



AdLib™
Personal Computer Music System

Ad Lib Synthesizer Card Programmer's Manual™

User Guide



AdLib™

Personal Computer Music System

Music Synthesizer Card Programmer's Manual

© 1987 Ad Lib Inc.

Copyright

This manual is protected by the copyright laws and therefore may not be reproduced in whole or in part, whether for sale or not, without written consent from Ad Lib Inc. Under the copyright laws, copying includes translation into another language or format.

The Ad Lib Sound Driver is protected by the copyright laws. It may not be reproduced for other than personal use without prior written authorization from Ad Lib Inc.

The purchaser may use the program on any computer in his or her possession, but on only one computer at a time. It is possible, however, to purchase a multi-user licence authorizing the purchaser to use the program on several computers in his or her possession, including a shared disk system.

It is also possible to obtain, for a nominal fee, a licence for the use of the Ad Lib Sound Driver in third party software products. However, Ad Lib Inc. reserves the right to accept or reject any application with or without cause.

Ad Lib Personal Computer Music System, Ad Lib Music Synthesizer Card, Ad Lib Instrument Maker and Ad Lib Visual Composer are trademarks of Ad Lib Inc.

Limited Warranty

Ad Lib Inc. warrants the products that it manufactures to be free of any defects in materials and workmanship for a period of ninety (90) days from the date of purchase. This warranty is limited to the original purchaser of the product and is not transferable.

Ad Lib Inc. will refund, repair or replace, at its option, any media or documentation at no additional charge, if found defective. The purchaser is responsible for returning the product, and must provide a dated proof-of-purchase.

Each program is sold "as is", and Ad Lib Inc. will not be held responsible in any way whatsoever for direct or indirect damages of any nature resulting from the use of the program.

The purchaser has, however, the right to the legal warranty when and to the extent that it is applicable, notwithstanding any limitation or exclusion.

Technical Support

Ad Lib Inc. is firmly committed to providing the highest level of customer service and product support. If you experience any difficulties when using our product, or if it fails to operate as described, we suggest you first consult the User Guide, and then, if you are still in need of assistance, contact your dealer or call our Technical Support Department: (418) 529-6252.

Notice

Ad Lib Inc. reserves the right to make changes or improvements in the products described in this manual at any time and without notice.

Table of Contents

Introduction	5
<hr/>	
Description of the Synthesizer	7
<hr/>	
A Fundamental Part of the Synthesizer: the Operator	7
The Methods of Synthesis	8
The Types of Sounds	9
The Parameters	10
Envelope Generator Parameters	11
Attack Rate	11
Decay Rate	11
Sustain Level	12
Release Rate	12
Sustaining Sound	12
Envelope Scaling	12
Oscillator Parameters	13
Frequency Multiplier	13
Pitch Vibrato	13
Modulation Feedback	13
Level Controller Parameters	13
Output Level	13
Level Scaling	14
Amplitude Vibrato	14

Table of Contents

Programming the Sound Driver	15
Description of the Ad Lib Sound Driver	15
The Active Voice	16
Note Data	17
Timbre Data	19
Attack Rate (AR)	19
Decay Rate (DR)	19
Sustain Level (SL)	19
Release Rate (RR)	20
Sustaining Sound (SS)	20
Envelope Scaling (KSR)	20
Multiple (MULTI)	20
Frequency Vibrato (VIB)	21
FeedBack (FB)	21
Output Level (OL)	21
Key Scale Level (KSL)	22
Amplitude Vibrato (AM)	22
Frequency Modulation/Additive (FM)	22
Functions Reference	22
Function 0 : Initialize the Sound Driver and the ALMSC	24
Function 2 : Set relative time start	24
Function 3 : Set state of ALMSC	24
Function 4 : Get state of ALMSC	25
Function 5 : Flush all Sound Driver queues	25

