method of audio synthesis is the same method used in expensive professional keyboards.
2) 32 digital voices can be independently controlled. This allows you to have up to 32 sound effects or instruments going at once.
3) The card is RAM-based. Since the instrument patches are kept on disk and only loaded when they are needed, they can be changed or improved at will.
4) All mixing is done on the card. There is no burden on the CPU to playback multiple channels.
5) Multiple cards can be installed in the same PC. With some relatively minor adjustments to the SDK software, multiple cards can be controlled in the same PC.
6) Hardware-assisted voice enveloping. A multi-point attack-decay-sustain-release envelope can be implemented that uses virtually no CPU overhead. Tremolo can also be done in hardware.
7) Sound effects can be pre-loaded into DRAM, using little or no PC memory to keep track of them. This leaves more space for your application and it's data. It also allows you to start or stop any sound whenever you wish.

1.4. The GF1 - 32 Voice Sound Synthesizer

The UltraSound uses Wave-Table Synthesis to produce sound output. This means that either sampled data from actual instruments or other synthesized digital audio is stored in DRAM on the UltraSound Board. The GF1 chip which controls the board is set up to play back relatively short digital audio samples and produce continuous sound closely reproducing the original instrument. The 32 voices are independently controlled and can be producing different sounds concurrently. Output from all voices is mixed into the left or right channels. 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.

Multi-track 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.

The GF1 is basically a pipeline processor. It constantly loops from voice #0 to the end of the active voices (how to define the number of active voices is shown later). The GF1 takes 1.6 microseconds to service a 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	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, the voice servicing rate 'X' can be calculated from:

$$1{,}000{,}000 / (X * 14) = 44100 \text{ Hz}$$
$$X = 1.619695497 \text{ microseconds}$$

Once X is known, the frequency 'divisor' is calculated by:

$$\text{divisor} = 1{,}000{,}000 / (1.619695497 * \# \text{ of active voices})$$

The lowlevel code pre-calculates this table so that floating point arithmetic doesn't need to be done. A frequency 'counter' is used to calculate how often a voice is updated by the GF1. To calculate an FC (frequency counter) for any given frequency with a particular # of active voices, run it through this formula:

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

PASCAL:

```
fc := Word(((Speed_Khz SHL 9)+(Divisor SHR 1)) DIV Divisor);
fc := fc SHL 1;
```

The left shift is needed since the FC is in bits 15-1. For more information about the frequency counter, see chapter 2, the hardware section.

This value 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 voice's 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 through 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 fourth time through it uses 100% of location 1.

The interpolation is done to a resolution of 16 bits, even for 8 bit playback. This 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.

1.5. MIDI Interface

The MIDI 101 interface consists of standard UART functionality - Motorola MC68C50. An interrupt to the PC can be generated for each byte of MIDI data received or transmitted. This hardware is independent of any of the other hardware. The main MIDI circuitry is included in the GF1 processor, but external to the chip 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 loop back transmit data to the receive data under software control. The serial interface has 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 specifications for the interface are:

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 joystick. An external cable assembly containing the optical isolator and driver is required to use the MIDI interface.

1.6. Joystick Interface

The joystick interface is an Eliminator Joystick interface designed by Advanced Gravis. It basically consists of a single 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 microsecond.

UltraSound boards with a revision version (printed on the card) of 3.3 and lower have a jumper which is used to enable or disable the joystick. Boards with a revision version of 3.4 and above have a software enable/disable for the joystick instead of a jumper. There is a joystick SDK available separately from Gravis.

2. Chapter 2 - Hardware Information

2.1. I/O Port Map

The following describes the 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, INT,DMA	R,W	ADDRESS HEX
UltraSound Base Port:	---	--	2X0
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 (2X0- bit 6 = 1)	I/O	W	2XB
DMA control register (2X0- bit 6 = 0)	I/O	W	2XB
Register Controls	I/O	R/W	2XF (Rev 3.4+)
Board Version	I/O	R	7X6 (Rev 3.7+)
Mixer Control			
Control Port	I/O	W	7X6
Data Port	I/O	W	3X6
CODEC			
Daughter Card	I/O	R/W	530-533
or	I/O	R/W	604-607
or	I/O	R/W	E80-E83
or	I/O	R/W	F40-F43
UltraMax	I/O	R/W	3XC-3XF
Codec Addr Select	I/O	R/W	Base+0
Codec Data	I/O	R/W	Base+1

Codec Status	I/O	R/W	Base+2
Codec PIO	I/O	R/W	Base+3

UltraMax Control Port (MAX ONLY)	I/O	W	3X6

At power-up the board has operational Joystick and MIDI interfaces. This allows their direct use with existing software. The GF1 ASIC powers up with all voices disabled, not requiring a software initialization. This helps eliminate noise at powerup and allows 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 address 3X0 hex and is write only.

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- 1 \ Master reset (when set)
  |   |   |   |   |   |   +-------- 1 /
  |   |   |   |   |   +------------ Reserved
  |   |   |   |   +---------------- Reserved
  |   |   |   +-------------------- Reserved
  |   |   +------------------------ 1 \ xmit IRQ enabled
  |   +---------------------------- 0 /
  +-------------------------------- 1 = Receive IRQ enabled
```

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 address 3X0 hex and is read-only.

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- Receive reg. full
  |   |   |   |   |   |   +-------- Transmit reg. empty
  |   |   |   |   |   +------------ Reserved
  |   |   |   |   +---------------- Reserved
  |   |   |   +-------------------- Framing Error
  |   |   +------------------------ Overrun error
  |   +---------------------------- Reserved
  +-------------------------------- Interrupt pending
```

The MIDI control interface behaves identically to a 6850 UART.

2.4. MIDI Data Port - 3X1

The transmit and receive 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. They 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)

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |  |  |  |  |  |  |  |
  |  |  |  |  |  |  |  +---- Enable DMA (1=go)
  |  |  |  |  |  |  +-------- DMA direction (1=read)
  |  |  |  |  |  +------------ DMA channel width
  |  |  |  |  +---------------- \
  |  |  |  +-------------------- / DMA Rate divider
  |  |  +------------------------ DMA IRQ Enable
  |  +---------------------------- (R) DMA IRQ Pending
  |                               (W) DATA SIZE (0=8bit,1=16bit)
  +-------------------------------- Invert MSB (write only)
```

Bit 0 - 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.
Bit 1 - DMA transfer direction. Read is taking data OUT of the UltraSound, Write sends data to it.
Bit 2 - 0 = if DMA channel is an 8 bit channel (0-3).
= If it is a 16 bit channel (4-7)
Note: This is INDEPENDENT of the data size.
Bit 3,4 - DMA Rate divisor. The Maximum rate is approx. 650 khz.
0,0 = divide by 1
0,1 = divide by 2
1,0 = divide by 3
1,1 = divide by 4
Bit 5 - DMA terminal count interrupt enable
Bit 6 - Read - DMA terminal count IRQ pending
Write - Data size 0 = 8 Bit data
1 = 16 bit data
Note: Data size is independent of channel size
Bit 7 - 1 = Invert High bit to flip data to twos complement form.
Note: This flips bit 7 for 8 bit data and bit 15 for
bit data.

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 or 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 begin on a 32 byte boundary. An additional address translation is necessary if a 16 bit DMA channel is used. For simple example code on how to do this translation, see the C function convert_to_16().

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)

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- Reserved (Set to 0)
  |   |   |   |   |   |   +-------- Reserved (Set to 0)
  |   |   |   |   |   +------------ Enable Timer 1 IRQ
  |   |   |   |   +---------------- Enable Timer 2 IRQ
  |   |   |   +-------------------- Reserved (Set to 0)
  |   |   +------------------------ Reserved (Set to 0)
  |   +---------------------------- Reserved (Set to 0)
  +-------------------------------- Reserved (Set to 0)
```

2.6.1.5. Timer 1 and Timer 2 Count - (46,47)

These counts are loaded by the application and then they will count up to $FF and generate and IRQ. Timer 1 has a granularity of 80 microseconds (0.00008 sec) and Timer 2 has a granularity of 320 microseconds (0.00032 sec).

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)

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- Start sampling
  |   |   |   |   |   |   +-------- Mode (0=mono, 1=stereo)
  |   |   |   |   |   +------------ DMA width (0=8bit,1=16bit)
  |   |   |   |   +---------------- Reserved (Set to 0)
  |   |   |   +-------------------- Reserved (Set to 0)
  |   |   +------------------------ DMA IRQ enable
  |   +---------------------------- (Read) DMA IRQ pending
  +-------------------------------- Invert MSB
```

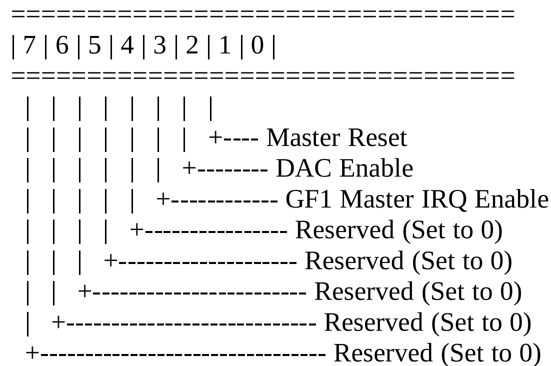
Bit 0 - If PC DMA controller is programmed, it will begin sampling as soon as this is enabled.
Bit 1 - 0 = mono
= stereo
In stereo mode, the order of the data bytes is left is first, and right is second. If a 16 bit data channel is used, the left is in the lower byte.

Bit 2 - DMA Channel width (0 = 8 bit, 1 = 16 bit)
Bit 5 - Enable DMA terminal count IRQ
Bit 6 - DMA terminal count IRQ pending
Bit 7 - Flip bit 7 to get non-twos compliment data

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 0 - GF1 Master Reset. 0 = reset, 1 = run. As long as this is a 0, it will be held in a reset state.
Bit 1 - Enable DAC output. DAC's will not run unless this bit is set.
Bit 2 - Master IRQ enable. This bit MUST be set to get ANY of the GF1-generated IRQs (wavetable, volume, etc).

This register will normally contain the value $07 while your application is running.

2.6.2. Voice-specific Registers

The following 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.

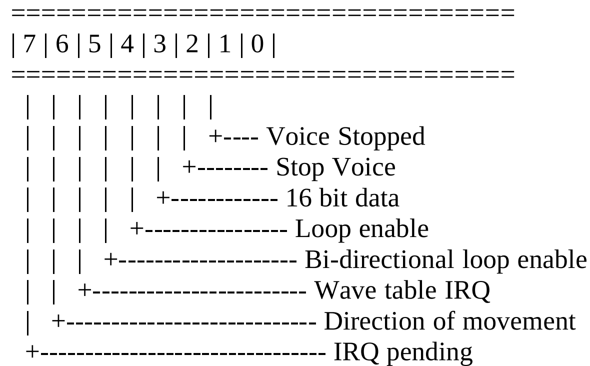
Write	Read	Width	Description
0	80	8	Voice Control
1	81	16	Frequency Control
2	82	16	Starting Address (HIGH)
3	83	16	Starting Address (LOW)
4	84	16	End Address (HIGH)
5	85	16	End Address (LOW)
6	86	8	Volume Ramp Rate
7	87	8	Volume Ramp Start
8	88	8	Volume Ramp End
9	89	16	Current Volume
A	8A	16	Current 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)
-	8F	8	IRQ Status (Voice independent)

There are several 'self-modifying' bits defined for these registers. This means that the GF1 may change them at anytime on its own. Due to the fact that the software must accommodate this phenomena, it is 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 it has 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 code, this is done with a function called GF1_Delay. The self-modifying bits are designated with an (*) after the particular bit definition.

Changing the start and end points of a voice while its playing can have some strange side effects. For example, if you change end position to a lower location than it is currently playing, you will get an IRQ (if they are enabled). Also, since the high and low bytes are set individually and asynchronously to when the GF1 is working on a voice, it is possible to get an unexpected IRQ if the current position and the new end position cross.

2.6.2.1. Voice Control Register - (0,80)



* Bit 0- 1 = Voice is stopped. This gets set by hitting the end address (not looping) or by setting bit 1 in this reg.

Bit 1- 1 = Stop Voice. Manually force voice to stop.

Bit 2- 1 = 16 bit wave data, 0 = 8 bit data

Bit 3- 1 = Loop to begin address when it hits the end address.

Bit 4- 1 = Bi-directional looping enabled

Bit 5- 1 = Enable wavetable IRQ. Generate an IRQ when the voice hits the end address. Will generate IRQ even if looping is enabled.

* Bit 6- 1 = Decreasing addresses, 0 = increasing addresses. It is self-modifying because it might shift directions when it hits one of the loop boundaries and looping is enabled.

* Bit 7- 1 = WaveTable IRQ pending. If IRQ's are enabled and looping is NOT enabled, an IRQ will be constantly generated until the voice is stopped. This means that you may get more than 1 IRQ if it isn't handled properly.

2.6.2.2. Frequency Control Register - (1,81)

Bits 15-10 - Integer Portion
Bits 9-1 - Fractional Portion
Bit 0 - Not used.

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 (register 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 (Addr 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. (Addr 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. (Addr 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. (Addr 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.

Please see section 2.22 for more information on hardware volume ramping.

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 the special section on volume ramping for a more complete explanation of how this register works.

Please see section 2.22 for more information on hardware volume ramping.

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 the special section on volume ramping 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.

Please see section 2.22 for more information on hardware volume ramping.

2.6.2.10. Current Volume - (9,89)

* Bits 15-12 Exponent
* Bits 11-4 Mantissa
Bits 3-0 Reserved (Set to 0)

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).

Please see section 2.22 for more information on hardware volume ramping.

2.6.2.11. Current Location HIGH - (A,8A)

Bits 15-13 Reserved (Set to 0)
Bits 12-0 High 13 bits of address (address lines 19-7)

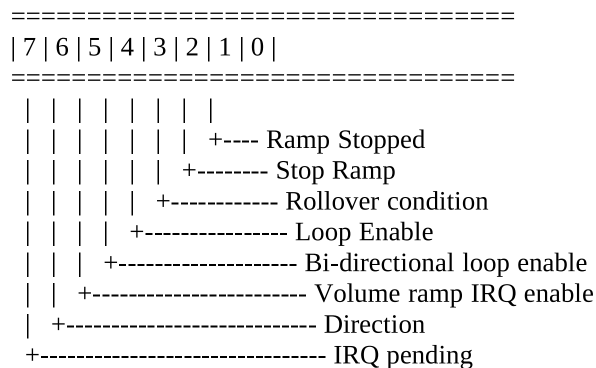
2.6.2.12. Current Location LOW - (B,8B)

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 Reserved (Set to 0)
Bits 3-0 Pan position. (0=full left, 15=full right)

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



* Bit 0- Show the ramp has stopped
Bit 1- Manually stop the ramp.
Bit 2- Roll over condition. 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 through 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 3- Enable looping. Loop from end to start (or start to end).
Bit 4- Enable bi-directional looping. When it hits end (or start) it will change directions and proceed toward the other limit.
Bit 5- Enable getting an IRQ when ramp hits end.
* Bit 6- Ramp direction. 0=increasing, 1=decreasing.
* Bit 7- (READ) Volume ramp IRQ pending.

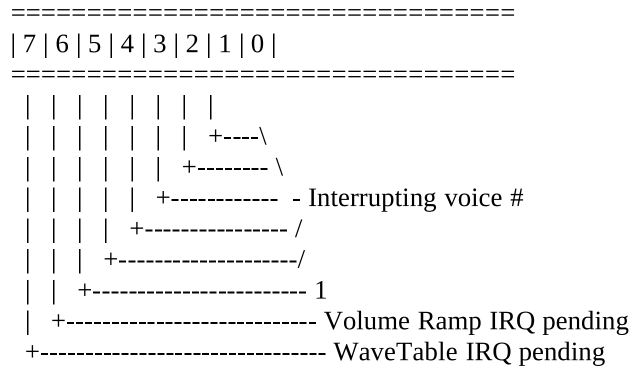
Please see section 2.22 for more information on hardware volume ramping.

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 0-4- Voice # (0-31) of interrupting voice
Bit 5- ALWAYS a 1
Bit 6- 0 = Volume Ramp IRQ occurred
Bit 7- 0 = Wavetable IRQ occurred

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

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

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- MIDI Transmit IRQ
  |   |   |   |   |   |   +-------- MIDI Receive IRQ
  |   |   |   |   |   +------------ Timer 1 IRQ
  |   |   |   |   +---------------- Timer 2 IRQ
  |   |   |   +-------------------- Reserved (Set to 0)
  |   |   +------------------------ WaveTable IRQ (any voice)
  |   +---------------------------- Volume Ramp IRQ (any voice)
  +-------------------------------- DMA TC IRQ (DRAM or Sample)
```

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.

2.11. Timer Data Register - 2X9

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- Timer 1 Start
  |   |   |   |   |   |   +-------- Timer 2 Start
  |   |   |   |   |   +------------ Reserved (Set to 0)
  |   |   |   |   +---------------- Reserved (Set to 0)
  |   |   |   +-------------------- Reserved (Set to 0)
  |   |   +------------------------ Mask Timer 2
  |   +---------------------------- Mask Timer 1
  +-------------------------------- Reset Timer IRQ
```

Bit 0 - Start up timer #1
Bit 1 - Start up timer #2
Bit 5 - Mask off timer 2
Bit 6 - Mask off timer 1
Bit 7 - Clear Timer IRQ

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

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- 0=Enable Line IN
  |   |   |   |   |   |   +-------- 0=Enable Line OUT
  |   |   |   |   |   +------------ 1=Enable MIC IN
  |   |   |   |   +---------------- Enable latches
  |   |   |   +-------------------- Combine chan1 IRQ with Chan2
  |   |   +------------------------ Enable MIDI loopback TxD to RxD
  |   +---------------------------- Control Reg Select
  +-------------------------------- Reserved (Set to 0)
```

Bit 0 - Enable UltraSound line Input
Bit 1 - Enable UltraSound Line output
Bit 2 - Enable Stereo Mic Input
Bit 3 - 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 4 - Combine Channel 1 (GF1) IRQ's with Channel 2 (MIDI)
Bit 5 - Enable MIDI loopback. Any data sent out Transmit port will be looped back into the input port.
Bit 6 - 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 accidentally corrupt the latches.

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=RESERVED, DO NOT USE
    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 Reserved (Set to 0)

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 #.

2.15. DMA Control Register - 2XB

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 - Reserved (Set to 0).

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 #.

C programmers can refer to the UltraSetInterface routine 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 an IRQ on the PC. Normally, this is not a problem, but it is something to be aware of.

2.16. Register Control - 2XF

This register is only valid for UltraSound boards that are at or above revision 3.4. It is not valid for boards which have a prior revision number.

On 3.4 and above boards, there is a bank of 6 registers that exist at location 2XB. Register 2XF is used to select which one is being talked to.

Register #	Use
0	Same as pre-3.4 boards
1-4	Unused - Reserved
5	Write a 0 to clear IRQs on power-up
6	'Jumper register'

Jumper register definition:

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- Reserved (Set to 0)
  |   |   |   |   |   |   +-------- 1=Enable MIDI port addr decode
  |   |   |   |   |   +------------ 1=Enable joystick port decode
  |   |   |   |   +---------------- Reserved (Set to 0)
  |   |   |   +-------------------- Reserved (Set to 0)
  |   |   +------------------------ Reserved (Set to 0)
  |   +---------------------------- Reserved (Set to 0)
  +-------------------------------- Reserved (Set to 0)
```

2.17. Mixer Control 7X6

See Chapter 5 for a complete description of the mixer control port.

2.18. Mixer Data - 3X6

See Chapter 5 for a complete description of the mixer data port.

2.19. Revision Level - 7X6

This register is only valid for UltraSound boards that are 3.7 or greater. Any reads from this port on boards prior to 3.7 will return 0xff. This is a way for software to detect whether or not the ICS-2101 mixer is present. Here is a table of what is currently defined for each revision.

Revision ID	Board Revision
0xff	Pre 3.7 boards. ICS mixer NOT present
5	Rev 3.7 with ICS Mixer. Some R/L: flip problems.

6-9	Revision 3.7 and above. ICS Mixer present
0x0A-NN	UltraMax. CS4231 present, no ICS mixer

2.20. Codec Registers

See Chapter 6 for a description of the register map of the CS4231 codec and how it fits in with both the UltraMax and the 16-bit daughter card.

2.21. UltraMax Control Register - 3X6

This is a write only register that is only valid for the UltraMax. Note that it is on the same port location as the data port for the ICS-2101 mixer. This implies that your software MUST know what device is there before it starts writing to them.

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- Codec Addr Decode Bit 4
  |   |   |   |   |   |   +-------- Codec Addr Decode Bit 5
  |   |   |   |   |   +------------ Codec Addr Decode Bit 6
  |   |   |   |   +---------------- Codec Addr Decode Bit 7
  |   |   |   +-------------------- Capture Channel type (0=8 bit, 1=16 bit)
  |   |   +------------------------ Playback Channel type (0=8 bit, 1=16 bit)
  |   +---------------------------- Codec Enable (0=disable, 1=enable)
  +-------------------------------- Reserved (Set to 0)
```

Bits 0-3 - Define the 'X' in 3XC to define the base port location. By convention, the 'X' is usually set to the same as the base port of the UltraSound, but it doesn't need to be. This value can range from 0 - F.