Table of Contents

Function 6 : Set mode of ALMSC	25
Function 7 : Get mode of ALMSC	26
Function 8 : Set relative volume	26
Function 9 : Set tempo	26
Function 10: Set keyboard transpose	27
Function 11: Get keyboard transpose	27
Function 12: Set active voice	27
Function 13: Get active voice	28
Function 14: Play note with delay	28
Function 15: Play note without delay	28
Function 16: Set voice timbre parameters	29
Function 17: Set pitch	29
Function 18: Set ticks per beat	29
Function 19: Direct note on	30
Function 20: Direct note off	30
Function 21: Direct set timbre parameters	31
 Programming Guidelines	 33
General Strategy for Playing Short Melodies	33
General Strategy for Playing Long Melodies	34
Other Types of Programs	35

Table of Contents

Programming the Synthesizer	37
The Ad Lib Music Synthesizer Card	37
Operators	38
Additive Synthesis	39
FM Synthesis	39
Composite Since Wave Synthesis	39
ALMSC Input / Output Map	40
Registers Reference	41
Test Register	41
Timers	42
CSM/Keyboard Split	42
AM/VIB/EG-TYP/KSR/Multiple	44
KSL/Total Level	46
ADSR	47
BLOCK/F-Number	47
FeedBack/Connection	48
Rhythm/AM Dep/VIB Dep	49
Wave Select	49
Appendices	51
Appendix A: Basic Example and Interface	51
Appendix B: C Language Example and Interface	57
Appendix C: File Structures	63

Introduction

This guide describes and explains the programming of the Ad Lib Music Synthesizer Card as (ALMSC) well as the Ad Lib Sound Driver which is provided with the Personal Computer Music System.

The first section, "Description of the Synthesizer", briefly explains the theory behind the two synthesis methods used with the ALMSC and the role of parameters in the creation of a sound.

The second section, "Programming the Sound Driver", describes the necessary information for programming with the driver program.

The third section, "Programming the Synthesizer Card", explains the information necessary to address the card directly.

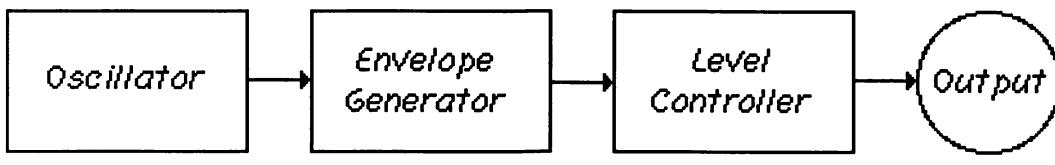
Finally, in appendices A and B, you will find an example of interfaces to the sound driver written for Basic and C languages. Since many programmers may wish to use data files from the Ad Lib software with their own programs, appendix C contains the file structures for instrument (.INS) and composition (.ROL) files.

Description of the Synthesizer

This section provides a brief description of the basic principles of sound synthesis and the different parameters which make up a sound.

A Fundamental Part of the Synthesizer: the Operator

Like most synthesizers, the Ad Lib Synthesizer is made up of oscillators and envelope generators. An oscillator and a generator are linked with a level controller to form what we call an "operator".



Components of an Operator

Refer to Section 3 for a more technical explanation of the operators of the ALMSC. The Ad Lib Music Synthesizer Card contains 18 operators which are generally grouped into pairs in order to produce instrumental sounds.

Description of the Synthesizer

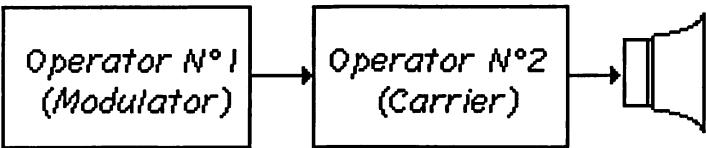
The Methods of Synthesis

The Methods of Synthesis

The Ad Lib Music Synthesizer Card generates sounds by using one of the two following methods:

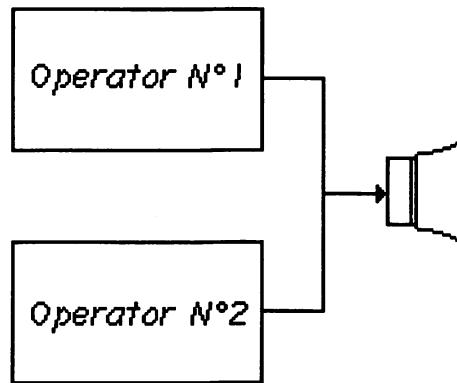
- Frequency Modulation Synthesis
- Additive Synthesis

On the ALMSC, both methods use a pair of operators to produce sounds. In frequency modulation synthesis mode, the output of the "Modulator" operator is used to modify the frequency of the "Carrier" operator.