Bit 4 - Defines the type of DMA channel used for recording. Set this bit to 0 for an 8 bit channel and a 1 for a 16 bit channel.

Bit 5 - Defines the type of DMA channel used for playback. Set this bit to 0 for an 8 bit channel and a 1 for a 16 bit channel.

Bits 4 and 5 need to be set up because the CS4231 is only capable of using and 8 bit DMA channel. To allow it to take advantage of the UltraSounds capability of using a 16-bit channel, extra hardware was added to buffer the data and send it out a 16 bit channel. An UltraMax CS4231 shares the DMA channels with base UltraSounds channels. See chapter 6 for more info on the interactions between the two.

Bit 6 is to enable the CS4231. Until this bit is set, the CS4231 is inactive and will not appear in the address space.

2.22. Volume ramping description

The UltraSound has built-in volume ramping to facilitate the implementation of the attack decay sustain release envelopes. This ramping uses the same principle as the voices for updating the volume values. 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 hardware volume registers and bit 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 1Fh (63), and the value for the slowest possible ramp is 1Ch (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 sweep ramping. Since a volume ramp usually goes between points in between the limits, the actual ramp times will be much smaller.

Allowing to let the volume ramps to go to the extremes 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 SDK routines protect against this by limiting how close to the rails you can get.

3. Chapter 3 - Programming the UltraSound

3.1. Introduction

The UltraSound is frequently referred to as 'GUS', for Gravis' UltraSound. The card has a very good reputation among programmers, since it is very easy to program using a high level language, and provides incredible sound. However, there are some things that you need to be aware of to successfully program the board. Users of either language should consult their respective example programs before attempting to use the card. Also, C users should consult section 3.13, while Pascal users should consult section 3.14.

3.2. Sound

A sound is something that you hear. It is produced by a source and received by the ears of the listener. There are several mediums for the storage of sound, including vinyl, magnetic surfaces, and more recently optical disks.

When something makes a sound, it creates waves of different air intensity. These waves are registered as 'sound', and the amplitude and frequency of these waves determine the 'type' of sound.

Amplitude is determined by the highest and lowest points of the wave, and is registered by your ear as the volume of the sound. Therefore, the loudness of a sound is changed by varying the amplitude of the sound's wave. Frequency is determined by the number of pulses (passes of a single wave peak and trough) which pass a certain point in a certain amount of time. The measure of sound frequency is done using the standard measure for frequency - Hertz (Hz). The higher the frequency, the higher the pitch of the sound that is heard.

Normal sound like speech and sound that comes from mixes of different instruments is a very complex pattern of changes in amplitude and frequency. To record the wave's pattern, a computer takes a large number of 'samples' of the wave produced by the sound. Then when it goes through the individual 'samples' at the same rate they were recorded, it can reconstruct the original wave and produce a sound which is close to the original sound. If the sample rate is not high enough, the computer can miss changes in the amplitude and frequency, and the sound reproduced will not sound exactly like the original sound.

The number of samples taken each second is referred to as the 'sampling rate', and the number of samples played back each second is referred to as the 'playback rate'. When a rock band plays a song, the combined sound wave produced by the singers and instruments is produced. To record this wave at CD-quality on a computer medium (like memory or a hard disk), the wave is 'sampled' 44100 times each second. Since the number of samples per second is a frequency, this rate is measured again in Hertz. Therefore, we say that the recording frequency is 44.1 kHz.

If I record a sound at 44.1 kHz for two seconds, I will have 88200 individual 'samples'. The data that comprises these 'samples' is typically referred to as a 'sample' itself.

When a recording made at 44.1 kHz is played back at 44.1 kHz, the original sound is reproduced almost exactly. Note that the recording and playback frequency are different from the frequency of the sound itself.

Most other sound cards do not have any RAM on them, and therefore have to use the PC's RAM to store the sounds they produce. This leaves less room for application programs, and is just another thing that the CPU has to track and process - slowing down the machine.

On the UltraSound, samples are stored on the card in it's DRAM. This leaves more space in PC RAM for programs. Also, the GF1 chip in the UltraSound can play 32 independent sounds at once without using any CPU time - so the machine doesn't slow down at all!

3.3. The Basics of the UltraSound

The only things you need to know to set up the card when your program starts is the maximum # of voices you are going to need and the maximum playback frequency you're going to need for any sample.

Consult the frequency table in section 1.4 to see how many voices you should initialize the card with. If you don't need a large number of voices, initializing the card to use fewer voices means that some 'oversampling' will occur, making the sound better in quality. You cannot initialize for fewer than 14 voices or larger than 32.

To open the card, you call the UltraOpen function. This will set up the interrupt vectors and control procedures for the card, and then reset the card to use the number of voices you specify. When you are finished with the card, you need to close it so that the interrupt procedures, voices, and volumes get stopped and reset to 'normal'. To do this, call the UltraClose function. Please see chapter 4, the reference guide, for more information on the syntax and usage of these routines.

3.4. Using GUS Memory

The UltraSound has a default amount of 256K of memory on the card that can be upgraded in 256K increments up to 1 MB. If you are developing for the card, you may want to upgrade to a full megabyte. Also, you should be aware that many users of the card do NOT have 1 MB of RAM installed, and that if you are going to load samples into the card's memory, you need to check to make sure you have enough room. The function UltraMaxAlloc (additionally for Pascal users the functions UltraMemAvail and UltraMaxAvail) will tell you how much RAM you still have available. Because of the way the GUS memory is structured physically, you cannot allocate chunks larger than 256k at one time.

The C and Pascal versions of the SDK use different methods for keeping track of blocks of memory that have been allocated and blocks that remain free. The C version uses a small portion of GUS RAM to keep track of the blocks, whereas the Pascal version uses a small amount of normal heap space. In any event, you do not have to worry about the internal operation of these routines. To see how the Pascal version uses a small amount of heap space to manage GUS memory, see section 3.14.3.

The 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. To obtain a legal block of UltraSound RAM that is guaranteed to be aligned on a 32-byte boundary, use the UltraMemAlloc function. To free up the memory after you have used it, use the UltraMemFree function. Once again, see chapter 4, the reference guide, for more information on the syntax and usage of these routines.

You DO NOT necessarily need to use the memory allocation and deallocation routines to program the card. You DO have to be careful if you choose to do without them. 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. A DMA to or from the card cannot cross a 256K boundary.

To send data from PC memory to the card, use the UltraDownload function. To read data from the card into PC memory, use the UltraUpload function. If you download data to the card, ensure that if the data is not in two's compliment form you convert it to two's compliment on the download. To do this, use the DMA control bits (see Appendix D). Any data you Upload from the card that is playable will be in two's compliment format.

3.5. What are Samples?

As mentioned above, a 'sample' is raw data that has been recorded using the GUS or another digital sound recording device. Most of these files will have extensions of .SND or .WAV. There are other extensions available, but these are the most common.

A technique known as 'frequency shifting' is sometimes used to produce alternate 'notes' of a specified instrument. Amiga '.MOD' file players use this technique to make a single instrument sample sound differently for each note played over 3 octaves. Essentially, the technique utilizes the 12-note scale with the principle that each half-step is 1/12 root of 2 times the value of its predecessor in frequency (working from left to right). For instance, if we take C as (1*frequency), C# would be (1.05946*frequency). Since there are 12 notes in an octave, when you reach B (the end of the octave), the next note is (2*frequency) - so doubling the original frequency means you raise the note one octave.

There is a drawback to the technique mentioned above. If a sample was recorded at 11kHz for two seconds, and then played back at 22kHz to raise the note one octave, it would only play for one second instead of two. This is because you have (11025 * 2 seconds) = 22050 bytes of data. If you play this at 22050 Hz, or 22050 bytes per second, you will have one 1 second of sound. Thus, if you raise or lower the frequency in this manner, the sound will respectively take a shorter or longer amount of time to play.

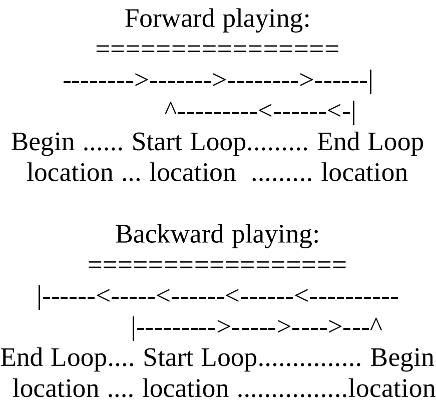
A common problem with samples is that the start and/or end of them is at a reasonably high amplitude. When the sample starts and/or stops, the values flowing through the DACs on the card change suddenly. This usually results in an audible 'click' or 'pop' through the speakers. Section 3.9 looks at the removal of this and other types of clicks and pops than can occur.

3.6. Using Voices

Depending on the number you choose when opening (or initializing) the card, you can use anywhere from 14 to 32 voices at once. You can look at the voices as those of the people in a choir. You can tell the voices to start or stop singing (UltraStartVoice, UltraStopVoice), sing at a specified pitch or frequency (UltraSetFrequency), loudly or softly (UltraSetVolume, UltraSetLinearVolume), and at a specified balance (UltraSetBalance).

The exciting thing about the voices is that any number of them can be played at once, and there is no extra drain on the CPU if you are playing a larger number of voices as opposed to a smaller one. Also, these voices can play any type of sampled sound, 8- or 16-bit, at frequencies up to 44.1 kHz. Any number of voices can even play the same sample at the same time.

To start up a voice, you use the UltraStartVoice or UltraPrimeVoice/ UltraGoVoice combination. Either one of these two routines will ask for a voice number, as well as a begin, start, and end location. The reason begin and start locations are both asked for is in case you wish to play a sample backwards or loop at a specific position which is not the start. The following diagram may make this clearer:



A voice starts at the 'begin' location, and continues playing until it hits the 'end' location. If the 'begin' is a greater location than the 'end', the sample will be played backwards. The 'start' location is the start of the loop if looping is enabled for the voice begin played. Depending on whether and how you wish to loop the voice, you can make various effects occur. Looping of the voices is discussed in the next section.

When you have finished with a voice, You should stop it (UltraStopVoice), and deallocate it (UltraFreeVoice). Please see section 3.9 on click removal for information on problems with stopping a voice abruptly.

3.7. Volumes

A volume is available for each voice. If the volume for a voice is not set to zero, the data at the position of the voice's accumulator will be summed into the final output. Therefore, it is important that if you are not using a voice you should set it's volume to zero.

The UltraSound uses logarithmic volumes using an exponent and a mantissa. For detailed information about how the volumes work, please see section 2.22. As an alternative to using the logarithmic units, a table has been provided in the lowlevel code which gives linear equivalents of the full volume range. You can therefore use the logarithmic volume routines or the linear volume routines - whichever your application requires.

Volumes for each voice may be 'ramped'. This means that you can specify a start and end volume, and the card will smoothly change the volume from the beginning level to the end level. In ramping the volumes, a 'rate' must be specified which assigns how fast the volume is changed. Please see section 2.22 for detailed information on how to set up a volume ramp rate.

3.8. Using Looping

You can loop both voices and volumes on the UltraSound. Looping a voice can result in sustaining a particular note, the generation of a unique sound, or any number of other 'special effects'. Looping the volume can result in a tremolo effect at various intervals.

Looping a sample is a simple task of setting the mode bits when specifying the start of a voice (through UltraPrimeVoice, UltraStartVoice), or when using UltraSetLoopMode. See appendix C for a definition of which bits in the mode byte control looping. Also, Pascal users should be aware of the constants that are available for use instead (see section 3.14).

Looping the volume is essentially as simple as looping a voice. When specifying the volume for a particular voice (using UltraSetVolume or UltraSetLinearVolume), you set the volume mode byte depending on how you wish to loop the volume. See appendix C for a definition of which bits in the mode byte control looping.

3.9. Clicks and click removal

As was mentioned in a few of the preceding sections, 'clicks' and 'pops' can occur 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 voices are trying to play.

In 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. For this reason, it is recommended that a voice be pointed to a location containing a 0 and that its volume be set to 0. At that point, the voice will have no contribution to the output. There are some particular cases where clicks most frequently occur:

During Reset
- Because of the way the card is reset, a pop can occur through the speakers when a reset occurs. The way to remove this pop is to disable the output, reset the card, and then enable the output again.

During a balance sweep
- Since there are only 16 pan positions and there is such a large jump between individual positions, a relatively fast balance sweep from one side to another may produce clicks. You can get a very smooth balance sweep using 2 voices and volume ramping. Set one voice up to one side and one to the other, and ramp one down from volume X to zero at the same rate as you ramp the other from 0 up to volume X. The result is a very smooth balance sweep.

When starting and stopping a voice
- 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.

End points not set properly

- Make sure your end points are at the ends of your samples. It is a very common mistake to set an end point to 1 sample beyond the end. For example, if a sample is 100 bytes long an starts at location 100, the end point is at position 199, NOT position 200. Also, it is possible that the GF1 will interpolate data at points beyond the end point. To ensure that this does not cause 'clicks', use a few extra bytes after the end point of the sample to maintain it.

Loop points not set properly

- The same problem as stated above (end points) is common for loop start and end points. Be sure that the data at the end of the loop and the beginning of the loop are the same, since if there is a large step between them a click will result because the DAC value will change suddenly.

See section 3.11 (Rollover Feature) for removal of clicks during seamless digital playback.

3.10. Interrupt Handling Functions

There are 9 functions (procedures for Pascal users) which 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.

UltraDramTcHandler UltraRecordHandler
UltraMIDIXmitHandler UltraMIDIRecvHandler
UltraTimer1Handler UltraTimer2Handler
UltraWaveHandler UltraVolumeHandler
UltraAuxHandler

All these routines return the old callback address so chaining could be done if desired. This is not usually necessary. It is also not necessary to restore the handler back to the old one when exiting your application. However, the UltraClose function 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 Appendices for the bit definitions.) It is also not necessary to set up a callback since it has a default associated with it.

You should be able to make any SDK library calls from within the interrupt handlers. All the library functions protect themselves from being interrupted while doing critical operations.

Note: You must poll the You MUST service any indicated IRQ's since a read of the IRQ Source Register (8F) 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.

3.11. Rollover feature

Each voice has a 'rollover' feature that allows an application to be notified when a voice's playback position passes over a particular place in DRAM. This is very useful for getting seamless digital audio playback. Basically, the GF1 will generate an IRQ when a voice's current position is equal to the end position. However, instead of stopping or looping back to the start position, the voice will continue playing in the same direction. This means that there will be no pause (or gap) in the playback. Note that this feature is enabled/disabled through the voice's VOLUME control register (since there are no more bits available in the voice control registers). A voice's loop enable bit takes precedence over the rollover. This means that if a voice's loop enable is on, it will loop when it hits the end position, regardless of the state of the rollover enable.
A simple example of this technique is:

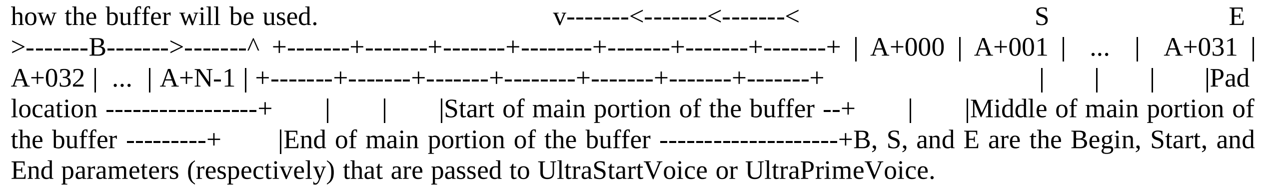
Allocate a chunk of DRAM: 20K, for example.
Load the entire 20K with wave data.
Start up a voice with looping disabled and rollover enabled. Set its end position to the MIDDLE of the buffer.
When the voice hits the middle, you will get an IRQ, but the voice will continue to play.
At this point, enable looping and disable the rollover. Also, set the end position to the end of the buffer. This will make the voice loop back to the beginning without stopping.
Now load the FIRST 10K with more wave data. This will make sure that there is correct data to play when the voice loops.
When the voice loops, you will get and IRQ.
At this point, disable looping, enable rollover and set the end position back to the middle of the buffer.
Now load the SECOND 10K with more wave data. This will make sure that there is correct data to play when the rollover occurs.
Continue in a loop, starting at step #4.

Note: This algorithm does not take care of an initial condition where not enough data exists to fill one complete buffer. It also doesn't address how to finish playing the last incomplete buffer. These points are left up to your implementation to resolve.

Also Note: Changing the start and end points of a voice while its playing can have some strange side effects. For example, if you change end position to a lower location than it is currently playing, you will get an IRQ (if they are enabled). Also, since the high and low bytes are set individually and asynchronously to when the GF1 is working on a voice, it is possible to get an unexpected IRQ if the current position and the new end position cross.

This method of handling seamless digital audio will result in clicking at the end of the buffer. The trick to getting rid of the clicks is to realize that a sample from one end of the buffer must be copied to a "pad location" adjacent to the other end of the buffer. This is to ensure that the samples at the endpoint and the start of the loop are the same. The loop must then be extended by one sample to include this "pad location". The example below demonstrates padding at the beginning of the buffer. This example also assumes 8-bit data so that one sample point fits neatly into one byte, but the technique is just as applicable to 16-bit data.

For streaming digital audio playback, the first step is to allocate the playback buffer. Assume that the SDK routines have been used to allocated a buffer that is at address A and is N bytes long, where N is a multiple of 32. The voice should be set to loop from (A+31) to (A+N-1). Then, every time data are transferred into the end of the buffer, a copy of the sample that will be at location (A+N-1) must be UltraPoked into (A+31). Here is a picture of how the buffer will be used.



B, S, and E are the Begin, Start, and End parameters (respectively) that are passed to UltraStartVoice or UltraPrimeVoice.

Most of the data transfers will be to the main portion of the buffer from locations (A+32) through locations (A+N-1). The main portion of the buffer will most likely be treated as two sub-buffers for double-bufferring. These transfers will be most likely, but not necessarrily, DMA transfers. Most of this is the same as regular double-bufferring. The three things that must be done differently are:1) "Dead space" must be left for the pad location at the beginning of the allocated buffer.2) Whenever a sample is transferred to location (A+N-1), a copy of this sample must be transferred to location (A+31) as well.3) The voice must be set to loop from location (A+31) through location
(A+N-1).

3.12. Stereo playback

Stereo data is represented in an interleaved format, meaning that the data alternates between a sample for the left channel and a sample for the right channel. It is either the responsibility of the hardware or the software to ensure that the data gets used properly. Stereo data can be played back two completely different ways, either under software control or hardware control:

Software Control
================

This is the most flexible method, since it allows for playback at any frequency. The basic algorithm is to de-interleave the data into a left channel buffer and a right channel buffer, then send the data to two independent voices on the card. Here is a brief description of the algorithm that would be used:

Allocate 2 voices. One for left and one for right.
Set the left voice pan position all the way left.
Set the right voice pan position all the way right.
Set both voice volumes to the same value.
Allocate 2 chunks of UltraSound DRAM of the same size, say 20K apiece.
Read in 40K of stereo data.
De-interleave data into 20K of left data and 20K of right data.
DMA left buffer into DRAM buffer #1.
DMA right buffer into DRAM buffer #2.
Prime right voice to loop on all 20K of data forever. No IRQ.
Prime left voice to play 10K of its data with the rollover feature enabled (See discussion on seamless digital playback above).
Start both channels up as close together as possible so they track together. The right channel will not be touched, except to DMA new data into its buffer.
At this point the software will loop on the following steps until the data is exhausted. The rollover will allow the software to play half the buffer and get notified when it has finished, but the playback will continue uninterrupted.
When the rollover IRQ happens, the left voice will be changed to loop at the 20K position. At this point the software will be able to load the next 10K of left channel data into the FIRST 10K of the left channel DRAM. The right channel is also sent 10K of data into its FIRST 10K.
Now each channel has the data to play when they loop at their 20K marks.
When the left channel's loop IRQ happens, both the left and right channel can be sent more data into their SECOND 10K buffer. Also, the left channel's looping will be turned off and it will be told to rollover at its 10K mark again. Now both channel's are ready to play through their 10K marks with no interruption.
Go back to step 13 and loop

The above algorithm does not detail the startup conditions or the shutdown conditions.