Frequency Modulation Synthesis

In additive synthesis mode, the outputs of the two operators are combined: one of these operators produces the fundamental of the desired frequency, the other produces a "harmonic", which is an exact multiple of the fundamental. Additive synthesis with 2 operators produces mostly "organ-like" sounds. Most programs will use frequency modulation synthesis as this produces a wider variety of sounds



Additive Synthesis

The ALMSC produces either 9 melodic sounds or 6 melodic sounds plus 5 percussion instruments. For the 9 melodic sounds the operators are grouped in pairs. For the percussion group, the operators are arranged as follows:

- 6 melodic instruments (12 operators)
- 1 Bass Drum (2 operators)
- 1 Snare Drum (1 operator)
- 1 Tom-Tom (1 operator)
- 1 Cymbal (1 operator)
- 1 Hi-Hat (1 operator)

Description of the Synthesizer

The Parameters

The Parameters

Timbre is the quality of tone which is unique to an instrument and it is the timbre which allows us to recognize a particular instrument when heard. The timbre of a sound produced by the ALMSC is determined by the parameters of the three components of the operators:

- Envelope generator parameters
- Oscillator parameters
- Level controller parameters

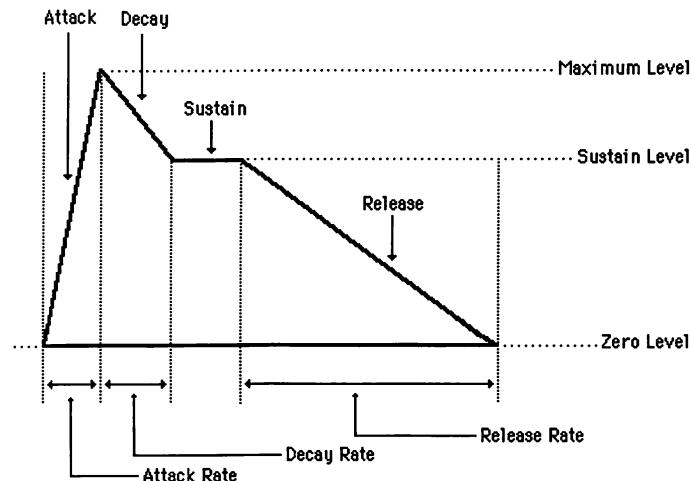
Because electronic sound is produced in a manner different from that of an acoustic instrument, the relationship between some of the following parameters and the final sound heard is difficult to explain in words. Experimentation is the best way to get a feeling for how a particular parameter will affect the final sound. (The Ad Lib Instrument Maker (TM) is a quick and easy way to test these relationships.)

Envelope Generator Parameters

The "envelope" of a sound (see chart below) describes the behaviour of a sound for its entire duration. Together, the following parameters comprise the envelope.

- **Attack Rate**

All envelopes start at level zero and work their way towards a maximum level at a speed determined by the *Attack Rate*. The envelope can reach the maximum level instantaneously or after a short delay, depending on the value given to this parameter.



An Envelope

- **Decay Rate**

After having reached the maximum level, the envelope moves towards the *Sustain Level* at a speed determined by the *Decay Rate*.

- **Sustain Level**

Once the envelope has reached the maximum level, it immediately begins diminishing towards the *Sustain Level*. The *Sustain level* can be either equal to or lower than the maximum level depending on the value that you have chosen for this parameter.

The envelope will remain on the *Sustain Level* only if the *Sustaining Sound* is in effect; if it is not in effect, the envelope will immediately start releasing and the sound will be in that case a short one. (See the description of the *Sustaining Sound* parameter below.)

- **Release Rate**

After having reached the sustain level, the envelope starts heading towards level zero at a speed determined by the *Release Rate*.

- **Sustaining Sound**

The *Sustaining Sound* parameter enables you to obtain a sound which is kept at the *Sustain Level* for as long as the note is held.