Things that you will need to accommodate are:

Less than 1 buffer full of data.
How to terminate when data ends in first buffer. You must allow it to loop back from the end and then stop at the end of the data instead of rolling over at the middle. Both right and left must be taken care of.
How to terminate when data ends in the second buffer. After filling in the data in the second buffer, you can turn off the rolling over at the middle and tell it to stop at the end of the data in the second buffer.

Be sure you understand all the starting and ending possibilities when coding you application.

The basic idea is to let the right voice loop forever on a buffer of data and feed it data based on IRQ generated from the left channel. The left channel can be monitored via the rollover feature to generate and IRQ at the halfway point and an IRQ at the loop point at the end of the buffer to know when the second buffer finishes. Data will then be sent to BOTH voices for the portion of data that it just finished playing.

Hardware Control
================

This method has the advantage of not needing to de-interleave the data before DMAing it into UltraSound DRAM: the data is sent to DRAM in the interleaved format. This means that a voice must be able to play every other sample to correctly separate the right and left channel. It is possible to set up the GF1 and the 2 voices to do this, however, it will only work for 2 usable frequencies. (44100 and 22050). Any other frequencies must be done using the software control de-interleaving method.

The theory behind how to achieve this is quite complex. It involves using the # of active voices and the voice's frequency control register to get it to grab the correct samples at the proper frequency. Here is the setup necessary to playback 44100 Hz stereo and 22050 Hz stereo data:

44100 Hz: Set the # of active voices to 14 and set both voices playback rate to 88200 hz.

22050 Hz: Set the # of active voices to 28 and set both voices playback rate to 44100 hz

These two configurations will make both voices have a frequency control register equal to 2. The frequency control register holds the amount added to the current voice playback location to get the next playback location.

Normally, this number is a fraction so interpolation is done to generate intermediate points between successive locations. With interleaved data, we don't want to interpolate since the neighboring data points belong to the other channel. We want to completely skip over the point after the current one, and go to our current position plus 2.

The number of active voices selected will determine the update rate. The more voices that are active, the longer time between updates, thus slowing down the frequency. 14 active voices will give and update rate of about 22.5 microseconds (44100 hz), whereas 28 voices will give and update rate of about 45 microseconds (22050 hz).

The initialization of the voices in this mode will obviously be different. Essentially, the voices must be set up to play the same data but offset their start and end points by one or two bytes, depending on whether it is an 8- or 16-bit sample.

3.13. C-specific information

Currently, 10 different C memory model/compiler configurations are supplied. Large, medium, small and tiny models for Borland and Microsoft compilers are available, as well as flat model for WatCom and MetaWare compilers. The level 0 library 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 codec libraries contain the functions used with the CS4231. Borland C++ 2.0 and 3.1, Microsoft C 6.0, Watcom C9.0/386, and Metaware High C/C++ were used to test the library routines. These are the naming conventions for the C libraries:

ultra0lb.lib Level 0, large model, Borland C
ultra1lb.lib Level 1, large model, Borland C
ult16lb.lib Codec, large model, Borland C

ultra0mb.lib Level 0, medium model, Borland C
ultra1mb.lib Level 1, medium model, Borland C
ult16mb.lib Codec, medium model, Borland C

ultra0sb.lib Level 0, small model, Borland C
ultra1sb.lib Level 1, small model, Borland C
ult16sb.lib Codec, small model, Borland C

ultra0tb.lib Level 0, tiny model, Borland C
ultra1tb.lib Level 1, tiny model, Borland C
ult16tb.lib Codec, tiny model, Borland C

ultra0lm.lib Level 0, large model, Microsoft C
ultra1lm.lib Level 1, large model, Microsoft C
ult16lm.lib Codec, large model, Microsoft C

ultra0mm.lib Level 0, medium model, Microsoft C

ultra1mm.lib Level 1, medium model, Microsoft C
ult16mm.lib Codec, medium model, Microsoft C

ultra0sm.lib Level 0, small model, Microsoft C
ultra1sm.lib Level 1, small model, Microsoft C
ult16sm.lib Codec, small model, Microsoft C

ultra0tm.lib Level 0, tiny model, Microsoft C
ultra1tm.lib Level 1, tiny model, Microsoft C
ult16tm.lib Codec, tiny model, Microsoft C

ultra0wc.lib Level 0, Flat model, Watcom C
ultra1wc.lib Level 1, Flat model, Watcom C
ult16wc.lib Codec, Flat model, Watcom C

ultra0mw.lib Level 0, Flat model, Metaware High C/C++
ultra1mw.lib Level 1, Flat model, Metaware High C/C++
ult16mw.lib Codec, Flat model, Metaware High C/C++

Several example applications are supplied on the tool kit 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.

3.14. PASCAL-specific information

Version 2.11 of the UltraSound SDK is the first version which has full support for Turbo Pascal 6.0 and 7.0, as well as Borland Pascal 7.0. For those of you who are curious, Borland Pascal 7.0 is the professional version of Turbo Pascal 7.0. It costs more, but includes a lot of bells and whistles that are not included in the regular 'Turbo' version (for more information, contact Borland).

The translation was done entirely by Kurt Kennett of Ingenuity Software, and is a direct translation of the C tool kit. The code was written in Borland Pascal 7 and then minor adjustments were made to make it compile under Turbo 6 and 7.

Due to the fact that the Pascal translation was done after the C code was finished, some of the naming conventions for some of the functions and record types reflects a 'C-like' syntax and style. Some conversion of function result types has been done to reflect a more Pascal-like structure.

A short section detailing specific information about types and constants available follows.

NOTE: Version 2.11 is the latest release of the PASCAL version of the SDK. The PASCAL version has NOT been updated to include ICS mixer or CS4231 functions.

3.14.1. Available constants and variables

The SDK TPU (or 'driver') has some predefined constants to assist you in programming various aspects of the card. There are three common frequencies that sampled sounds are played at: 11, 22, and 44 KHz:

```
Khz_11 = 11025;
Khz_22 = 22050;
Khz_44 = 44100;
```

Whenever volume is specified, there is a volume control byte. The bits in this byte control several important functions. OR the following constants together with zero if you need them. If you don't, just specify zero as the volume control byte:

```
Loop_Volume           = 08;
Bi_Directional_Volume = 16;
Enable_Volume_Handler = 32;
```

Whenever a voice is started or changed, there is a voice control byte. The bits in this byte control several important functions for each voice. OR the following constants together with zero if you need them. If you don't, just specify zero as the voice control byte:

```
Voice_Data_16Bit       = 04;
Loop_Voice             = 08;
Bi_Directional_Voice   = 16;
Enable_VoiceEnd_Handler = 32;
```

When a DMA transfer is started, you need to tell the DMA controller what type of data you are sending to the UltraSound, as well as if the controller should convert the data to 2's complement. If you are downloading ram .SND files to the card, you need to OR the Convert_Data constant to 0 as the control byte. If you are downloading 16 Bit data, you need to OR the control byte with DMA_Data_16Bit.

```
DMA_Data_16Bit = 64;
Convert_Data   = 128;
```

There are several predefined variables to help you manage the UltraSound card as you use it. These variables determine the Configuration, error codes, and whether or not the card is installed:

Ultra_Installed : BOOLEAN;
If the driver cannot read the environment variable 'ULTRASND' when it is initialized (i.e. when your program starts), this BOOLEAN will remain false. Otherwise, the variable Ultra_Config (see below) will be set up with the values found in the environment variable.

Ultra_Config : Ultra_CFG;
If the driver was able to read a configuration from the 'ULTRASND' environment variable, this variable record will be filled in with the appropriate information found in the environment. If the environment string is not found, this variable will assume a default configuration. All of the functions and procedures in the driver use this configuration to initialize and use the card. Be careful if you're going to go sticking values into this record. The main use of this variable is to display the active configuration to the user.

The driver predefines three variables to help you with error control:

UltraOk : Boolean;
This boolean always maintains the status of the last UltraSound procedure or function. If there was a problem with the routine, this variable will be FALSE. Otherwise, it will be TRUE (indicating no error).

UltraError : Integer;
This is the "Error Code" of the Error. This variable is set to 0 if no error has occurred, and is only set when UltraOk becomes FALSE.

UltraErrorStr : String;
This is the "description" of the error which has occurred. As an added bonus to you, the programmer, this string is set with a (descriptive) error message whenever a card-related error occurs.

3.14.2. Examples

There are several example programs included with the units described in the previous section:

GUSINIT.PAS
This example file is a small unit that you may want to use to initialize the GUS before you use it. The important thing about this unit is that it installs an 'Exit Procedure' which is called upon program termination (normal or crash-out). The exit procedure closes down the card before returning to DOS (or the IDE, depending upon where you are). In TP and BP 7, you can type 'ExitProc' into the IDE, move the cursor on top of it, and hit CTRL-F1 for a detailed explanation of what an exit procedure is. After including this unit in your 'uses' clause, your main program can just go ahead and start using it. You don't need to call UltraOpen, UltraReset, or even UltraClose when you're finished.

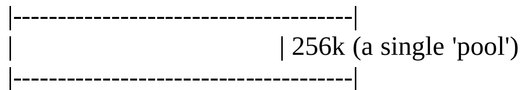
GUS1.PAS
This is the primitive example program for the card which loads three sounds (a laser blast, a photon torpedo, and a missile) into the card's RAM and then allows you to press the '1'..'3' keys to play them. This example uses the GUSINIT unit - showing you how easy it is to use.

LOADMOD.PAS
This is a unit which provides services to load Amiga '.MOD' files into GUS RAM and main memory. After calling a single function, the file will have been loaded and the samples contained in it set up to play. It is left up to you to supply playback routines or trivial sample-playing routines.

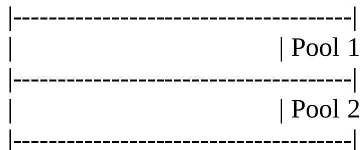
3.14.3. Management of GUS RAM.

The Pascal version of the SDK employs a different approach to memory control than the C tool kit. In the C SDK, a small amount of the GUS RAM is used to keep track of allocated blocks. In the Pascal SDK, a very small amount of heap space is used to keep track of the blocks.

Because of the way the UltraSound RAM is structured, any particular card has a total amount of installed RAM that is a multiple of 256k. Since samples cannot be played across these boundaries, the driver unit divides the total RAM by 256k into 1 to 4 'pools'.

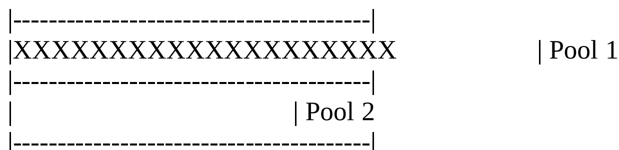


Each of these pools is seen as a single unit which can be subdivided. For the purposes of this example, we will use a GUS which has 512k installed. If this is the case, there will be two pools of RAM available:



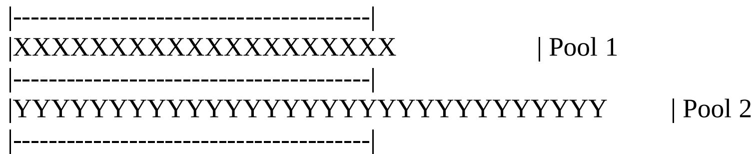
Once the card has been initialized, there will be a small amount of heap space being used to keep track of these pools (usually less than 100 bytes). At this point, the UltraMaxFree function will return 256k, since the size of the largest block that can be allocated is one entire pool. The UlraMemAvail function will return 512k however, reflecting the total amount of memory still available.

If a user were to allocate a 128k chunk of memory, the allocation routines would look in each pool sequentially until a pool is found with enough room for the new block. At this point in our example, it would use Pool 1.

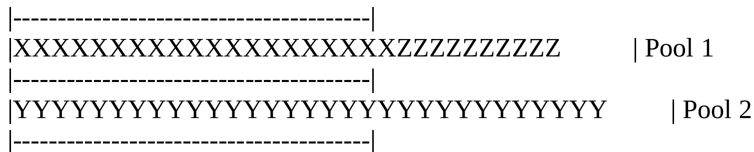


The X's in the diagram above reflect used space. If the size of the block being allocated is not divisible by 32, the actual size of the block in memory is increased to the next multiple of 32. For example, if I were to ask for a 50 byte chunk of RAM, the actual size of ram that I would be using would be 64 bytes in size. This is so that all memory locations returned will be aligned on a 32-byte boundary. At this point, the UltraMaxFree function will return 256k, since the size of the largest block that can be allocated is still one entire pool (Pool 2). The UltraMemAvail function will return 384k, reflecting the total amount of memory still available.

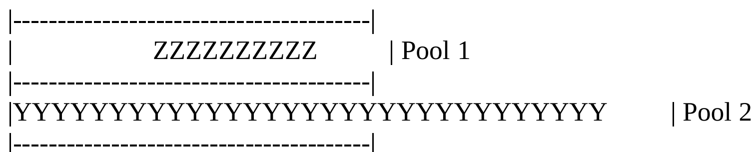
If a user were to now allocate a 200k chunk of memory, the allocation routines would once again look in each pool sequentially until a pool is found with enough room for the new block. At this point in our example, it would have to use Pool 2, since there is not enough room left in Pool 1.



At this point, the UltraMaxFree function would return 128k, since the largest block free is in Pool 1. UltraMemFree would now return 184k. If we were to allocate a third block of 70k, the diagram would look like:



If we were to now Deallocate the first block (the X's in the diagram), we would be left with a 'hole' in the ram. This is reminiscent of a hard disk's free space fragmentation.



Note that the SDK's memory allocation routines do NOT relocate samples to fend off fragmentation. Note also that the memory routines are set up and are ready to be used as soon as you call UltraOpen, and are closed down automatically (and the heap space they use deallocated) when you call UltraClose.

3.15. Technical Support

Questions regarding the SDK can be sent to:

Advanced Gravis
Attn: Developer Support
North Fraser Way
Burnaby, British Columbia V5J 5E9
FAX (604) 451-9358

Please make sure you have consulted this documentation before contacting technical support. (RTFM please)

4. Chapter 4 - Reference Guide

This chapter is a reference guide for the routines that comprise both the C and Pascal versions of the SDK. Please see the table of contents to look up a particular routine quickly by name.

When using the following functions, be sure to include:
forte.h
gf1proto.h
extern.h
ultraerr.h

4.1. UltraCalcRate

Purpose: To calculate the rate for a volume ramp.

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

```
PASCAL:FUNCTION UltraCalcRate(StartV   : WORD;
                              EndV     : WORD;
                              Mil_Secs : LONGINT) : BYTE;
```

Remarks: This routine calculates the rate necessary to ramp the volume from StartV volume (logarithmic) to EndV volume (logarithmic) 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: A value which can be passed to UltraRampVolume or UltraRampLinearVolume.

See also: UltraRampVolume, UltraRampLinearVolume

4.2. UltraClose

Purpose: To close out the UltraSound card.

```
int UltraClose(void);
```

```
PASCAL:FUNCTION UltraClose : BOOLEAN;
```

Remarks: This routine 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 (resets interrupt vectors etc).

Returns: TRUE if Close was successful, FALSE otherwise.

See also: UltraOpen, UltraReset

4.3. UltraDownload

Purpose: Download a chunk of data into UltraSound DRAM.

```
int UltraDownload(dataptr,control,dram_loc,len,wait);
    void *dataptr;
    unsigned char control;
    unsigned long dram_loc;
    unsigned int len;
    int wait;
```

```
PASCAL:FUNCTION UltraDownLoad(DataPtr  : POINTER;
                              Control  : BYTE;
                              DRAM_Loc : LONGINT;
                              Len      : WORD;
                              Wait     : BOOLEAN) : BOOLEAN;
```

Remarks: This routine will transfer a chunk of data from the PC's RAM to the UltraSound's DRAM. It will transfer 'Len' # of bytes from DataPtr (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 some cases where 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 transferred.

Returns: C: ULTRA_OK if no problem.
DMA_BUSY if DMA Channel is busy.

PASCAL: TRUE if transfer was successful. FALSE otherwise.
If unsuccessful, check UltraErrorStr for the reason.

See also: UltraUpload, UltraDRAMDMABusy

NOTE METAWARE USERS: Due to a bug in the Metaware libraries, UltraDownload can not use dma transfers. UltraDownload pokes the data into GUS DRAM instead, which is very slow. Metaware has release a fix for their libraries, which is available on their BBS. A fix to UltraDownload that takes advantage of the fixed Metaware libraries will be available on our Gravis BBS shortly.

4.4. UltraDRAMDMABusy

Purpose: Test to see if the DMA channel is busy.

```
int UltraDramDmaBusy(void);
```

PASCAL:FUNCTION UltraDRAMDMABusy : BOOLEAN;

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

Returns: TRUE if the channel is still busy. FALSE if it's free.

See also: UltraWaitDRAMDMA, UltraDownload, UltraUpload

4.5. UltraGoRecord

Purpose: To start up a pre-set record.

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

PASCAL:FUNCTION UltraGoRecord(Control : BYTE) : BOOLEAN;

Remarks: This routine will start up a pre-primed record (done with UltraPrimeRecord). It can also be used to restart a indefinite recording process from the recording handler callback.

Returns: C: ULTRA_OK if no error
DMA_BUSY if the channel is busy

PASCAL: TRUE if Record started ok. FALSE otherwise.
If FALSE check the UltraErrorStr for the reason.

See also: UltraRecordData, UltraPrimeRecord, UltraRecordHandler

4.6. UltraGoVoice

Purpose: To start a voice that has already been primed.

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

```
PASCAL:PROCEDURE UltraGoVoice(Voice : INTEGER;
                        VMode : BYTE);
```

Remarks: This routine will start up a voice that has already been primed with set-up 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.

See also: UltraPrimeVoice, UltraStartVoice

4.7. UltraDisableLineIn

Purpose: To disable line level input.

```
void UltraDisableLineIn(void);
```

PASCAL:PROCEDURE UltraDisableLineIn;

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.

See also: UltraEnableLineIn, UltraGetLineIn

4.8. UltraDisableMicIn

Purpose: To disable microphone input.

```
void UltraDisableMicIn(void);
```

PASCAL:PROCEDURE UltraDisableMicIn;

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.

See also: UltraEnableMicIn, UltraGetMicIn

4.9. UltraDisableOutput

Purpose: To turn off all output from the UltraSound.

```
void UltraDisableOutput(void);
```

PASCAL:PROCEDURE UltraDisableOutput;

Remarks: This routine 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 since that will help eliminate 'pops' during initialization.

See also: UltraEnableOutput, UltraGetOutput

4.10. UltraEnableLineIn

Purpose: To enable line level input.

```
void UltraEnableLineIn(void);
```

PASCAL:PROCEDURE UltraEnableLineIn;

Remarks: Turns 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).

See also: UltraDisableLineIn, UltraGetLineIn

4.11. UltraEnableMicIn

Purpose: To turn on the microphone input.

```
void UltraEnableMicIn(void);
```

PASCAL:PROCEDURE UltraEnableMicIn;

Remarks: This routine 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, since it will reduce noise on the output.

See also: UltraDisableMicIn, UltraGetMicIn

4.12. UltraEnableOutput

Purpose: To enable output from the UltraSound.

```
void UltraEnableOutput(void);
```

PASCAL:PROCEDURE UltraEnableOutput;

Remarks: This routine must be called to enable any output from the UltraSound. This can be used to turn on output after muting it with UltraDisableOutput.

See also: UltraDisableOutput, UltraGetOutput

4.13. UltraGetLineIn

Purpose: To return the current state of line level input

```
int UltraGetLineIn(void);
```

PASCAL:FUNCTION UltraGetLineIn : BOOLEAN;

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: TRUE if LineIn is Enabled, FALSE otherwise.

See also: UltraEnableLineIn, UltraDisableLineIn

4.14. UltraGetOutput

Purpose: To return the current output enabled state.

```
int UltraGetOutput(void);
```

PASCAL:FUNCTION UltraGetOutput;

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: TRUE if Output is Enabled, FALSE otherwise.

See also: UltraEnableOutput, UltraDisableOutput

4.15. UltraGetMicIn

Purpose: To return the current state of the microphone input.

```
int UltraGetMicIn(void);
```

PASCAL:FUNCTION UltraGetMicIn;

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: TRUE if Output is Enabled, FALSE otherwise.

See also: UltraEnableMicIn, UltraDisableMicIn

Callbacks

When coding the callbacks, it is important that you remember that these routines will be called from within interrupts. This means you cannot do any DOS calls from the routines. Stack checking may also cause problems.

NOTE: Certain dos extenders do not reflect interrupts from real mode to the protected mode handler if high IRQ (ie >= 8) are used. This will cause problems since any interrupt generated in real mode will be lost.

4.16. UltraDRAMTcHandler

Purpose: To define a callback for a DMA transfer completion.

```
PFV UltraDramTcHandler(handler);
    PFV handler;
```

PASCAL:PROCEDURE UltraDRAMTcHandler(VAR Handler : PFV);

Remarks: This procedure defines a callback for whenever a DMA transfer to the UltraSound has been completed. The routine address passed as a parameter is set as the new handler. No parameters are passed to your new handler.

Returns: For C users, this routine returns the address of the old handler.

PASCAL
Example: Since PASCAL cannot return function or procedure type values from functions, the equivalent routine was made into a procedure which took a single VAR parameter. Thus, when you are going to change the address of a callback routine, you put the new address into a variable, run it through the procedure, and what will then be in the variable is the address of the old handler:

```
VAR
  OldProc : PFV;

Procedure MyNewProc;
  { The new handler - defined as a normal procedure }
  begin
    ...
  end;


...
{ Set OldProc to address of new handler }
OldProc := MyNewProc;
{ Now set the new handler and return the old one }
UltraDRAMTcHandler(OldProc);
...
```

At this point, OldProc holds the address of the old routine. Therefore, to chain to the old routine, you would simply have to call OldProc from your handler.

4.17. UltraMIDIXmitHandler

Purpose: To define a callback for MIDI transmit interrupt.

```
PFV UltraMIDIXmitHandler(handler);
      PFV handler;
```

PASCAL:PROCEDURE UltraMIDIXMitHandler(VAR Handler : WORD_PROC);

Remarks: This procedure defines a callback for whenever a MIDI transmit empty interrupt occurs. This can be used to send out MIDI data under interrupt control. The routine address passed as a parameter is set as the new handler. The MIDI Status Byte is passed to the new handler defined, therefore the handler should have a prototype of the following form:

void handler (unsigned int midiStatus);

Returns: For C users, this routine returns the address of the old handler.

PASCAL
Example: See the example for UltraDRAMTcHandler.

See also: UltraMIDIRecvHandler

4.18. UltraMIDIRecvHandler

Purpose: To define a callback for a MIDI receive interrupt.

```
PFV UltraMIDIRecvHandler(handler);
      PFV handler;
```

PASCAL:PROCEDURE UltraMIDIRecvHandler(VAR Handler : TWOWORD_PROC);

Remarks: This procedure defines a callback for whenever a byte is received in the MIDI input port. This can be used to get data from the MIDI port under interrupt control. The routine address passed as a parameter is set as the new handler. The MIDI port status and MIDI data are passed to your handler. The status bits are defined in Appendix D.

The handler should have a prototype of the following form:

void handler (unsigned int midiStatus, unsigned int midiData);

Returns: For C users, this routine returns the address of the old handler.

PASCAL
Example: See the example for UltraDRAMTcHandler.

See also: UltraMIDIXmitHandler

4.19. UltraTimer1Handler

Purpose: To define a callback from Timer #1.

```
PFV UltraTimer1Handler(handler);
    PFV handler;
```

PASCAL:PROCEDURE UltraTimer1Handler(VAR Handler : PFV);

Remarks: This procedure defines a callback for whenever the UltraSound's Timer 1 'Ticks'. The routine address passed as a parameter is set as the new handler. No parameters are passed to this handler.

Returns: For C users, this routine returns the address of the old handler.

PASCAL
Example: See the example for UltraDRAMTcHandler.

See also: UltraTimer2Handler, UltraStartTimer

4.20. UltraTimer2Handler

Purpose: To define a callback for Timer #2.

```
PFV UltraTimer2Handler(handler);
    PFV handler;
```

PASCAL:PROCEDURE UltraTimer2Handler(VAR Handler : PFV);

Remarks: This procedure defines a callback for whenever the UltraSound's Timer 2 'Ticks'. The routine address passed as a parameter is set as the new handler. No parameters are passed to this handler.

Returns: For C users, this routine returns the address of the old handler.

PASCAL
Example: See the example for UltraDRAMTcHandler.

See also: UltraTimer1Handler, UltraStartTimer

4.21. UltraWaveHandler

Purpose: To define a callback for an end-of-wave interrupt.

```
PFV UltraWaveHandler(handler);
      PFV handler;
```

PASCAL:PROCEDURE UltraWaveHandler(VAR Handler : INT_PROC);

Remarks: This procedure defines a callback for whenever a voice generates a wavetable interrupt. This happens when a voice hits it's end and interrupts are enabled. It will happen even if looping is on (i.e. the voice keeps playing). The routine address passed as a parameter is set as the new handler.

Normally, This procedure is used to signify that a voice is done playing. This handler can be useful for starting another voice or counting the # of times that a voice goes through a loop. The voice # that generated the interrupt is passed back to your handler as an unsigned int.

Returns: For C users, this routine returns the address of the old handler.

PASCAL
Example: See the example for UltraDRAMTcHandler.

4.22. UltraVolumeHandler

Purpose: To define a callback for volume ramp complete interrupt.

```
PFV UltraVolumeHandler(handler);
      PFV handler;
```

PASCAL:PROCEDURE UltraVolumeHandler(VAR Handler : INT_PROC);

Remarks: This procedure defines a callback for whenever a volume ramp ends. The routine address passed as a parameter is set as the new handler. This routine 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 interrupt will be passed back to your handler as an unsigned int.

Returns: For C users, this routine returns the address of the old handler.

PASCAL

Example: See the example for UltraDRAMTcHandler.

4.23. UltraRecordHandler

Purpose: To define a callback for a DMA record complete interrupt.

```
PFV UltraRecordHandler(handler);
     PFV handler;
```

PASCAL:PROCEDURE UltraRecordHandler(VAR Handler : PFV);

Remarks: This routine is called when a buffer that was being recorded into is full. The routine address passed as a parameter is set as the new handler. Normally, this procedure would be used to let the application start up another record. A double buffering scheme could be used to record data continuously. No parameters are 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 your application may have a problem with throughput.

Returns: For C users, this routine returns the address of the old handler.

PASCAL
Example: See the example for UltraDRAMTcHandler.

4.24. UltraAuxHandler

Purpose: To define a callback for an auxiliary interrupt.

```
PFV UltraAuxHandler(handler);
     PFV handler;
```

PASCAL:PROCEDURE UltraAuxHandler(VAR Handler : PFV);

Remarks: This handler will be called at the end of ALL interrupts that happen on the UltraSound. Its primary purpose is for use with the new UltraMax card since its shares its IRQ with the GF1.

Returns: For C users, this routine returns the address of the old handler.

PASCAL
Example: See the example for UltraDRAMTcHandler.

4.25. UltraMaxAlloc

Purpose: To find the size of the largest allocatable block of UltraSound DRAM.

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

PASCAL:FUNCTION UltraMaxAlloc : LONGINT;

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

Returns: # of bytes in largest available block.

See also: UltraMemAlloc, UltraMemFree,
for C users UltraMemInit
and for PASCAL users UltraMemAvail and UltraMaxAvail

4.26. UltraMaxAvail

Purpose: To find the size of the largest allocatable block of UltraSound DRAM.

Use the UltraMaxAlloc function.

PASCAL:FUNCTION UltraMaxAvail : LONGINT;

Remarks: This routine will return the largest block of DRAM (in bytes) that can still be allocated. This can be useful for determining whether or not a patch or sample can be loaded. The maximum size of a block is 256K. This routine is included to provide consistency of naming for PASCAL Programmers.

Returns: # of bytes in largest available block.

See also: UltraMemAlloc, UltraMemFree,
for C users UltraMemInit
and for PASCAL users UltraMemAvail

4.27. UltraMemAvail

Purpose: To find the total amount of free UltraSound DRAM.

This function is not available.

PASCAL:FUNCTION UltraMemAvail : LONGINT;

Remarks: For PASCAL users, this will return the total amount of DRAM still available for use on the UltraSound.

Returns: # of bytes in left available.

See also: UltraMemAlloc, UltraMemFree, UltraMaxAlloc
for C users UltraMemInit
and for PASCAL users UltraMaxAvail

4.28. UltraMemAlloc

Purpose: To safely allocate a chunk of UltraSound DRAM.

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

```
PASCAL:FUNCTION UltraMemAlloc( Size     : LONGINT;
                           VAR Location : LONGINT) : BOOLEAN;
```

Remarks: This routine allocates a chunk of DRAM of 'Size' bytes from the UltraSound's DRAM. The memory allocation structures are set up by UltraOpen. 'Location' is filled in with the DRAM location of the start of the chunk of memory. The memory address returned will ALWAYS be aligned on a 32 byte boundary so that the DRAM can be DMA'ed into without error. Also, the size will be rounded UP to the next 32 byte boundary. PASCAL users can reference section 3.14.3 for a detailed look at how the memory is managed.

Returns: C: ULTRA_OK if no problem.
NO_MEMORY if there is no chunk of DRAM large enough.

PASCAL: TRUE if the allocation was successful.
If FALSE, check UltraErrorStr for the reason.

See also: UltraMemFree, UltraMaxAlloc,
for C users UltraMemInit
and for PASCAL users UltraMaxAvail and UltraMemAvail

4.29. UltraMemFree

Purpose: To free a chunk of DRAM previously allocated with UltraMemAlloc.

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

```
PASCAL:FUNCTION  UltraMemFree(Size     : LONGINT;
                          Location : LONGINT) : BOOLEAN;
```

Remarks: Frees a previously allocated chunk of UltraSound memory. The size will

automatically be rounded UP to the next 32 byte boundary.

Returns: C: ULTRA_OK if no problem.
CORRUPT_MEM if the memory structures are corrupted.

PASCAL: TRUE if deallocation was successful.
If FALSE, check UltraErrorStr for the reason.

See also: UltraMemInit, UltraMemAlloc
for C users UltraMemInit
and for PASCAL users UltraMaxAvail and UltraMemAvail

4.30. UltraMemInit

Purpose: To initialize or re-initialize the memory pool structures.

```
unsigned long UltraMemInit(void);
```

PASCAL:This routine is not used. The unit automatically calls these routines when your program starts.

Remarks: This routine 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 greater than 32 bytes and less than 256K in size.

See also: UltraMemAlloc, UltraMemFree, and UltraMaxAlloc

4.31. UltraMidiDisableRecv

Purpose: To disable the MIDI receive data interrupt.

```
void UltraMidiDisableRecv(void);
```

PASCAL:PROCEDURE UltraMIDIDisableRecv;

Remarks: This routine will disable the receive data interrupts from the MIDI. If the interrupt is enabled, it should be disabled before leaving your application.

See also: UltraMIDIEnableRecv, UltraMIDIRecvHandler

4.32. UltraDisableMIDIXmit

Purpose: To disable MIDI transmit interrupts.

```
void UltraMidiDisableXmit(void);
```

PASCAL:PROCEDURE UltraDisableMIDIXmit;

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

See also: UltraMIDIXmitHandler, UltraMIDIEnableXmit

4.33. UltraMIDIEnableRecv

Purpose: To enable receive data interrupts for MIDI port.

```
void UltraMidiEnableRecv(void);
```

PASCAL:PROCEDURE UltraMIDIEnableRecv;

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

See also: UltraMIDIRecvhandler, UltraMIDIDisableRecv

4.34. UltraEnableMIDIXmit

Purpose: To enable transmit interrupts from MIDI.

```
void UltraMidiEnableXmit(void);
```

PASCAL:PROCEDURE UltraEnableMIDIXmit;

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

See also: UltraMIDIXmitHandler, UltraMIDIDisableXmit

4.35. UltraMIDIRecv

Purpose: To read a byte from the MIDI port.

```
unsigned char UltraMidiRecv(void);
```

PASCAL:FUNCTION UltraMIDIRecv : BYTE;

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

Returns: MIDI data byte

See also: UltraMIDIRecvHandler, UltraMIDIStatus

4.36. UltraMIDIReset

Purpose: To reset the MIDI port.

```
void UltraMidiReset(void);
```

PASCAL:PROCEDURE UltraMIDIReset;

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

See also: UltraEnableMIDIXmit, UltraEnableMIDIRecv

4.37. UltraMIDIStatus

Purpose: To read the MIDI status byte.

```
unsigned char UltraMidiStatus(void);
```

PASCAL:FUNCTION UltraMIDIStatus : BYTE;

Remarks: This routine returns the current MIDI port status 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 MIDI status byte. See chapter 2.

See also: UltraMIDIXmit, UltraMIDIRecv

4.38. UltraMIDIXmit

Purpose: To send a byte out the MIDI port.

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

PASCAL:PROCEDURE UltraMIDIXmit(Data : BYTE);

Remarks: This routine will send the 'Data' byte out the MIDI data port. If interrupts are enabled, an interrupt will be generated when the Byte has been transmitted.

See also: UltraMIDIXmitHandler

4.39. UltraOpen

Purpose: To open and initialize the UltraSound card and the SDK code.

```
int UltraOpen(config,voices);
     ULTRA_CFG *config;
     int voices;
```

```
PASCAL:FUNCTION  UltraOpen(VAR Config : Ultra_CFG;
                          Voices : INTEGER) : BOOLEAN;
```

Remarks: This routine 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 over sampling that occurs on playback. This will make the sound much 'cleaner'. If you specify a number of voices less than 14, the card will still be initialized to use 14 voices. Likewise, if you select larger than 32 voices, you will still only have 32.

Returns: C: ULTRA_OK if no problem.
NO_ULTRA if no UltraSound card found.
BAD_NUM_OF_VOICES if # of active voices out of range

PASCAL: TRUE if the card was successfully opened.
If FALSE, check UltraErrorStr for the reason.

See also: UltraClose, UltraProbe

4.40. UltraPeekData

Purpose: To examine any DRAM location on the UltraSound.

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

```
PASCAL:FUNCTION UltraPeekData(PPort   : INTEGER;
                      Address : LONGINT) : BYTE;
```

Remarks: This routine is used to allow an application to look at any location in UltraSound's

DRAM. This can be handy for obtaining VU information or any other time it is nice to know what is in DRAM. Be aware that if the data is playable it will be in twos compliment form. If the data that you want is 16 bit data, you will need to 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 specified.

See also: UltraPokeData, UltraUpload

4.41. UltraPing

Purpose: To quickly check to see if an UltraSound is present at a specified port.

```
int UltraPing(baseport);
      int baseport;
```

PASCAL:FUNCTION UltraPing(PPort : WORD) : BOOLEAN;

Remarks: This routine will determine if an UltraSound is present by attempting to read and write to its DRAM. This routine assumes that at least a simple reset has been done since power-up so that the board is no longer in a reset state. If it is on a reset state, this routine will ALWAYS fail. UltraProbe pulls a quick reset and then calls UltraPing.

Returns: C: ULTRA_OK if no problem.
NO_ULTRA if no board is found at specified I/O port.

PASCAL: TRUE if the card was found. FALSE otherwise.
If unsuccessful, check UltraErrorStr for the reason.

See also: UltraProbe

4.42. UltraPokeData

Purpose: To poke a byte into the UltraSound's DRAM.

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

```
PASCAL:PROCEDURE UltraPokeData(PPort   : INTEGER;
                               Address : LONGINT;
                               Data    : BYTE);
```

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 automatic conversion of data poked into DRAM. Since the UltraSound can only play twos compliment data, make sure that the data you are poking is in that format.

Also be careful with 16 bit data.

See also: UltraPeekData, UltraDownload

4.43. UltraPrimeRecord

Purpose: To prime the record DMA channel for use.

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

```
PASCAL:FUNCTION UltraPrimeRecord(PC_Ptr  : POINTER;
                                 Size    : WORD;
                                 RRepeat : BOOLEAN) : BOOLEAN;
```

Remarks: This routine will setup the DMA channel to do a record, but does 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 routine will help alleviate this problem by doing the programming ahead of time.

Returns: C: ULTRA_OK if no problem.
DMA_BUSY if DMA Channel is busy.

PASCAL: TRUE if setup was successful. FALSE otherwise.
If unsuccessful, check UltraErrorStr for the reason.

See also: UltraGoRecord

4.44. UltraPrimeVoice

Purpose: To prime a voice with values but NOT start it.

```
unsigned char UltraPrimeVoice(voice,begin,start,end,mode);
    int voice;
    unsigned long begin;
    unsigned long start;
    unsigned long end;
    unsigned char mode;
```

```
PASCAL:FUNCTION UltraPrimeVoice(Voice    : INTEGER;
                                VBegin   : LONGINT;
                                VStart   : LONGINT;
```

```
                VEnd    : LONGINT;
                VMode   : BYTE) : BYTE;
```

Remarks: This routine 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 then immediately 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

4.45. UltraProbe

Purpose: To probe for the existence of an UltraSound at a specified port.

```
int UltraProbe(base_port);
    unsigned int base_port;
```

PASCAL:FUNCTION UltraProbe(PPort : WORD) : BOOLEAN;

Remarks: This routine 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 assumes this has already been done.

Returns: C: ULTRA_OK if no problem.
NO_ULTRA if no card was found at the port specified.

PASCAL: TRUE if the card is found. FALSE otherwise.
If unsuccessful, check UltraErrorStr for the reason.

See also: UltraOpen, UltraPing

4.46. UltraRampVolume

Purpose: To ramp a voice's volume up or down.

```
void UltraRampVolume(voice,start,end,rate,mode);
    int voice;
    unsigned int start;
    unsigned int end;
    unsigned char rate;
    unsigned char mode;
```

```
PASCAL:PROCEDURE UltraRampVolume(Voice  : INTEGER;
                        StartV : WORD;
                        EndV   : WORD;
                        VRate  : BYTE;
                        VMode  : BYTE);
```

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 routine to call for a volume interrupt. This is done with the routine UltraVolumeHandler. The volume mode bits are defined in Appendix B.

See also: UltraCalcRate, UltraVolumeHandler

4.47. UltraReadRecordPosition

Purpose: To return the # of bytes recorded so far in a record.

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

PASCAL:FUNCTION UltraReadRecordPosition : WORD;

Remarks: This routine can be used to monitor the amount data recorded.

Returns: Number of BYTES recorded so far.

See also: UltraRecordData

4.48. UltraReadVoice

Purpose: To read a voice's current location.

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

PASCAL:FUNCTION UltraReadVoice(Voice : INTEGER) : LONGINT;

Remarks: This routine can be used to monitor a voice's progress.

Returns: The voice's current position in DRAM.

See also: UltraSetVoice

4.49. UltraReadVolume

Purpose: To read a voice's current volume.

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

PASCAL:FUNCTION UltraReadVolume(Voice : INTEGER) : WORD;

Remarks: This routine returns the current volume of a voice. This can be useful when used in conjunction with volume ramps. The value returned is logarithmic, not linear. PASCAL users can use UltraReadLinearVolume.

Returns: 0-4095 value, the current logarithmic volume.

See also: UltraSetVolume, UltraRampVolume

4.50. UltraRecordData

Purpose: To record some data with the UltraSound from the active 'In' ports.

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

```
PASCAL:FUNCTION UltraRecordData(PC_Ptr  : POINTER;
                                Control : BYTE;
                                Size    : WORD;
                                Wait    : BOOLEAN;
                                RRepeat : BOOLEAN) : BOOLEAN;
```

Remarks: This routine will record a buffer of data from UltraSound. It can be in either 8-bit mono or stereo. In stereo, there are two bytes and the left byte is first. 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 routine will not return until the buffer has been filled. If 'repeat' is TRUE, then the DMA channel will be set up in auto-init 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 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, however your application must be careful that it does not lose samples during the transition.

Returns: C: ULTRA_OK if no problem.
DMA_BUSY if sampling DMA Channel is busy.
BAD_DMA_ADDR if autoinit buffer crosses 64k page.

PASCAL: TRUE if successful. FALSE otherwise.
If unsuccessful, check UltraErrorStr for the reason.

See also: UltraWaitRecordDMA, UltraSetRecordFrequency
UltraGoRecord

4.51. UltraRecordDMABusy

Purpose: To see if the record DMA channel is busy.

```
int UltraRecordDmaBusy(void);
```

PASCAL:FUNCTION UltraRecordDMABusy : BOOLEAN;

Remarks: This routine 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: C: 0 = Channel not Busy
= Channel Busy

PASCAL: TRUE if the channel is not available. FALSE if it is.

See also: UltraRecordData

4.52. UltraReset

Purpose: To reset the UltraSound.

```
int UltraReset(voices);
      int voices;
```

PASCAL:FUNCTION UltraReset(Voices : INTEGER) : BOOLEAN;

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

Returns: C: ULTRA_OK if no problem.
BAD_NUM_OF_VOICES if # of voices not in range.