- **Envelope Scaling**

With several musical instruments, the sound envelope differs according to the pitch of the note. For example, a high note on a piano is noticeably shorter than a low note. When the *Envelope Scaling* is in effect, this phenomenon will be reproduced.

Oscillator Parameters

- **Frequency Multiplier**
The *Frequency Multiplier* enables you to modify the oscillator frequency so that the sound played becomes a multiple of the note's frequency. These multiples, the harmonics, make the sound more "intricate".
- **Frequency Vibrato**
The *Frequency Vibrato* automatically brings about a slight fluctuation to the oscillator frequency. This is sometimes referred to as Pitch Vibrato.
- **Modulation Feedback**
The oscillator used in the modulator can itself be modulated by feeding back the output signal into the input. The *Modulation Feedback* enables you to adjust the modulation of this oscillator.

Level Controller Parameters

The following parameters deal with the overall output levels of each operator.

- **Output Level**
The *Output Level* enables you to adjust an operator's maximal output level. When the two operators are used in frequency modulation, the modulator output level determines the intensity of the modulation of the carrier; the carrier's output level changes the overall volume of a sound.

Description of the Synthesizer

The Parameters

- **Level Scaling**

The sound level and timbre of acoustic instruments varies according to the pitch of the note. For example, the lower notes of a piano sound louder than the higher notes. *Level Scaling* allows you to reproduce this phenomenon; increasing the value given to this parameter decreases the loudness of the higher notes. (Also called Key Scale Level.)

- **Amplitude Vibrato**

The *Amplitude Vibrato* automatically brings about a slight fluctuation to an operator's output level.

Programming the Sound Driver

This section provides a description of the Ad Lib Sound Driver included with the Ad Lib Personal Computer Music System as well as the information necessary to program the driver.

Description of the Ad Lib Sound Driver

The Ad Lib Sound Driver is a memory resident program (size = 12K + buffer size) used to control sound generation by the Ad Lib Music Synthesizer Card (ALMSC). Your application program communicates with the Sound Driver by using a software interrupt. The Sound Driver analyses the requested operation and sends the appropriate commands to the ALMSC.

The Sound Driver operates as a background task. The requested operations are stored in a queue and are performed when appropriate. In this way, a complete song can be loaded at once and will be played at the requested tempo while the application program is performing other tasks.

When directly programming the Sound Driver, it is necessary to specify the following aspects using a series of functions described below:

- active voice
- note data
- timbre data

The Active Voice

Voice refers to a pair of operators. Active voice refers to the voice currently being worked with. When entering notes or interpretative data, it is always necessary to specify the active voice.(ref. Function 12) There are 2 voice modes possible with the ALMSC: melodic mode and percussive mode. When the card is set up in the Melodic mode there are 9 melodic voices (numbered 0 through 8). In the Percussive mode, there are 6 melodic voices and 5 percussive voices. The following table gives a cross reference for voices and their corresponding operators:

Voice		Operator number		Percussion name
Melo.	Perc.	Melodic	Percussive	
0	0	1, 4	1, 4	melo 0
1	1	2, 5	2, 5	melo 1
2	2	3, 6	3, 6	melo 2
3	3	7, 10	7, 10	melo 3
4	4	8, 11	8, 11	melo 4
5	5	9, 12	9, 12	melo 5
6	6	13, 16	13, 16	Base Drum (BD)
7	7	14, 17	17	Snare Drum (SD)
8	8	15, 18	15	Tom-Tom (TT)
---	9	---	18	Top-Cymbal(CY)
---	10	---	14	Hi-Hat (HH)

Percussive sounds are created using voices operators 13 through 18. Although some percussion voices use only one operator they are not totally independent. The card will always recognize a pair of operators for a voice regardless of its mode. Therefore, this means that a change to a percussive voice may cause another voice to change also.

Note Data

A note is composed of three elements: pitch, duration and the delay before playing the next note.

The pitch of a note is designated by an integer between -48 and 47. The value 0 corresponds to the pitch of middle C on the piano keyboard.