PASCAL: TRUE if card is successfully reset.
If FALSE, check UltraErrorStr for the reason.

See also: UltraOpen, UltraClose

4.53. UltraSetBalance

Purpose: To set a voice's pan position.

```
void UltraSetBalance(voice,data);
      int voice;
      int data;
```

```
PASCAL:PROCEDURE UltraSetBalance(Voice : INTEGER;
                                 Data  : BYTE);
```

Remarks: This routine sets the voice's position between right and left speakers. A 0 will place the audio all the way to the left whereas a 15 will put the sound all the way to the right. Values that are out of range will move the balance to the nearest extreme.

4.54. UltraSetFrequency

Purpose: To set a voice's playback frequency.

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

```
PASCAL:PROCEDURE UltraSetFrequency(Voice     : INTEGER;
                                   Speed_Khz : LONGINT);
```

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

4.55. UltraSetLoopMode

Purpose: To set a voice's loop mode.

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

```
PASCAL:PROCEDURE UltraSetLoopMode(Voice : INTEGER;
                  VMode : BYTE);
```

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

4.56. UltraSetRecordFrequency

Purpose: To set the recording rate.

```
void UltraSetRecordFrequency(rate);
     unsigned long rate;
```

```
PASCAL:PROCEDURE UltraSetRecordFrequency(Rate : LONGINT);
```

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

4.57. UltraSetVoice

Purpose: To set a voice to an absolute position in DRAM.

```
void UltraSetVoice(voice,location);
     int voice;
     unsigned long location;
```

```
PASCAL:PROCEDURE UltraSetVoice(Voice    : INTEGER;
                 Location : LONGINT);
```

Remarks: This routine 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 contains a significant value.

4.58. UltraSetVoiceEnd

Purpose: To set a voice's end position.

```
void UltraSetVoiceEnd(voice,end);
      int voice;
        unsigned long end;
```

```
PASCAL:PROCEDURE UltraSetVoiceEnd(Voice    : INTEGER;
                                  VEnd     : LONGINT);
```

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

See also: UltraSetLoopMode

4.59. UltraSetVolume

Purpose: To set a voice's current logarithmic volume.

```
void UltraSetVolume(voice,volume);
      int voice;
        unsigned int volume;
```

```
PASCAL:PROCEDURE UltraSetVolume(Voice  : INTEGER;
                                Volume : WORD);
```

Remarks: This routine sets the volume of the voice to a specific logarithmic value. The range is from 0 to 4095. Use UltraSetLinearVolume to do linear volumes.

See also: UltraSetLinearVolume

4.60. UltraSizeDRAM

Purpose: To Find the amount of DRAM available on the UltraSound.

```
int UltraSizeDram(void);
```

PASCAL:FUNCTION UltraSizeDRAM : INTEGER;

Note: This routine will always return the same value, which is the amount of DRAM that has been installed on the card. Use UltraMemAvail to see how much DRAM is still free.

Remarks: For C users, this routine 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 kilobytes found on the UltraSound.

See also: UltraMemInit

4.61. UltraStartTimer

Purpose: To set up and start either Timer 1 or Timer 2.

```
void UltraStartTimer(timer,duration);
    int timer;
      unsigned char duration;
```

PASCAL:
```
PROCEDURE UltraStartTimer(Timer    : INTEGER;
                          Duration : BYTE);
```

Remarks: This routine can be used to start up one of two available hardware timers. Timer #1 is an 80 microsecond (.00008s) timer. Timer #2 is a 320 microsecond (.00032s) timer. When starting either timer, you supply the # of counts before the timer 'ticks'. When the timer 'ticks', it ALWAYS calls the callback routine defined for it. If you don't supply one, a default is used.

These timers can be used to trigger various real time events. They are used extensively in music composition programs. 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 specified timer is automatically restarted after it ticks. Your application must explicitly call UltraStopTimer to shut it off.

See also: UltraStopTimer, UltraTimerStopped
UltraTimer1Handler, UltraTimer2Handler.

Example: Giving Timer 1 a duration of 10 counts would make the timer tick every 800 microseconds (.0008s). This would mean your callback routine is called 1250 times a second.

4.62. UltraStartVoice

Purpose: To set up and start a voice playing.

```
void UltraStartVoice(voice,begin,start,end,mode);
    int voice;
      unsigned long begin;
      unsigned long start;
      unsigned long end;
      unsigned char mode;
```

```
PASCAL:PROCEDURE UltraStartVoice(Voice  : INTEGER;
                                 VBegin : LONGINT;
                                 VStart : LONGINT;
                                 VEnd   : LONGINT;
                                 VMode  : BYTE);
```

Remarks: This routine will set up and start up a voice. The voice will begin playback at the 'begin' and continue to 'end'. If looping is enabled for this voice, then it will then jump to 'start' and then continue to 'end'. The method of looping is determined by 'mode'. See Appendix C for the definition of these bits.

See also: UltraSetVoiceEnd, UltraSetLoopMode

4.63. UltraStopTimer

Purpose: To stop a timer which has been previously activated.

```
void UltraStopTimer(timer);
      int timer;
```

PASCAL:PROCEDURE UltraStopTimer(Timer : INTEGER);

Remarks: If you have started a timer with UltraStartTimer, you will probably want to shut it off before shutting down your application. If you forget, UltraClose automatically shuts it down.

See also: UltraStartTimer

4.64. UltraStopVoice

Purpose: To stop a voice which is playing.

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

PASCAL:PROCEDURE UltraStopVoice(Voice : INTEGER);

Remarks: This routine will stop a voice which is playing at it's current position. Don't forget that if a voice is left at an unknown position, the data that is at that position will still be summed into the final output.

See also: UltraStartVoice, UltraSetVoiceEnd

4.65. UltraStopVolume

Purpose: To stop a currently active volume ramp.

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

PASCAL:PROCEDURE UltraStopVolume(Voice : INTEGER);

Remarks: You can use this routine if you wish to stop an volume ramp after a specified period of time, or you wish to make sure that a volume ramp has stopped.

See also: UltraRampVolume, UltraRampLinearVolume

4.66. UltraTimerStopped

Purpose: To check to see if a timer is running.

```
int UltraTimerStopped(timer);
      int timer;
```

PASCAL:FUNCTION UltraTimerStopped(Timer : INTEGER) : BOOLEAN;

Remarks: This routine can be used to see if a timer is currently being used.

Returns: FALSE if still running, TRUE if stopped.

See also: UltraStartTimer, UltraStopTimer

4.67. UltraTrimJoystick

Purpose: To set the trim voltage on the joystick port.

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

PASCAL:PROCEDURE UltraTrimJoystick(JoyVal : BYTE);

Remarks: This routine 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 which is included with all UltraSound stock software is used to set this value.

4.68. UltraUpload

Purpose: To transfer the contents of a block of DRAM to PC RAM.

```
int UltraUpload(dataptr,control,dram_loc,len,wait);
      void *dataptr;
        unsigned char control;
        unsigned long dram_loc;
        unsigned int len;
        int wait;
```

```
PASCAL:FUNCTION UltraUpload(DataPtr  : POINTER;
                            Control  : BYTE;
                            DRAM_Loc : LONGINT;
                            Len      : WORD;
                            Wait     : BOOLEAN) : BOOLEAN;
```

Remarks: This routine 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: C: ULTRA_OK if no problem.
DMA_BUSY if DMA Channel is busy.

PASCAL: TRUE if transfer was successful. FALSE otherwise.
If unsuccessful, check UltraErrorStr for the reason.

See also: UltraDownload, UltraDRAMDMAWait

4.69. UltraVectorVolume

Purpose: To ramp a volume from it's current position to a new value.

```
void UltraVectorVolume(voice,end,rate,mode);
      int voice;
        unsigned int end;
        unsigned char rate;
        unsigned char mode;
```

```
PASCAL:PROCEDURE UltraVectorVolume(Voice : INTEGER;
                                   VEnd  : WORD;
                                   VRate : BYTE;
                                   VMode : BYTE);
```

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

See also: UltraRampVolume, UltraStopVolume

4.70. UltraVersion

Purpose: To return the version of the SDK code being used.

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

PASCAL:PROCEDURE UltraVersion(VAR Major : BYTE;
VAR Minor : BYTE);

Remarks: This routine can be useful to track the version of the SDK that was used in compiling your source code.

Returns: Major (1 digit) and minor (2 digits) version of the unit.

See also: UltraVersionStr

4.71. UltraVersionStr

Purpose: To return the version of the SDK code being used in a string.

This function is not available.

PASCAL:FUNCTION UltraVersionStr : String;

Remarks: This routine can be useful to track the version of the SDK that was used in compiling your source code.

Returns: Major and minor version of the unit, in string format.

See also: UltraVersion

4.72. UltraVoiceStopped

Purpose: To return whether a voice is playing.

```
int UltraVoiceStopped(voice);
      int voice;
```

PASCAL:FUNCTION UltraVoiceStopped(Voice : INTEGER) : BOOLEAN;

Remarks: This routine can be used to see if a sample has finished playing .

Returns: TRUE if the voice is stopped, FALSE otherwise.

4.73. UltraVolumeStopped

Purpose: To determine if a volume ramp is running for a particular voice.

```
int UltraVolumeStopped(voice);
      int voice;
```

PASCAL:FUNCTION UltraVolumeStopped(Voice : INTEGER) : BOOLEAN;

Remarks: This routine is used to determine the current state of the volume of the voice. It can be used to see if a volume ramp is still running.

Returns: TRUE if no volume ramp is running, FALSE otherwise.

See also: UltraRampVolume, UltraRampLinearVolume

4.74. UltraWaitDRAMDMA

Purpose: To wait for a DRAM DMA transfer to complete.

```
void UltraWaitDramDMA(void);
```

PASCAL:PROCEDURE UltraWaitDRAMDMA;

Remarks: If a DMA transfer to/from DRAM is started but told not to wait for it to complete, this routine can be used to wait until it has completed.

See also: UltraDownload, UltraUpload

4.75. UltraWaitRecordDMA

Purpose: To wait for a recording DMA transfer to complete.

```
void UltraWaitRecordDMA(void);
```

PASCAL:PROCEDURE UltraWaitRecordDMA;

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

See also: UltraRecordData

4.76. UltraAllocVoice

Purpose: To allocate a new voice not currently being used.

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

PASCAL:FUNCTION UltraAllocVoice(VAR Voice_Num : INTEGER) : BOOLEAN;

Remarks: This routine 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 routine will only allocate voices up to the # of active voices specified in the UltraOpen function.

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

PASCAL: TRUE if there was no problem allocating the voice.
If FALSE, check UltraErrorStr for the reason.

See also: UltraClearVoices, UltraFreeVoices

4.77. UltraClearVoices

Purpose: To reset all voices to an un-allocated state.

```
void UltraClearVoices(void);
```

PASCAL:PROCEDURE UltraClearVoices;

Remarks: This routine will deallocate all previously allocated voices. It would be advisable to call this before using either UltraAllocVoice or UltraFreeVoice.

See also: UltraAllocVoice, UltraFreeVoice.

4.78. UltraFreeVoice

Purpose: To free up an allocated voice.

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

PASCAL: PROCEDURE UltraFreeVoice(Voice_Num : INTEGER);

Remarks: This routine 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.

See also: UltraClearVoices, UltraAllocVoice.

4.79. UltraVoiceOff

Purpose: To allow flexibility in stopping a voice.

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

```
PASCAL:PROCEDURE UltraVoiceOff(Voice : INTEGER;
                               VEnd  : BOOLEAN);
```

Remarks: This routine will either stop a voice immediately or let it finish its current loop. If 'VEnd' is FALSE then it will stop abruptly, otherwise it will finish the current loop. UltraReadVoice could be called afterwards if you need to know where the loop finished.

If used with UltraSetVoiceEnd, you could implement 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 routine with VEnd set to TRUE.

See also: UltraReadVoice, UltraSetVoiceEnd

4.80. UltraVoiceOn

Purpose: To turn a voice on at a given frequency.

```
void UltraVoiceOn(voice,begin,s_loop,e_loop,control,freq);
    int voice;
    unsigned long begin;
    unsigned long s_loop;
    unsigned long e_loop;
    unsigned char control;
    unsigned long freq;
```

```
PASCAL:PROCEDURE UltraVoiceOn(Voice      : INTEGER;
                              VBegin     : LONGINT;
                              Start_Loop : LONGINT;
                              End_Loop   : LONGINT;
                              Control    : BYTE;
                              Freq       : LONGINT);
```

Remarks: This routine just sets the frequency with UltraSetFrequency and then calls UltraStartVoice.

See also: UltraSetFrequency, UltraStartVoice

4.81. UltraSetLinearVolume

Purpose: To set a voice's volume to a linearized value.

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

PASCAL:PROCEDURE UltraSetLinearVolume(Voice : INTEGER;
Index : INTEGER);

Remarks: This routine indexes into a table to translate a linear volume (0-511) to a logarithmic one (0-4095), and then calls UltraSetVolume.

See also: UltraSetVolume

4.82. UltraReadLinearVolume

Purpose: To read a voice's volume as a linear value.

This function is not available.

PASCAL:FUNCTION UltraReadLinearVolume(Voice : INTEGER) : INTEGER;

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

Returns: Linear Volume Value (0-511).

See also: UltraSetVolume, UltraSetLinearVolume

4.83. UltraRampLinearVolume

Purpose: To ramp a voice's volume between linear volume values.

```
void UltraRampLinearVolume(voice,start,end,msecs,mode);
     int voice;
```

```
unsigned int start;
unsigned int end;
unsigned long msecs;
unsigned char mode;
```

```
PASCAL:PROCEDURE UltraRampLinearVolume(Voice     : INTEGER;
                                       Start_Idx : WORD;
                                       End_Idx   : WORD;
                                       Msecs     : LONGINT;
                                       VMode     : BYTE);
```

Remarks: This routine is used to ramp between linear volume settings. It uses the same method as UltraSetLinearVolume to determine the actual volume settings to use, and then calls UltraRampVolume.

See also: UltraRampVolume, UltraSetLinearVolume

4.84. UltraVectorLinearVolume

Purpose: To ramp a linear volume from it's current position to a known end point.

```
void UltraVectorLinearVolume(voice,end,rate,mode)
     int voice;
       unsigned int end;
       unsigned char rate;
       unsigned char mode;
```

```
PASCAL:PROCEDURE UltraVectorLinearVolume(Voice   : INTEGER;
                                         End_Idx : WORD;
                                         VRate   : BYTE;
                                         VMode   : BYTE);
```

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

See also: UltraSetLinearVolume, UltraReadLinearVolume

5. Chapter 5 - Mixer

5.1. Introduction

Revision 3.7 and above UltraSounds have had a new mixer added to them. This document will not attempt to describe how the internals of the mixer works. If you want more detailed information that is provided by this document or the example code in mixer.c, you can get a data sheet from ICS or an ICS data book. The part number is ICS-2101. You can contact ICS directly at:

Integrated Circuit Systems, Inc
2435 Boulevard of the Generals
P.O. Box 968
Valley Forge, Pa. 19482-0968
Phone # (215)-630-5300
or
Phone # (215)-630-5399

5.2. Block Diagram

µ §

5.3. Notes

To determine if one of these mixers is present, you MUST look at the revision of the board. This can be determined by reading port location 7X6. If the value read back is FF, then no revision ID is present and means that the version is pre-3.4 and therefore no mixer is present. If the revision is between 5 and 9, then the version number of the base UltraSound is 3.7 or above. All those versions will have this mixer. If the version is 10 or above, then it is an UltraMax. This mixer is NOT used on an UltraMax. The codec on the UltraMax has a mixer built into it. See chapter 6 for more information on that.

If your software determines that the board revision is 5, then you must do some special stuff. That specific revision of the UltraSound had a couple of the right\left input lines reversed. The ICS-2101 has the ability to flip them around inside the chip to correct for this problem. Subsequent versions of the board corrected this. The lines that were flipped were the synth input to the mixer and the master output. Look at the mixer.c code in the SDK sources to see and example of how to do this.

Any application that uses the mixer MUST reset back to its transparent values when it exits so the any application that isn't aware of the mixer will operate properly. The only exception to this may be some app that wants to adjust the volumes outside any running apps (TSR). Just be aware that any other programs that use the mixer will probably overwrite any values you may have set up. Since the mixer is a write-only device, that app would have no idea that its values were changed.

5.4. Mixer Functions

5.5. UltraMixProbe

Purpose: To see whether or not mixer could be on the board.

```
int UltraMixProbe(base_port);
      unsigned int base_port;
```

Remarks: This routine will look at the revision ID of the board to help determine the type of board and what components are on it. A revision between 5 and 9 indicates a mixer should be present.

Returns: C: 5 thru 9 means a mixer should be present
0 means no mixer is on this board

5.6. UltraMixAttn

Purpose: To set a channels attenuation value

```
void UltraMixAttn(channel,left_right,value,revision);
    int channel;
      int left_right;
      int value;
      int revision;
```

Remarks: This routine will set a particular channels right or left attenuation value to the proper setting. The channel values are:

Mixer channels used on GUS
Channel #4 is NOT used

```
#define MIX_MIKE_IN     MIX_CHAN_0
#define MIX_LINE_IN     MIX_CHAN_1
#define MIX_CD_IN       MIX_CHAN_2
#define MIX_GF1_OUT     MIX_CHAN_3
#define MIX_MASTER      MIX_CHAN_5
```

These are the defines for left_right flag.

```
#define MIX_LEFT 0
#define MIX_RIGHT 1
```

The value to set the attenuation to is 0-127. 127 is no attenuation and 0 is maximum attenuation (-xx db).

The revision parameter is necessary to enable it to flip the channels on a revision 5 board. See notes above.

Returns: C: 5 thru 9 means a mixer should be present
0 means no mixer is on this board

5.7. UltraMixXparent

Purpose: To set the mixer back to nominal values

```
void UltraMixXparent(revision);
    int revision;
```

Remarks: This routine will reset the mixer's values back to nominal values that will allow subsequent apps to be heard without needing to know that the mixer is present. It is highly recommended that all apps call this function when they terminate. A lot of apps that were written for the base UltraSound do not know about the mixer and their performance may be affected by not putting the mixer back to a transparent state.

The revision is needed to make sure the right->left flip is done on revision 5 boards.

6. Chapter 6 - 16 Bit Codec Functions

6.1. Introduction

Both the 16-bit daughter card and the UltraMax use the same chip to perform the extra recording and mixing functions. It is built by Crystal Semiconductors CS4231 Audio Codec. This section will describe how it fits into each of those boards. It will NOT describe much about how the internals of the CS4231 work. To get a data sheet on the part, contact Crystal Semiconductor at:

Crystal Semiconductor Corporation
P.O. Box 17847
Austin, TX 78760

Phone # (512) 445-7222
Fax # (512) 445-7581

Thsi SDK is not meant to be a complete toolkit for programming the CS4231. The CS4231 is a very versitile part that can be used in many different ways. This SDK give some simple simple example routines that can be very useful to get the part running. It is quite likely that you may find it necessary to modify or re-write some of the SDK code for use in your app. Please don't be afraid of doing just that.

6.2. Features:

- 8 or 16 bit playback and recording
    - Up to 48kHz
    - 2 DMA channels for simultaneous playback and record
    - 8 or 16 bit DMA channel selection (software selectable)
    - Multiple data formats
        - Linear PCM data
        - u-law compression (2-1)
        - a-law compression (2-1)
        - ADPCM (4-1)
    - 10 microsecond granularity timer
    - Extensive mixer
        - Line in level adjust
        - Line out level adjust
        - AUX1 and AUX2 input level adjust
        - Mic input level adjust

6.3. Block Diagram

µ §

Notes:

The L/M blocks are level adjustments and mutes. Some can perform gain and attenuation. Some can do gain only. Refer to the CS4231 data sheet to see the use and range of each of these blocks.

The microphone is mono only. The left side is connected to both right and left. This allows the same the recording to be the same whether the recording is being done thru MONO IN or MIC IN. MONO IN was connected to the MIC so that some user selectable gain could be used.

This block diagram is the same for both the UltraMax and the 16-bit daughter card.

Due to the need for backward compatibility, the mute settings for the base Ultrasound still apply. If the either codec mutes or the mute switches on the base Ultrasound are set, the corresponding input WILL be muted. To use only the codec mixer, use UltraEnableLineIn, UltraEnableMicIn, and UltraEnableOutput to ensure that the inputs are not being muted before they reach the CS4231.

6.4. Daughter card Specific Info

The daughter card has its own jumper selectable DMA channel and IRQ. These MUST be different than the software programmable ones on the base UltraSound. If not, conflicts will occur and will result in unpredictable behavior. Only 1,2 and 3 are valid DMA channels. No 16 bit channels are available. Since only 1 DMA channel is provided, simultaneous playback and record thru the codec is not possible. The base address for the codec on a daughter card can be at one of 4 locations (530,604,E80 and F40)

6.5. UltraMax Codec Specific Info

Even though the same part is used for both the 16-bit daughter card and the Max, the Max has quite a bit more functionality. First of all, it can use a 16-bit DMA channel. It also has 2 DMA channels available so that it can do simultaneous playback and record. The Windows drivers demonstrate this feature.

To support these added features, some extra hardware was added to an UltraMax. Since the CS4231 is an 8-bit DMA device, some hardware buffers were added that takes the output of the CS4231 and sends it out a 16-bit DMA channel. These buffers essentially collect 2 8-bit DMA transfers and then does 1 16-bit transfer. This is why there are 2 control bits in register 3X6. The UltraMax hardware has to know if the destination DMA channels is 8 or 16 so it knows to do this buffering.

The Max shares its DMA channels and IRQ with the base UltraSound. This limits the amount of resources that your Max will need from your system. The CS4231 will generate irqs on the same irq as the GF1. Your software will need to look at the codec (along with the GF1) when that particular level irq is generated to see if it need to be serviced.

6.6. UltraMax DMA Channel block diagram

µ §

Note that the DMA channels appear flipped between the GF1 and the codec. This is to allow us to DMA to dram to load patches while the codec is busy playing back .WAV data. We can also record the output of the GF1 (MIDI) by making sure that the patches have been loaded prior to starting the record. Since those two functions share the same DMA channel, you cannot be loading DRAM via DMA while you are recording with the codec. The codec is capable of combining its record channel with its play channel inside the codec. (That is what the 'C' is means inside the diagram above) However, because the Max does the 8-16 bit DMA channel conversion OUTSIDE the codec, this will only work on an 8 bit channel. If you try and combine the channels inside the codec and you are using 16-bit channels, the recording will fail.

6.7. Codec Functions

The following functions require:
codec.h
codecos.h
extern16.h
proto16.h

6.7.1. ULTRA16PROBE

Purpose: To probe for the existence of the CODEC

```
int Ultra16Probe(int, ULTRA16_CFG *);
      int gus_base;
      ULTRA16_CFG *config;
```

Remarks: This routine will probe for the existence of a CODEC. The config structure MUST be filled in with the appropriate values before calling this function. This structure is defined in extern16.h and looks like this:

```
typedef struct {
        unsigned int base_port;
        unsigned int cdrom_base;
        unsigned int play_dma;
        unsigned int rec_dma;
        unsigned int irq_num;
        unsigned int type;
} ULTRA16_CFG;
```

The codec base port MUST be a valid one. They are defined as:
Daughter card: 530, 604, E80 or F40
UltraMAX: 3XC, where X is 0 - F

The cdrom_base is not used and should be set to 0.

The play_dma is the channel # to be used to play data thru the CODEC. For a daughter card, the valid channels are 1,2 and 3 and MUST be different than either channel used for the base UltraSound.
For an UltraMax, this should be the same as the RECORD channel for the base UltraSound since they share these channels.

The rec_dma is the channel # to be used to record data thru the CODEC. For a daughter card, this MUST be the same as the play channel since it only has 1 8 bit channel.
For an UltraMax, this should be the same as the PLAY channel for the base UltraSound since they share these channels.

The irq_num is the interrupt # that the CODEC will use when it generates an irq. For a daughter card, the valid ones are 3,4,5,6,7 or 9 and MUST be different than either of the irqs used for the base UltraSound. For an UltraMax, this number should be the same as the GF1 interrupt since they share this

interrupt level.

The 'type' field in the config structure is defined as:
0 = 16 bit daughter card
1 = UltraMax.

The gus_base parameter is needed for an UltraMax to configure the board appropriately.

Returns: ULTRA_OK - if CODEC was found at this address
NO_ULTRA - if no CODEC detected

See Also: Ultra16Open

6.7.2. ULTRA16OPEN

Purpose: To init the CODEC to a known state

```
int Ultra16Open(int, ULTRA16_CFG *);
      int gus_base;
      ULTRA16_CFG config;
```

Remarks: This routine will probe (Ultra16Probe) for the CODEC first and then initialize it (if it found it). See Ultra16Probe for a definition of the parameters.

Returns: ULTRA_OK - if CODEC was found at this address
NO_ULTRA - if no CODEC detected

See Also: Ultra16Probe

6.7.3. ULTRA16CLOSE

Purpose: To set the CODEC back to a known state

```
int Ultra16Close(void);
```

Remarks: This routine will reset the CODEC back to a known state. It will stop and current DMA operations and reset the interrupt state. It will NOT reset the mixer back to transparent mode. Your application must do that by calling Ultra16Xparent().

Returns: Nothing

See Also: Ultra16Open, Ultra16Xparent

6.7.4. ULTRA16DISABLEIRQS

Purpose: To disable all irqs from the CODEC

```
void Ultra16DisableIrqs(void);
```

Remarks: This routine is provided to allow the application to disable all CODEC irqs if its necessary.

Returns: Nothing

See Also: Ultra16EnableIrqs

6.7.5. ULTRA16ENABLEIRQS

Purpose: To enable irqs from the CODEC

```
void Ultra16EnableIrqs(void);
```

Remarks: This routine is provided to allow the application to enable all CODEC irqs if its necessary. This is usually done sometime after Ultra16DisableIrqs().

Returns: Nothing

See Also: Ultra16DisableIrqs

6.7.6. IRQ CALLBACK HANDLERS

Purpose: To set an interrupt callback routine

```
void (*Ultra16CaptureHandler(void (*)()))();
void (*Ultra16PlaybackHandler(void (*)()))();
void (*Ultra16TimerHandler(void (*)()))();
```

Remarks: These routines define your applications callback routine for the appropriate irq. They will be called from the interrupt handler with irqs disabled, so be careful what you try to do in your callback routine.

Returns: Old callback routine in case you want to chain.

6.7.7. ULTRA16REVISION

Purpose: To read the CODEC's internal revision number

```
unsigned char Ultra16Revision(void);
```

Remarks: This routine can be used to return what the internal revision # is for the CS4231.

Returns: The internal revision #

6.7.8. ULTRA16VERSION

Purpose: Return the current version of the CODEC SDK software

```
void Ultra16Version(unsigned int *, unsigned int *);
      unsigned int *major;
      unsigned int *minor;
```

Remarks: This function can be used to get the current revision level of the CODEC SDK software. It is currently not used.

Returns: Major and minor revision #'s

6.7.9. ULTRA16XPARENT

Purpose: To put the CODEC into a transparent mode

```
void Ultra16Xparent(void);
```

Remarks: This functions should always be called by your application to reset the CODEC's internal mixer to a transparent mode. This will allow other apps that are not aware of the CODEC to work properly. All mixer levels will be either set to 0 db or muted, whichever is appropriate.

Returns: Nothing

6.7.10. CD INPUT LEVELS

Purpose: To set the CODEC's Aux2 input levels

```
unsigned char Left_CD_Input_Level(unsigned char value);
unsigned char Right_CD_Input_Level(unsigned char value);
      unsigned char value;
```

Remarks: This routine sets the level to the new value and returns the previous value. The value ranges from 0 to 31. A change of 1 count will add or subtract 1.5 db. A 0 will add 12 db gain and a value of 31 will attenuate the signal by -34.5 db. A value of 8 will give 0 db gain.

Returns: The old input level

6.7.11. GF1 (SYNTH) INPUT LEVELS

Purpose: To set the CODEC's AUX1 input levels

```
unsigned char Left_GF1_Input_Level(unsigned char value);
unsigned char Right_GF1_Input_Level(unsigned char value);
      unsigned char value;
```

Remarks: This routine sets the level to the new value and returns the previous value. The value ranges from 0 to 31. A change of 1 count will add or subtract 1.5 db. A 0 will add 12 db gain and a value of 31 will attenuate the signal by -34.5 db. A value of 8 will give 0 db gain.

Returns: The old input level

6.7.12. LINE INPUT LEVELS

Purpose: To set the CODEC's Line input levels

```
unsigned char Left_Line_Input_Level(unsigned char value);
unsigned char Right_Line_Input_Level(unsigned char value);
      unsigned char value;
```

Remarks: This routine sets the level to the new value and returns the previous value. The value ranges from 0 to 31. A change of 1 count will add or subtract 1.5 db. A 0 will add 12 db gain and a value of 31 will attenuate the signal by -34.5 db. A value of 8 will give 0 db gain.

Returns: The old input level

6.7.13. MIC INPUT LEVEL

Purpose: To set the CODEC's mono input level

```
unsigned char Mono_Input_Level(unsigned char value);
    unsigned char value;
```

Remarks: This routine sets the level to the new value and returns the previous value. The value ranges from 0 to 15. A change of 1 count will attenuate the signal by 3 db. A value of 0 will attenuate the signal 0 db. A value of 15 will give -45db. The left channel of the microphone is connected to the mono input of the CS4231. This allows us to attenuate the signal, but limits us to a mono input on the microphone. The left channel is also connected to BOTH the left and right inputs to the recording MUX. This will allow recording 2 channels (like stereo), but the same signal will be on both channels. See the block diagram to see the routing.

Returns: The old input level

6.7.14. INPUT MUTES

Purpose: To set or clear the input mute states

```
unsigned char Left_CD_Input_Mute(unsigned char on_off);
unsigned char Right_CD_Input_Mute(unsigned char on_off);
unsigned char Left_GF1_Input_Mute(unsigned char on_off);
unsigned char Right_GF1_Input_Mute(unsigned char on_off);
unsigned char Left_Line_Input_Mute(unsigned char on_off);
unsigned char Right_Line_Input_Mute(unsigned char on_off);
unsigned char Mono_Input_Mute(unsigned char on_off);
    unsigned char on_off; Value is either ON or OFF
```

Remarks: These routines will set or clear the mute state of the specified channel. When the mute value is ON, no audio is passed through. When it is off, then the appropriate level takes effect.

Returns: Old mute value

ON - If previous state was MUTE ON
OFF - If previous state was MUTE OFF

6.7.15. MIC GAIN

Purpose: To set or clear the 20 db mic input gain

```
unsigned char Left_Mic_Gain_Enable(unsigned char enable);
unsigned char Right_Mic_Gain_Enable(unsigned char);
      unsigned char enable; Value is either ON or OFF
```

Remarks: This routine will enable or disable the 20 db gain on the microphone input to the recording MUX. See block diagram.

Returns: Old enable/disable value
- ON - If previous state was Enabled
- OFF - If previous state was Disabled

6.7.16. RECORDING SOURCE

Purpose: To set up the recording source in the input MUX

```
unsigned char Left_Input_Source(unsigned char source);
unsigned char Right_Input_Source(unsigned char source);
      unsigned char source;
```

Remarks: This routine will set the MUX up to the desired input source. The choices are: line, aux 1 (gf1 input), mic, and mixed input. The defines for these can be found in codec.h.

Returns: Old MUX setting.

6.7.17. RECORDING GAIN

Purpose: To set the gain after the MUX

```
unsigned char Left_Input_Gain_Select(unsigned char gain);
unsigned char Right_Input_Gain_Select(unsigned char gain);
      unsigned char gain;
```

Remarks: This routine will set the post-MUX gain value. It ranges from 0 to 15. A change of 1 count will add 1.5 db gain. A value of 0 gives 0 db gain, and a value of 15 gives a +22.5 gain. This gain block is immediately after the MUX. See block diagram.

Returns: The old gain setting.

6.7.18. OUTPUT LEVELS

Purpose: To set the attenuation on playback

```
unsigned char Left_Output_Attn_Select(unsigned char value);
unsigned char Right_Output_Attn_Select(unsigned char value);
      unsigned char value;
```

Remarks: These routines will set the playback attenuation value. It

ranges from 0 to 63. 0 is 0 db attenuation and 63 is -94.5 db attenuation. A change of 1 count is a 1.5 db attenuation change.

Returns: Old attenuation value

6.7.19. OUTPUT MUTES

Purpose: To set or clear the output mutes

```
unsigned char Left_Output_Mute(unsigned char);
unsigned char Right_Output_Mute(unsigned char);
unsigned char Mono_Output_Mute(unsigned char);
    unsigned char on_off; Value is either ON or OFF
```

Remarks: This routine will set or clear the current state of the output mutes.

Returns: Old mute value

ON - If previous state was MUTE ON
OFF - If previous state was MUTE OFF

6.7.20. ULTRA16SETFREQ

Purpose: To set the record and playback frequency

```
unsigned int Ultra16SetFreq(unsigned int);
      unsigned int hertz;
```

Remarks: This routine will set the operating frequency of the CODEC.
This frequency will be for BOTH playback and record. The CODEC
is capable of 14 possible frequencies. If you pass it one that it
is not capable of, the code will pick the next highest one that
it can do.

```
Possible frequencies
====================
     5.51  kHz
     6.62  kHz
     8.00  kHz
     9.60  kHz
    11.025 kHz
    16.00  kHz
    18.90  kHz
    22.05  kHz
    27.42  kHz
    32.00  kHz
    33.075 kHz
    37.80  kHz
    44.10  kHz
    48.00  kHz
```

Returns: Previous frequency setting

6.7.21. ULTRA16GOPLAY

Purpose: To start up a 'primed' playback

```
int Ultra16GoPlay(unsigned char control);
      unsigned char control;          (Currently unused)
```

Remarks: This routine will enable the CODEC to start transferring
data on the DMA channel.

Returns: ULTRA_OK - Playback started OK

6.7.22. ULTRA16PLAYDATA

Purpose: To begin playing data thru CODEC

```
int Ultra16PlayData(void far *pc_ptr, unsigned char control,
                    unsigned int size, int wait, int repeat);
    void far *pc_ptr;          // pointer to PC Buffer of PCM data
    unsigned char control;     // unused, set to 0
    unsigned int size;         // # of bytes to xfer
    int wait;                  // TRUE if it shouldn't return till done
    int repeat;                // TRUE if DMA should loop (autoinit)
```

Remarks: This function will start up a playback of data from the buffer provided. An irq will be generated to the playback callback function if irqs were enabled on the CODEC.

Returns: ULTRA_OK - Play started OK
DMA_BUSY - DMA channel is already doing something
DMA_BAD_ADDR - Attempt to repeat on a buffer that crosses page

See Also: Ultra16EnableIrqs

6.7.23. ULTRA16PLAYFORMAT

Purpose: To set the data type of playback

```
unsigned int Ultra16PlayFormat(int stereo,
                               int sixteen_bit, int compress);
    int stereo;          // TRUE if mode is stereo
    int sixteen_bit;     // TRUE if data is 16-bit
    int compress;        // compressed data type
```

Remarks: This routine will program the CODEC with a new playback format and do the calibration (if necessary).

These are the defines for the compression types:

```
#define COMPRESS_NONE     0
#define COMPRESS_ADPCM    1
#define COMPRESS_ULAW     2
#define COMPRESS_ALAW     3
```

Returns: Old format bits

6.7.24. ULTRA16PROGPLAYCNT

Purpose: To set the # of samples before generating IRQ

```
void Ultra16ProgPlayCnt(unsigned int size);
    unsigned int size;
```

Remarks: This routine will set the CODEC's playback sample count register. Note that this is passed the # of bytes to send and converts it to the number of samples based on the current format. It is also possible to program a different size than the size of the DMA buffer. The CODEC generates irqs based on its count registers rather than the PC DMA controller count registers.

Returns: Nothing

See Also: Ultra16PlayFormat, Ultra16EnableIrqs

6.7.25. ULTRA16READPLAYPOSITION

Purpose: To read DMA count of xferred bytes

```
unsigned int Ultra16ReadPlayPosition(void);
```

Remarks: This routine will return the # of bytes transferred on the playback DMA channel.

Returns: The # of bytes transferred so far. If the DMA channel is in autoinit mode, then this is the # of bytes from the BEGINNING of the data buffer.

6.7.26. ULTRA16STARTPLAYDMA

Purpose: To let CODEC start playing

```
void Ultra16StartPlayDma(unsigned char);
      unsigned char control;        // Currently unused. Pass it a 0
```

Remarks: This routine will turn on the playback section of the CODEC. If the PC's DMA controller was set up to run, playback will start immediately.

Returns: Nothing

6.7.27. ULTRA16PLAYDMABUSY

Purpose: To see if play DMA channel is busy

```
int Ultra16PlayDmaBusy(void);
```

Remarks: This routine will tell you whether that particular DMA channel busy. It is possible that this channel may be busy doing something besides playback. Since the DMA channels may combined at various stages, it is necessary to check to see if the actual channel is busy, not just if a CODEC playback is in progress.

Returns: 0 - if not busy

DMA_PENDING - if channel is busy doing something

6.7.28. ULTRA16WAITPLAYDMA

Purpose: To wait till play DMA is done

void Ultra16WaitPlayDma(void);

Remarks: This routine will wait until the entire transfer is complete. Note that if the channel is in autoinit mode, it will NEVER finish.

Returns: Nothing

6.7.29. ULTRA16STOPPLAYDMA

Purpose: To shut down play DMA (PC and CODEC)

```
void Ultra16StopPlayDma(void);
```

Remarks: This routine will shut down both the CODEC and the PC's DMA controller. It should be called when ever you finish playing something or when exiting your app.

Returns: Nothing

6.7.30. START_PLAY

Purpose: Same as Ultra16PlayData

```
void Start_Play(void far *, unsigned int);
      void far *pc_ptr;     // pointer to data in PC memory
      unsigned int size;    // size (in bytes) of buffer
```

Remarks: This routine is equivalent to using this:

```
Ultra16PlayData(pc_ptr,0,size,FALSE,TRUE);
```

Returns: Nothing

6.7.31. ULTRA16READRECORDPOSITION

Purpose: To read # of bytes xferred on record channel

```
unsigned int Ultra16ReadRecordPosition(void);
```

Remarks: This routine will return the # of bytes transferred on the playback DMA channel.

Returns: The # of bytes transferred so far. If the DMA channel is in autoinit mode, then this is the # of bytes from the BEGINNING of the data buffer.

6.7.32. ULTRA16PROGRECCNT

Purpose: To set number of samples to record in CODEC

```
void Ultra16ProgRecCnt(unsigned int size);
    unsigned int size;    // # of bytes to record
```

Remarks: This routine will set the CODEC's record sample count register.
Note that this is passed the # of bytes to record and converts it to the number of samples based on the current format. It is also possible to program a different size than the size of the DMA buffer. The CODEC generates irqs based on its count registers rather than the PC DMA controller count registers.

Returns: Nothing

6.7.33. ULTRA16RECFORMAT

Purpose: To set the data type and frequency for recording

```
unsigned int Ultra16RecFormat(int stereo,
                              int sixteen_bit, int compress);
    int stereo;               // TRUE if mode is stereo
    int sixteen_bit;     // TRUE if data is 16-bit
    int compress;             // compressed data type
```

Remarks: This routine will program the CODEC with a new recording format and do the calibration (if necessary).

These are the defines for the compression types:

```
#define COMPRESS_NONE      0
#define COMPRESS_ADPCM     1
#define COMPRESS_ULAW      2
#define COMPRESS_ALAW      3
```

Returns: Old format bits

6.7.34. ULTRA16GORECORD

Purpose: To start up a 'primed' record

```
int Ultra16GoRecord(unsigned char control);
    unsigned char control;          (Currently unused)
```

Remarks: This routine will enable the CODEC to start transferring data on the DMA channel. It can be used to resume a DMA transfer that was stopped by Ultra16StopRecord().

Returns: ULTRA_OK - Record started OK

6.7.35. ULTRA16RECORDDATA

Purpose: To record a buffer of data

```
int Ultra16RecordData(void far *pc_ptr, unsigned char control,
```

```
                    unsigned int size, int wait, int repeat);
    void far *pc_ptr;          // pointer to PC Buffer of PCM data
    unsigned char control;     // unused, set to 0
    unsigned int size;         // # of bytes to xfer
    int wait;                  // TRUE if it shouldn't return till done
    int repeat;                // TRUE if DMA should loop (autoinit)
```

Remarks: This function will start up a record of data to the buffer
provided. An irq will be generated to the capture callback
function if irqs were enabled on the CODEC.

Returns: ULTRA_OK - Record started OK
DMA_BUSY - DMA channel is already doing something
DMA_BAD_ADDR - Attempt to repeat on a buffer that crosses page

See Also: Ultra16EnableIrqs

6.7.36. ULTRA16STARTRECORDDMA

Purpose: To let CODEC start recording

```
void Ultra16StartRecordDma(unsigned char);
    unsigned char control;      // Currently unused. Pass it a 0
```

Remarks: This routine will turn on the record section of the CODEC. If the PC's DMA controller was set up to run, recording will start immediately.