C	-48	-36	-24	-12	0	12	24	36
C#	-47	-35	-23	-11	1	13	25	37
D	-46	-34	-22	-10	2	14	26	38
D#	-45	-33	-21	-9	3	15	27	39
E	-44	-32	-20	-8	4	16	28	40
F	-43	-31	-19	-7	5	17	29	41
F#	-42	-30	-18	-6	6	18	30	42
G	-41	-29	-17	-5	7	19	31	43
G#	-40	-28	-16	-4	8	20	32	44
A	-39	-27	-15	-3	9	21	33	45
A#	-38	-26	-14	-2	10	22	34	46
B	-37	-25	-13	-1	11	23	35	47

Programming the Sound Driver

Description of the Ad Lib Sound Driver

The duration of a note is defined as two integers representing the numerator and the denominator of a fraction of a beat. The fraction 1/1 represents one beat, thus a quarter note. The following table illustrates the integers corresponding to the duration of the note.

Name	Num.	Den.
Whole note	4	1
Dotted half note	3	1
Half note	2	1
Dotted quarter note	3	2
Quarter note	1	1
Dotted eighth note	3	4
Eighth note	1	2
Dotted sixteenth	3	8
Sixteenth	1	4

The effective duration of a note depends on the duration of a beat. The tempo is defined as the number of beats per minute (ref. Function 9).

The third element of a note is the delay. This element is used to schedule notes. When the Sound Driver begins playing a note, it schedules the ending of the note after the specified duration. The Sound Driver also schedules the beginning of the next note. The delay element is used to determine how long after the start of the current note will the next note start playing. This delay is defined in the same format as the duration. Usually, the delay is the same as the duration. If it is longer than the duration, there will be a rest.

Rests can be expressed as notes of duration 0 with the desired delay.

Since durations and delays are defined as fractions of a beat, it is possible to speed up or slow down a song by simply changing the tempo. The relative durations of the notes will be preserved.

Timbre Data

In order to define how an instrument will sound we have to send to the ALMSC an array of various parameters as described below. (Refer to illustration on p. 11.)

- **Attack Rate (AR)** (array index = 3)
The Attack Rate determines the speed which the envelope rises from level zero to its maximum level. This value is from 0 to 15. A value of 1 generates a slow rising envelope while 15 would be a fast rising envelope.
- **Decay Rate (DR)** (array index = 6)
After having reached the maximum level, the envelope moves towards the Sustain Level at a speed determined by the Decay Rate (value = 0 to 15).
- **Sustain Level (SL)** (array index = 4)
The Sustain Level can take values from 0 to 15 (from maximum to zero level). Once the envelope has reached the maximum level, it immediately begins diminishing towards the Sustain Level. The Sustain Level can be either equal to or lower than the maximum level depending on the value that you have chosen for this parameter.

The envelope will remain on the Sustain Level only if the Sustaining Sound is in effect; if it not in effect, the envelope will immediately start releasing and the sound will be in that case a short one. (See the description of the Sustaining Sound parameter below.)

- **Release Rate (RR)** (array index = 7)
After having reached the sustain level, the envelope starts heading towards level zero at a speed determined by the Release Rate (RR = 0 to 15).
- **Sustaining Sound (SS)** (array index = 5)
The SS parameter (1 = on, 0 = off) enables you to obtain a sound which is kept at the sustain level for as long as the note is held. With this parameter on, the sound release occurs the moment the note is released (the Key-Off event) whereupon the sound diminishes according to the Release Rate.
- **Envelope Scaling (KSR)** (array index = 11)
Sound envelopes of musical instruments differ according to the pitch of the note. For example, a high note on a piano is noticeably shorter than a low note. When the KSR parameter is 1 (1 = on, 0 = off) this phenomenon will automatically be reproduced.
- **Multiple (MULTI)** (array index = 1)
Multiplies the frequencies of the carrier and modulator by the given factor in the following table.

MULTI	Factor	MULTI	Factor
0	0.5	8	8
1	1	9	9
2	2	10	10
3	3	11	10
4	4	12	12
5	5	13	12
6	6	14	15

- **Frequency Vibrato (VIB)** (array index = 10)
When this parameter is on (VIB = 1) the depth of the oscillator's frequency fluctuation is set to 7 cents. ("Cents" is a measure of pitch difference. There are 1200 cents in an octave).
- **FeedBack (FB)** (array index = 2)
Defines the multiplication factor used by the modulator to feedback its output into its input. This parameter is not used by the carrier operator.