Returns: Nothing

6.7.37. ULTRA16RECORDDMABUSY

Purpose: To see if record channel is busy

```
int Ultra16RecordDmaBusy(void);
```

Remarks: This routine will tell you whether that particular DMA channel busy. It is possible that this channel may be busy doing something besides recording. Since the DMA channels may combined at various stages, it is necessary to check to see if the actual channel is busy, not just if a CODEC recording is in progress.

Returns: 0 - if not busy
DMA_PENDING - if channel is busy doing something

6.7.38. ULTRA16WAITRECORDDMA

Purpose: To wait till record is finished

```
void Ultra16WaitRecordDma(void);
```

Remarks: This routine will wait until the entire transfer is complete. Note that if the channel is in autoinit mode, it will NEVER finish.

Returns: Nothing

6.7.39. ULTRA16STOPRECORDDMA

Purpose: To shut down a record dma channel (PC and CODEC)

```
void Ultra16StopRecordDma(void);
```

Remarks: This routine will shut down both the CODEC and the PC's DMA controller. It should be called when ever you finish recording something or when exiting your app.

Returns: Nothing

6.7.40. START_RECORDING

Purpose: Same as Ultra16RecordData

```
void Start_Recording(void far *pc_ptr, unsigned int size);
      void far *pc_ptr;     // pointer to data in PC memory
      unsigned int size;    // size (in bytes) of buffer
```

Remarks: This routine is equivalent to using this:

```
Ultra16RecordData(pc_ptr,0,size,FALSE,TRUE);
```

Returns: Nothing

6.7.41. ULTRA16STARTTIMER

Purpose: To start up the CODEC's timer

```
void Ultra16StartTimer(void);
```

Remarks: Start up the CODEC's 10 microsecond timer.

Returns: Nothing

6.7.42. ULTRA16STOPTIMER

Purpose: To stop the CODEC's timer

```
void Ultra16StopTimer(void);
```

Remarks: Stop the CODEC's 10 microsecond timer.

Returns: Nothing

6.7.43. ULTRA16SETTIMER

Purpose: To set the CODEC's timer count

```
void Ultra16SetTimer(unsigned int count);
      unsigned int count; // # of 10 microsecond slices before it ticks
```

Remarks: This timer has a 10 microsecond granularity. Since the count register is a 16 bit counter, this gives a maximum period of 655.35 ms. This timer is restarted automatically when it counts down to 0. If irqs have been enabled, it will call the timer callback routine. The timer count is reset whenever a new count is written to it.

Returns: Nothing

7. Chapter 7 - Utility Functions

7.1. Introduction

This section provides information to the various utility functions.

7.2. UltraGetCfg

Purpose: To get ULTRASND environment string's contents

```
int UltraGetCfg(ULTRA_CFG *config);
      ULTRA_CFG *config;
```

Remarks: Read the environment string ULTRASND and return the
PORT, DMA and IRQ parameters. This routine will init the
structure to the defaults and will overwrite them with the
values found in the environment. If no environment strings
was found, it will return the default values in the structure
but will return FALSE rather than TRUE.

Returns: TRUE if structure set by environment string.
FALSE if structure contains defaults (no environment string
found)

7.3. GetUltra16Cfg

Purpose: To get ULTRA16 environment string values

```
int GetUltra16Cfg(ULTRA_CFG *config, ULTRA16_CFG *config16);
      ULTRA_CFG *config;
      ULTRA16_CFG *config16;
```

Remarks: Read the environment string ULTRA16 and return the
PORT, DMA and IRQ parameters. This routine will init the
structure to the defaults and will overwrite them with the
values found in the environment. If no environment string
was found, it will return the default values in the structure

and will return FALSE rather than TRUE.

Returns: TRUE if structure set by environment string.
FALSE if structure contains defaults (no environment string found)

7.4. MallocAlignedBuff

Purpose: TO get a page aligned buffer for autoinit DMA

```
void far * MallocAlignedBuff(unsigned int size);
    unsigned int size;
```

Remarks: This routine will return a buffer that sits entirely within one DMA page of PC memory. This is very useful for getting buffers that can be used as autoinit DMA buffers. Since the DMA controller cannot loop on memory that crosses a 64K page, a buffer MUST be completely within a 64K page if it is to be used in this mode.

NOTE: This routine can only be called once. Subsequent calls will not necessarily return an aligned buffer.

Returns: NULL if it couldn't get the memory. A valid pointer if it got the memory OK.

8. Appendix A - Error Codes

======

======

Main routines: (these are defined in ULTRAERR.H)

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

=========
PASCAL:
=========

Please see the discussion of the PASCAL error handling techniques at the end of section 3.14.1.

9. Appendix B - Volume Control

Here are the volume ramp control bit definitions:

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- Reserved
  |   |   |   |   |   |   +-------- Reserved
  |   |   |   |   |   +------------ Reserved
  |   |   |   |   +---------------- loop enable (0=no loop, 1=loop)
  |   |   |   +-------------------- bi-direction. loop (1=enable)
  |   |   +------------------------ Enable volume ramp IRQ
  |   +---------------------------- Reserved
  +-------------------------------- Reserved
```

The bits that should be set by the application to get a particular type of volume ramp. The completed value should be supplied to UltraRampVolume and UltraRampLinearVolume. If volume interrupts are enabled, make sure that you have set up a volume interrupt handler (see UltraVolumeHandler). This can be used to create your own multi-point volume envelopes.

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

10. Appendix C - Voice Control

Here are the voice control bit definitions:

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- Rollover when hit ending addr
  |   |   |   |   |   |   +-------- Reserved
  |   |   |   |   |   +------------ data type (0=8 bit 1=16 bit)
  |   |   |   |   +---------------- loop enable (0=no loop, 1=loop)
  |   |   |   +-------------------- bi-direction. loop (1=enable)
  |   |   +------------------------ Enable wavetable IRQ
  |   +---------------------------- Direction (0=inc, 1=dec)
  +-------------------------------- 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 uni-directional or bi-directional mode. If you have asked that a particular voice generate a wavetable interrupt when it hits the end of the data (or loop point, if looping is specified), be sure you have specified a wavetable interrupt hander (UltraWaveHander). The mode bits would be constructed and passed to UltraStartVoice and UltraSetLoopMode. The rollover bit is used to tell the software to program the hardware: generate an interrupt when the voice hits the end address, but don't stop playing the voice (or loop if looping is enabled). This allows a voice to continue to play but gives the software a 'marker point' in the playback. This can be very powerful to allow a seamless playback. The hardware bit for turning rollover on is actually in the voice's volume control register (there were no spare bits in the voice control register). One of the bits is shared to tell the tool kit software to enable the rollover in the volume control register.

11. Appendix D - DMA Control

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

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- Reserved
  |   |   |   |   |   |   +-------- 0=write,1=Read
  |   |   |   |   |   +------------ Reserved
  |   |   |   |   +---------------- Reserved
  |   |   |   +-------------------- Reserved
  |   |   +------------------------ Reserved
  |   +---------------------------- Data size (0=8 bit, 1=16 bit)
  +-------------------------------- (1=convert to 2's comp.)
```

Note: The UltraSound DRAM location MUST be on a 32 byte boundary. UltraMemAlloc enforces this stipulation.

You define the byte to be passed to UltraUpload and UltraDownload using these bits. To DMA the data out of DRAM, use UltraUpload. To send data to the DRAM, use UltraDownload. Be sure to specify the sample size and whether or not the data is in two's 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 when specifying the mode when you call UltraDownload - 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.

12. Appendix E - Recording Control

Here are the recording control bit definitions:

```
=================================
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
=================================
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   +---- Reserved
  |   |   |   |   |   |   +-------- mode (0=mono,1=stereo)
  |   |   |   |   |   +------------ Reserved
  |   |   |   |   +---------------- Reserved
  |   |   |   +-------------------- Reserved
  |   |   +------------------------ Reserved
  |   +---------------------------- Reserved
  +-------------------------------- (1=Convert to 2's comp.)
```

13. Appendix F - Patch Files

This appendix contains information about the patch file format. It assumes the reader is familiar with wave table synthesis, music, sound,
and the GF1 ASIC chip that is the heart of the UltraSound audio card.

The UltraSound patch file is a collection of data structures and sound data in the following format:

```
patch header (# instruments)
    instrument header 1 (#layers)
        layer 1 header (# waves)
                wave 1 header
                wave 1 data
            wave 2 header
            wave 2 data
            ...
            wave n header
            wave n data
        layer 2 header (# waves)
            wave 1 header
            wave 1 data
            wave 2 header
            wave 2 data
            ...
            wave n header
            wave n data
    instrument header 2 (#layers)
etc.
```

All of the Advanced Gravis UltraSound patches contain one instrument at the current time. Except for the main patch header, each of the headers has a size field which can be used to seek past unwanted data. For example, if a patch has two instruments and you want to skip the first instrument, you would read the first instrument header, and then skip INSTRUMENT_SIZE bytes. What follows is the patch header definitions given in C and Pascal. After the definitions, a field-by-field breakdown of the headers is given.

======

======

```
#define ENVELOPES                       6
#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"
#define MAX_LAYERS                      4
#define INST_NAME_SIZE                          16

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;

   char            modes;

   /* bit 0 = 0=8bit 1=16bit wave data. */
   /* bit 1 = 0=Signed  1=Unsigned data. */
   /* bit 2 = 1=looping enabled. */
   /* bit 3 = 0=uni-direct loop. 1=bi-direct loop. */
   /* bit 4 = 0=loop forward. 1=loop backward. */
   /* bit 5 = 1=Turn sustaining on. (Env. pts. 3)*/
   /* bit 6 = 1=Enable envelopes */

   int             scale_frequency;
   unsigned int    scale_factor;    /* from 0 to 2048 or 0 to 2 */

   char            reserved[ PATCH_DATA_RESERVED_SIZE ];
} PATCHDATA
```

...end of the C section. Please see the next page for the PASCAL section...

=========
PASCAL:
=========

```
CONST
  ENVELOPES                  = 6;
  HEADER_SIZE                = 12;
  ID_SIZE                    = 10;
  DESC_SIZE                  = 60;
  RESERVED_SIZE              = 40;
  PATCH_HEADER_RESERVED_SIZE = 36;
  LAYER_RESERVED_SIZE        = 40;
  PATCH_DATA_RESERVED_SIZE   = 36;
  GF1_HEADER_TEXT            : ARRAY[1..12] OF CHAR = 'GF1PATCH110';

TYPE
  PATCHHEADER = RECORD
    Header        : ARRAY[1..HEADER_SIZE] OF CHAR;
    Gravis_ID     : ARRAY[1..ID_SIZE] OF CHAR;  { Id = 'ID#000002' }
    Description   : ARRAY[1..DESC_SIZE] OF CHAR;
    Instruments   : SHORTINT;
    Voices        : SHORTINT;
    Channels      : SHORTINT;
    Wave_Forms    : WORD;
    Master_Volume : WORD;
    Data_Size     : LONGINT;
    Reserved      : ARRAY[1..PATCH_HEADER_RESERVED_SIZE] OF BYTE;
   END;

  INSTRUMENTDATA = RECORD
    Instrument      : WORD;
    Instrument_Name : ARRAY[1..16] OF CHAR;
    Instrument_Size : LONGINT;
    Layers          : SHORTINT;
    Reserved        : ARRAY[1..RESERVED_SIZE] OF BYTE;
   END;

  LAYERDATA = RECORD
    Layer_Duplicate : SHORTINT;
    Layer           : SHORTINT;
    Layer_Size      : LONGINT;
    Samples         : SHORTINT;
    Reserved        : ARRAY[1..RESERVED_SIZE] OF BYTE;
   END;
```

```
PATCHDATA = RECORD
  Wave_Name       : ARRAY[1..7] OF CHAR;
  Fractions       : BYTE;
  Wave_Size       : LONGINT;
  Start_Loop      : LONGINT;
  End_Loop        : LONGINT;
  Sample_Rate     : WORD;
  Low_Frequency   : LONGINT;
  High_Frequency  : LONGINT;
  Root_Frequency  : LONGINT;
  Tune            : INTEGER;
  Balance         : BYTE;
  Envelope_Rate   : ARRAY[1..ENVELOPES] OF BYTE;
  Envelope_Offset : ARRAY[1..ENVELOPES] OF BYTE;
  Tremolo_Sweep   : BYTE;
  Tremolo_Rate    : BYTE;
  Tremolo_Depth   : BYTE;
  Vibrato_Sweep   : BYTE;
  Vibrato_Rate    : BYTE;
  Vibrato_Depth   : BYTE;

  Modes           : SHORTINT;

  { bit 0 = 0=8bit 1=16bit wave data.           }
  { bit 1 = 0=Signed 1=Unsigned data.           }
  { bit 2 = 1=looping enabled.                  }
  { bit 3 = 0=uni-direct loop. 1=bi-direct loop.}
  { bit 4 = 0=loop forward. 1=loop backward.    }
  { bit 5 = 1=Turn sustaining on. (Env. pts. 3) }
  { bit 6 = 1=Enable envelopes                  }

  Scale_Frequency : INTEGER;
  Scale_Factor    : WORD; { From 0 to 2048 (0 to 2 semitones) }
  Reserved        : ARRAY[1..PATCH_DATA_RESERVED_SIZE] OF BYTE;
 END;
```

13.1. File Header

'Header'

The header field should contain the text "GF1PATCH110." The first 8 bytes will always be GF1PATCH. The next three bytes are the version number of the patch format. As fields are added to the patch, the number will be incremented. All of the UltraSound patches are currently at version 110. The older 100 patches are obsolete and should no longer be in use.

'Description'

This description field is usually for copyright information.

'Instruments'

The number of instruments in the patch. All of the Gravis patches contain only one instrument.

'Voices'

The GF1 synth can update 14 voices at 44.1Khz. As the number of voices increases, the actual output rate of each voice drops. This field should contain the number of voices that were used when creating the patch. This field could be used by a MIDI engine to try and make the patch sound the same regardless of the number of active voices. Currently this feature is not implemented in the Gravis MIDI engine.

'Channels'

This field is unused. Only mono data can be played out a voice with the GF1.

'Wave_Forms'

The total number of waveforms in the patch. This field is used by programs which need to preallocate space for wave form headers before the patch is loaded.

'Master_Volume'

This field is currently unused.

'Data_Size'

The size of the patch data after it is loaded into GF1 dram. This number includes the space needed to align each of the waveforms on 32 byte boundaries. If you patch loader maintains linked list of waveforms in dram, and you need to use an extra 32 bytes per waveform, then you can use the wave_forms field to figure out how much memory you will need to load the patch. i.e. data_size + (32 * wave_forms).

13.2. Instrument Header

'Instrument'
    This field is currently unused.

'Instrument_Name'
    This field is currently unused.

'Instrument_Size'
    The size of the instrument. The number of bytes to skip in order to read the next instrument header.

'Layers'
    The number of layers in this instrument. Multiple layers are usually used in patches where more than one sound is required with a single MIDI event (note on). For example, a piano would could have two layers. The first would be the actual tone from the hammer hitting the strings, and the second would be the (thunk) of the key being struck and all of the mechanisms moving inside the piano. The mechanisms would probably be frequency independent, and also independent of the length of the tone.

13.3. Layer Header

'Layer_Duplicate'
    Currently unused. If the layer duplicate is non zero, then this layer should use the data from the previous layer. Only the headers will follow.

'Layer'
    The current layer number for this instrument

'Layer_Size'
    The size of this layer. Can be used to seek past this layer in the file.

'Samples'
    The number of waveforms in this layer. This field is ignored if layer_duplicate is true.

13.4. Wave Header

'Wave_Name'
    This field is currently unused

'Fractions'
    The start_loop and end_loop are the integer portions of the wavetable address. The GF1 can interpolate between sample points and therefore more resolution than just the integer address is needed. The most significant four bits are the fractional address for the start_loop, and the least significant four bits are the fractional address for the end_loop.

'Wave_Size'

The number of bytes of wave table data that follows -- not # of samples.

'Start_Loop'

The integer portion of the starting loop address relative to the beginning of the wave. This address is the relative number of bytes, and not the number of samples.

'End_Loop'

The integer portion of the ending loop address relative to the beginning of the wave. This address is the relative number of bytes, and not the number of samples.

'Sample_Rate'

This is the sample rate of the recorded data. This number is not related to the actual pitch of the recorded tone.

'Low_Frequency'

Each wave covers a specific frequency range. This is the lowest frequency that this wave can be used to play. This field is scaled be 1000 for accuracy.

'High_Frequency'

Each wave covers a specific frequency range. This is the highest frequency that this wave can be used to play. This field is scaled by 1000 for accuracy. If there is another wave adjacent to this one and its range overlaps this range, then the next waveform will always be chosen.

'Root_Frequency'

If this wave is played back at the original sample_rate, then this number should be the pitch of the original tone. This field is modified to tune the wave to a particular pitch. This field is scaled by 1000 for accuracy.

'Tune'

This field is unused. Tuning is accomplished by modifying the root frequency.

'Balance'

is 100% to the left and 15 is 100% to the right. As the balance is shifted from left to right, the total output power of both channels is constant.

'Envelope_Rate'

An array of 6 rates to implement a 6-point envelope. The first three rates can be used for attack and decay. If the sustain flag is set, than the third envelope point will be the sustain point. The last three envelope points are for the release, and an optional "echo" effect. If the last envelope point is left at an audible level, then a sampled release can heard after the last envelope point. The rate values are sent directly to the GF1 hardware, and are described in the volume ramping section.

'Envelope_Offset'

An array of 6 offsets to implement a 6-point envelope. The first three offsets can be used for attack and decay. If the sustain flag is set, than the third envelope point will be the sustain point. The last three envelope points are for the release, and an optional "echo" effect. If the last envelope point is left at an audible level, then a sampled release can heard after the last envelope point. The offset values are sent directly to the GF1 hardware, and are described in the volume ramping section.

'Tremolo_Sweep'

Not implemented. Tremolo starts automatically when the note sustains. This will be changed in future software to gradually sweep in the tremolo depth from 0 at the rate of tremolo_sweep / 45 seconds.

'Tremolo_Rate'

The rate of amplitude modulation. 0 is 0.05 Hz, and 255 is 6 Hz. A complete table is in Appendix H.

'Tremolo_Depth'

means turn tremolo off. 255 provides a 16 dB modulation. A complete table is in Appendix H..

'Vibrato_Sweep'

Gradually sweep in the vibrato depth from 0 at the rate of vibrato_sweep / 45 seconds.

'Vibrato_Rate'

The rate of frequency modulation. 0 is 0.05 Hz, and 255 is 6 Hz.

'Vibrato_Depth'

means turn vibrato off. 255 is a one octave modulation.

'Modes'

A set of bit fields describing modes and data type:

BIT 0- 16 bit data
BIT 1- unsigned data
BIT 2- looping enabled
BIT 3- bi-directional loop
BIT 4- play patch backwards. start at end address, loop backwards, and then end at beginning address.
BIT 5- sustain - enveloping stops at third envelope point. a note off will continue enveloping.
BIT 6- currently means enveloping enabled. All Gravis patches have this bit set. This field will be modified in the near future to implement a sampled release at note off instead of at the last envelope point. If this bit is on, the sample release occurs after the last envelope point. If this bit is off, the sampled release occurs at the note off.
BIT 7- fast release. The last three envelope points are ignored.

'Scale_Frequency'

Keyboard frequency scaling. Normally, MIDI note 64 plays a middle C. A 65 plays a C#. frequency scaling changes the distance in pitch of each MIDI note. The scale_frequency is the MIDI note number which is the pivot point for scaling. If scale_frequency is 64, then MIDI note 64 will sound like a C4 regardless of scale factor.

'Scale_Factor'

means normal scaling. Each MIDI note is one semitone away from its neighbor. 512 would be 1/2 semitone apart. 2048 is two semitones apart. This field can range from 0 semitones (you won't hear
it) to 2 semitones. A 512 will give you 1/2 semitone steps. a 256 will give you a 1/4 semitone step.
A 1526 will give you a 1 1/2 semitone step, etc.

14. Appendix G - Modulation Tables

14.1. RATE table for TREMOLO* and VIBRATO**

rate	frequency	rate	frequency	rate	frequency
000:	0.050 Hz	085:	2.034 Hz	170:	4.018 Hz
001:	0.073 Hz	086:	2.057 Hz	171:	4.041 Hz
002:	0.097 Hz	087:	2.081 Hz	172:	4.064 Hz
003:	0.120 Hz	088:	2.104 Hz	173:	4.088 Hz
004:	0.143 Hz	089:	2.127 Hz	174:	4.111 Hz
005:	0.167 Hz	090:	2.151 Hz	175:	4.135 Hz
006:	0.190 Hz	091:	2.174 Hz	176:	4.158 Hz
007:	0.213 Hz	092:	2.197 Hz	177:	4.181 Hz
008:	0.237 Hz	093:	2.221 Hz	178:	4.205 Hz
009:	0.260 Hz	094:	2.244 Hz	179:	4.228 Hz
010:	0.283 Hz	095:	2.267 Hz	180:	4.251 Hz
011:	0.307 Hz	096:	2.291 Hz	181:	4.275 Hz
012:	0.330 Hz	097:	2.314 Hz	182:	4.298 Hz
013:	0.353 Hz	098:	2.337 Hz	183:	4.321 Hz
014:	0.377 Hz	099:	2.361 Hz	184:	4.345 Hz
015:	0.400 Hz	100:	2.384 Hz	185:	4.368 Hz
016:	0.423 Hz	101:	2.407 Hz	186:	4.391 Hz
017:	0.447 Hz	102:	2.431 Hz	187:	4.415 Hz
018:	0.470 Hz	103:	2.454 Hz	188:	4.438 Hz
019:	0.493 Hz	104:	2.477 Hz	189:	4.461 Hz
020:	0.517 Hz	105:	2.501 Hz	190:	4.485 Hz
021:	0.540 Hz	106:	2.524 Hz	191:	4.508 Hz
022:	0.563 Hz	107:	2.547 Hz	192:	4.531 Hz
023:	0.587 Hz	108:	2.571 Hz	193:	4.555 Hz
024:	0.610 Hz	109:	2.594 Hz	194:	4.578 Hz
025:	0.633 Hz	110:	2.617 Hz	195:	4.601 Hz
026:	0.657 Hz	111:	2.641 Hz	196:	4.625 Hz
027:	0.680 Hz	112:	2.664 Hz	197:	4.648 Hz
028:	0.704 Hz	113:	2.687 Hz	198:	4.671 Hz
029:	0.727 Hz	114:	2.711 Hz	199:	4.695 Hz
030:	0.750 Hz	115:	2.734 Hz	200:	4.718 Hz
031:	0.774 Hz	116:	2.757 Hz	201:	4.741 Hz
032:	0.797 Hz	117:	2.781 Hz	202:	4.765 Hz
033:	0.820 Hz	118:	2.804 Hz	203:	4.788 Hz
034:	0.844 Hz	119:	2.827 Hz	204:	4.811 Hz
035:	0.867 Hz	120:	2.851 Hz	205:	4.835 Hz
036:	0.890 Hz	121:	2.874 Hz	206:	4.858 Hz
037:	0.914 Hz	122:	2.897 Hz	207:	4.881 Hz
038:	0.937 Hz	123:	2.921 Hz	208:	4.905 Hz
039:	0.960 Hz	124:	2.944 Hz	209:	4.928 Hz
040:	0.984 Hz	125:	2.967 Hz	210:	4.951 Hz
041:	1.007 Hz	126:	2.991 Hz	211:	4.975 Hz
042:	1.030 Hz	127:	3.014 Hz	212:	4.998 Hz
043:	1.054 Hz	128:	3.038 Hz	213:	5.021 Hz
044:	1.077 Hz	129:	3.061 Hz	214:	5.045 Hz

045:	1.100 Hz	130:	3.084 Hz	215:	5.068 Hz
046:	1.124 Hz	131:	3.108 Hz	216:	5.091 Hz
047:	1.147 Hz	132:	3.131 Hz	217:	5.115 Hz
048:	1.170 Hz	133:	3.154 Hz	218:	5.138 Hz
049:	1.194 Hz	134:	3.178 Hz	219:	5.161 Hz
050:	1.217 Hz	135:	3.201 Hz	220:	5.185 Hz

rate	frequency	rate	frequency	rate	frequency
051:	1.240 Hz	136:	3.224 Hz	221:	5.208 Hz
052:	1.264 Hz	137:	3.248 Hz	222:	5.231 Hz
053:	1.287 Hz	138:	3.271 Hz	223:	5.255 Hz
054:	1.310 Hz	139:	3.294 Hz	224:	5.278 Hz
055:	1.334 Hz	140:	3.318 Hz	225:	5.301 Hz
056:	1.357 Hz	141:	3.341 Hz	226:	5.325 Hz
057:	1.380 Hz	142:	3.364 Hz	227:	5.348 Hz
058:	1.404 Hz	143:	3.388 Hz	228:	5.372 Hz
059:	1.427 Hz	144:	3.411 Hz	229:	5.395 Hz
060:	1.450 Hz	145:	3.434 Hz	230:	5.418 Hz
061:	1.474 Hz	146:	3.458 Hz	231:	5.442 Hz
062:	1.497 Hz	147:	3.481 Hz	232:	5.465 Hz
063:	1.520 Hz	148:	3.504 Hz	233:	5.488 Hz
064:	1.544 Hz	149:	3.528 Hz	234:	5.512 Hz
065:	1.567 Hz	150:	3.551 Hz	235:	5.535 Hz
066:	1.590 Hz	151:	3.574 Hz	236:	5.558 Hz
067:	1.614 Hz	152:	3.598 Hz	237:	5.582 Hz
068:	1.637 Hz	153:	3.621 Hz	238:	5.605 Hz
069:	1.660 Hz	154:	3.644 Hz	239:	5.628 Hz
070:	1.684 Hz	155:	3.668 Hz	240:	5.652 Hz
071:	1.707 Hz	156:	3.691 Hz	241:	5.675 Hz
072:	1.730 Hz	157:	3.714 Hz	242:	5.698 Hz
073:	1.754 Hz	158:	3.738 Hz	243:	5.722 Hz
074:	1.777 Hz	159:	3.761 Hz	244:	5.745 Hz
075:	1.800 Hz	160:	3.784 Hz	245:	5.768 Hz
076:	1.824 Hz	161:	3.808 Hz	246:	5.792 Hz
077:	1.847 Hz	162:	3.831 Hz	247:	5.815 Hz
078:	1.871 Hz	163:	3.854 Hz	248:	5.838 Hz
079:	1.894 Hz	164:	3.878 Hz	249:	5.862 Hz
080:	1.917 Hz	165:	3.901 Hz	250:	5.885 Hz
081:	1.941 Hz	166:	3.924 Hz	251:	5.908 Hz
082:	1.964 Hz	167:	3.948 Hz	252:	5.932 Hz
083:	1.987 Hz	168:	3.971 Hz	253:	5.955 Hz
084:	2.011 Hz	169:	3.994 Hz	254:	5.978 Hz
				255:	6.002 Hz

* Tremolo rates for the GF1 UltraSound vary a small amount by depth. The larger the depth the more accurate the rate.

** Vibrato rates for the GF1 UltraSound lose resolution as the rate increases. In other words, you will hear a rate difference between 0 and 1, but not between 169 and 179.

14.2. DEPTH table for TREMOLO

depth	decibels	depth	decibels	depth	decibels
000:	OFF	085:	4.031 dB	170:	8.016 dB
001:	0.094 dB	086:	4.078 dB	171:	8.062 dB
002:	0.141 dB	087:	4.125 dB	172:	8.109 dB
003:	0.188 dB	088:	4.172 dB	173:	8.156 dB
004:	0.234 dB	089:	4.219 dB	174:	8.203 dB
005:	0.281 dB	090:	4.266 dB	175:	8.250 dB
006:	0.328 dB	091:	4.312 dB	176:	8.297 dB
007:	0.375 dB	092:	4.359 dB	177:	8.344 dB
008:	0.422 dB	093:	4.406 dB	178:	8.391 dB
009:	0.469 dB	094:	4.453 dB	179:	8.438 dB
010:	0.516 dB	095:	4.500 dB	180:	8.484 dB
011:	0.562 dB	096:	4.547 dB	181:	8.531 dB
012:	0.609 dB	097:	4.594 dB	182:	8.578 dB
013:	0.656 dB	098:	4.641 dB	183:	8.625 dB
014:	0.703 dB	099:	4.688 dB	184:	8.672 dB
015:	0.750 dB	100:	4.734 dB	185:	8.719 dB
016:	0.797 dB	101:	4.781 dB	186:	8.766 dB
017:	0.844 dB	102:	4.828 dB	187:	8.812 dB
018:	0.891 dB	103:	4.875 dB	188:	8.859 dB
019:	0.938 dB	104:	4.922 dB	189:	8.906 dB
020:	0.984 dB	105:	4.969 dB	190:	8.953 dB
021:	1.031 dB	106:	5.016 dB	191:	9.000 dB
022:	1.078 dB	107:	5.062 dB	192:	9.047 dB
023:	1.125 dB	108:	5.109 dB	193:	9.094 dB
024:	1.172 dB	109:	5.156 dB	194:	9.141 dB
025:	1.219 dB	110:	5.203 dB	195:	9.188 dB
026:	1.266 dB	111:	5.250 dB	196:	9.234 dB
027:	1.312 dB	112:	5.297 dB	197:	9.281 dB
028:	1.359 dB	113:	5.344 dB	198:	9.328 dB
029:	1.406 dB	114:	5.391 dB	199:	9.375 dB
030:	1.453 dB	115:	5.438 dB	200:	9.422 dB
031:	1.500 dB	116:	5.484 dB	201:	9.469 dB
032:	1.547 dB	117:	5.531 dB	202:	9.516 dB
033:	1.594 dB	118:	5.578 dB	203:	9.562 dB
034:	1.641 dB	119:	5.625 dB	204:	9.609 dB
035:	1.688 dB	120:	5.672 dB	205:	9.656 dB
036:	1.734 dB	121:	5.719 dB	206:	9.703 dB
037:	1.781 dB	122:	5.766 dB	207:	9.750 dB
038:	1.828 dB	123:	5.812 dB	208:	9.797 dB
039:	1.875 dB	124:	5.859 dB	209:	9.844 dB
040:	1.922 dB	125:	5.906 dB	210:	9.891 dB
041:	1.969 dB	126:	5.953 dB	211:	9.938 dB
042:	2.016 dB	127:	6.000 dB	212:	9.984 dB
043:	2.062 dB	128:	6.047 dB	213:	10.031 dB
044:	2.109 dB	129:	6.094 dB	214:	10.078 dB
045:	2.156 dB	130:	6.141 dB	215:	10.125 dB
046:	2.203 dB	131:	6.188 dB	216:	10.172 dB
047:	2.250 dB	132:	6.234 dB	217:	10.219 dB
048:	2.297 dB	133:	6.281 dB	218:	10.266 dB
049:	2.344 dB	134:	6.328 dB	219:	10.312 dB

050:	2.391 dB	135:	6.375 dB	220:	10.359 dB

depth	decibels	depth	decibels	depth	decibels
051:	2.438 dB	136:	6.422 dB	221:	10.406 dB
052:	2.484 dB	137:	6.469 dB	222:	10.453 dB
053:	2.531 dB	138:	6.516 dB	223:	10.500 dB
054:	2.578 dB	139:	6.562 dB	224:	10.547 dB
055:	2.625 dB	140:	6.609 dB	225:	10.594 dB
056:	2.672 dB	141:	6.656 dB	226:	10.641 dB
057:	2.719 dB	142:	6.703 dB	227:	10.688 dB
058:	2.766 dB	143:	6.750 dB	228:	10.734 dB
059:	2.812 dB	144:	6.797 dB	229:	10.781 dB
060:	2.859 dB	145:	6.844 dB	230:	10.828 dB
061:	2.906 dB	146:	6.891 dB	231:	10.875 dB
062:	2.953 dB	147:	6.938 dB	232:	10.922 dB
063:	3.000 dB	148:	6.984 dB	233:	10.969 dB
064:	3.047 dB	149:	7.031 dB	234:	11.016 dB
065:	3.094 dB	150:	7.078 dB	235:	11.062 dB
066:	3.141 dB	151:	7.125 dB	236:	11.109 dB
067:	3.188 dB	152:	7.172 dB	237:	11.156 dB
068:	3.234 dB	153:	7.219 dB	238:	11.203 dB
069:	3.281 dB	154:	7.266 dB	239:	11.250 dB
070:	3.328 dB	155:	7.312 dB	240:	11.297 dB
071:	3.375 dB	156:	7.359 dB	241:	11.344 dB
072:	3.422 dB	157:	7.406 dB	242:	11.391 dB
073:	3.469 dB	158:	7.453 dB	243:	11.438 dB
074:	3.516 dB	159:	7.500 dB	244:	11.484 dB
075:	3.562 dB	160:	7.547 dB	245:	11.531 dB
076:	3.609 dB	161:	7.594 dB	246:	11.578 dB
077:	3.656 dB	162:	7.641 dB	247:	11.625 dB
078:	3.703 dB	163:	7.688 dB	248:	11.672 dB
079:	3.750 dB	164:	7.734 dB	249:	11.719 dB
080:	3.797 dB	165:	7.781 dB	250:	11.766 dB
081:	3.844 dB	166:	7.828 dB	251:	11.812 dB
082:	3.891 dB	167:	7.875 dB	252:	11.859 dB
083:	3.938 dB	168:	7.922 dB	253:	11.906 dB
084:	3.984 dB	169:	7.969 dB	254:	11.953 dB
				255:	12.000 dB

14.3. DEPTH table for VIBRATO

depth	centibels	depth	centibels	depth	centibels
000:	OFF	085:	0403.1 cents	170:	0801.6 cents
001:	0009.4 cents	086:	0407.8 cents	171:	0806.2 cents
002:	0014.1 cents	087:	0412.5 cents	172:	0810.9 cents
003:	0018.8 cents	088:	0417.2 cents	173:	0815.6 cents
004:	0023.4 cents	089:	0421.9 cents	174:	0820.3 cents
005:	0028.1 cents	090:	0426.6 cents	175:	0825.0 cents
006:	0032.8 cents	091:	0431.2 cents	176:	0829.7 cents
007:	0037.5 cents	092:	0435.9 cents	177:	0834.4 cents
008:	0042.2 cents	093:	0440.6 cents	178:	0839.1 cents
009:	0046.9 cents	094:	0445.3 cents	179:	0843.8 cents
010:	0051.6 cents	095:	0450.0 cents	180:	0848.4 cents
011:	0056.2 cents	096:	0454.7 cents	181:	0853.1 cents
012:	0060.9 cents	097:	0459.4 cents	182:	0857.8 cents
013:	0065.6 cents	098:	0464.1 cents	183:	0862.5 cents
014:	0070.3 cents	099:	0468.8 cents	184:	0867.2 cents
015:	0075.0 cents	100:	0473.4 cents	185:	0871.9 cents
016:	0079.7 cents	101:	0478.1 cents	186:	0876.6 cents
017:	0084.4 cents	102:	0482.8 cents	187:	0881.2 cents
018:	0089.1 cents	103:	0487.5 cents	188:	0885.9 cents
019:	0093.8 cents	104:	0492.2 cents	189:	0890.6 cents
020:	0098.4 cents	105:	0496.9 cents	190:	0895.3 cents
021:	0103.1 cents	106:	0501.6 cents	191:	0900.0 cents
022:	0107.8 cents	107:	0506.2 cents	192:	0904.7 cents
023:	0112.5 cents	108:	0510.9 cents	193:	0909.4 cents
024:	0117.2 cents	109:	0515.6 cents	194:	0914.1 cents
025:	0121.9 cents	110:	0520.3 cents	195:	0918.8 cents
026:	0126.6 cents	111:	0525.0 cents	196:	0923.4 cents
027:	0131.2 cents	112:	0529.7 cents	197:	0928.1 cents
028:	0135.9 cents	113:	0534.4 cents	198:	0932.8 cents
029:	0140.6 cents	114:	0539.1 cents	199:	0937.5 cents
030:	0145.3 cents	115:	0543.8 cents	200:	0942.2 cents
031:	0150.0 cents	116:	0548.4 cents	201:	0946.9 cents
032:	0154.7 cents	117:	0553.1 cents	202:	0951.6 cents
033:	0159.4 cents	118:	0557.8 cents	203:	0956.2 cents
034:	0164.1 cents	119:	0562.5 cents	204:	0960.9 cents
035:	0168.8 cents	120:	0567.2 cents	205:	0965.6 cents
036:	0173.4 cents	121:	0571.9 cents	206:	0970.3 cents
037:	0178.1 cents	122:	0576.6 cents	207:	0975.0 cents
038:	0182.8 cents	123:	0581.2 cents	208:	0979.7 cents
039:	0187.5 cents	124:	0585.9 cents	209:	0984.4 cents
040:	0192.2 cents	125:	0590.6 cents	210:	0989.1 cents
041:	0196.9 cents	126:	0595.3 cents	211:	0993.8 cents
042:	0201.6 cents	127:	0600.0 cents	212:	0998.4 cents
043:	0206.2 cents	128:	0604.7 cents	213:	1003.1 cents
044:	0210.9 cents	129:	0609.4 cents	214:	1007.8 cents
045:	0215.6 cents	130:	0614.1 cents	215:	1012.5 cents
046:	0220.3 cents	131:	0618.8 cents	216:	1017.2 cents
047:	0225.0 cents	132:	0623.4 cents	217:	1021.9 cents
048:	0229.7 cents	133:	0628.1 cents	218:	1026.6 cents

049:	0234.4 cents	134:	0632.8 cents	219:	1031.2 cents
050:	0239.1 cents	135:	0637.5 cents	220:	1035.9 cents

depth	centibels	depth	centibels	depth	centibels
051:	0243.8 cents	136:	0642.2 cents	221:	1040.6 cents
052:	0248.4 cents	137:	0646.9 cents	222:	1045.3 cents
053:	0253.1 cents	138:	0651.6 cents	223:	1050.0 cents
054:	0257.8 cents	139:	0656.2 cents	224:	1054.7 cents
055:	0262.5 cents	140:	0660.9 cents	225:	1059.4 cents
056:	0267.2 cents	141:	0665.6 cents	226:	1064.1 cents
057:	0271.9 cents	142:	0670.3 cents	227:	1068.8 cents
058:	0276.6 cents	143:	0675.0 cents	228:	1073.4 cents
059:	0281.2 cents	144:	0679.7 cents	229:	1078.1 cents
060:	0285.9 cents	145:	0684.4 cents	230:	1082.8 cents
061:	0290.6 cents	146:	0689.1 cents	231:	1087.5 cents
062:	0295.3 cents	147:	0693.8 cents	232:	1092.2 cents
063:	0300.0 cents	148:	0698.4 cents	233:	1096.9 cents
064:	0304.7 cents	149:	0703.1 cents	234:	1101.6 cents
065:	0309.4 cents	150:	0707.8 cents	235:	1106.2 cents
066:	0314.1 cents	151:	0712.5 cents	236:	1110.9 cents
067:	0318.8 cents	152:	0717.2 cents	237:	1115.6 cents
068:	0323.4 cents	153:	0721.9 cents	238:	1120.3 cents
069:	0328.1 cents	154:	0726.6 cents	239:	1125.0 cents
070:	0332.8 cents	155:	0731.2 cents	240:	1129.7 cents
071:	0337.5 cents	156:	0735.9 cents	241:	1134.4 cents
072:	0342.2 cents	157:	0740.6 cents	242:	1139.1 cents
073:	0346.9 cents	158:	0745.3 cents	243:	1143.8 cents
074:	0351.6 cents	159:	0750.0 cents	244:	1148.4 cents
075:	0356.2 cents	160:	0754.7 cents	245:	1153.1 cents
076:	0360.9 cents	161:	0759.4 cents	246:	1157.8 cents
077:	0365.6 cents	162:	0764.1 cents	247:	1162.5 cents
078:	0370.3 cents	163:	0768.8 cents	248:	1167.2 cents
079:	0375.0 cents	164:	0773.4 cents	249:	1171.9 cents
080:	0379.7 cents	165:	0778.1 cents	250:	1176.6 cents
081:	0384.4 cents	166:	0782.8 cents	251:	1181.2 cents
082:	0389.1 cents	167:	0787.5 cents	252:	1185.9 cents
083:	0393.8 cents	168:	0792.2 cents	253:	1190.6 cents
084:	0398.4 cents	169:	0796.9 cents	254:	1195.3 cents
				255:	1200.0 cents

1 cent = 1/100 of a semitone
1 semitone = 1 half step

Patch editor rate table for envelopes.

Best values	good values	Overlaps
055 - 063	047 - 063	111 - 118 overlaps 061 - 063
119 - 127	111 - 127	175 - 182 overlaps 125 - 127
183 - 191	175 - 191	239 - 246 overlaps 189 - 191
247 - 255	239 - 255	