FB	0	1	2	3	4	5	6	7
Modulation	0	$\pi/16$	$\pi/8$	$\pi/4$	$\pi/2$	π	2π	4π

- **Output Level (OL)** (array index = 8)
The Output Level enables you to adjust an operator's maximal output level. When the two operators are used in frequency modulation, the modulator's output level determines the intensity of the modulation of the carrier; the carrier's output level changes the overall volume of a sound.

Programming the Sound Driver

Description of the Ad Lib Sound Driver

The range varies from 0 to 63, with 0 being the maximum and 63 the minimum. These values can be converted to dB by multiplying them by 0.75dB.

- **Key Scale Level (KSL)** (array index = 0)
The KSL, also called Level Scaling parameter, which can take a value from 0 to 3, will reduce the output level as the pitch of the sound rises. The corresponding rate for the values 0 through 3 are 0dB, 3dB, 1.5dB, and 6dB per octave respectively.
- **Amplitude Vibrato (AM)** (array index = 9)
The AM parameter, frequently called tremolo, automatically brings about a slight fluctuation to an operator's output level. The depth = 1dB when AM = 1 and 0 when AM = 0.
- **Frequency Modulation/Additive (FM)** (array index = 12)
This flag changes the synthesis method used by the specified voice, it is only valid for the modulator operator. If FM=1 the method used will be Additive and Frequency Modulation when 0.

Functions Reference

This section describes each ALMSC driver function in detail. Refer to Appendices A and B for an example and the interface written in Basic and C languages. Here is a list of functions by function number:

<u>Number</u>	<u>Function</u>
0	Initialize Sound Driver and ALMSC..
2	Set relative time start.
3	Set state of ALMSC.
4	Get state of ALMSC.
5	Flush all Sound Driver queues.
6	Set mode of ALMSC.
7	Get mode of ALMSC.
8	Set relative volume.
9	Set tempo.
10	Set keyboard transpose.
11	Get keyboard transpose.
12	Set active voice.
13	Get active voice.
14	Play note with delay.
15	Play note without delay.
16	Set voice timbre parameters.
17	Set pitch.
18	Set ticks per beat.
19	Direct note on.
20	Direct note off.
21	Direct set timbre parameters.

Each function description specifies the following data:

- **Input**
If relevant, the parameters are listed in the order that they must appear and their data type.
- **Output**
If relevant, the type of parameter returned by the function call is indicated.

Function 0: Initialize the Sound Driver and the ALMSC

Input: ---

Output: ---

This function initializes the Sound driver and the ALMSC. The relative volume is set to 100% for all voices and the pitch is reset to 0. All voices are set in melodic mode with the appropriate parameters to generate an electric piano. All software queues are emptied and the tempo is set to 90 beats per minute.

Function 2: Set relative time start

Input: TimeNum, TimeDen: unsigned integer

Output: ---

All timing references use the variables TimeNum and TimeDen. This function sets the time origin equal to the value of TimeNum/TimeDen where TimeDen is not equal to zero. Future uses of TimeNum and TimeDen will then act relative to this origin.

Function 3: Set state of ALMSC

Input: State: integer

Output: ---

This function is used to set up parameters in the queues before starting to play the notes. When State = 1 the Sound Driver will start its internal clock and begin to play the notes that are in queue. A value of 0 will stop the clock and will suspend play.

NOTE: When sending parameters or actions to the Sound Driver make sure you don't overflow the buffer before all events that will occur at the start have been sent.

Function 4: Get state of ALMSC

Input: ---
Output: State: boolean

Function 4 returns the state of the Sound Driver. If the value returned is true (1) the Sound Driver is still playing. However, when the State is false (0) either the song is over or the Sound Driver was stopped by Function 3.

Function 5: Flush all Sound Driver queues

Input: ---
Output: ---

This command flushes all event queues in the Sound Driver but stays active. The Sound Driver will silence all voices.

Function 6: Set mode of ALMSC

Input: PercussionMode: integer
Output: ---

When PercussionMode is equal to 0, Function 6 will set the ALMSC to melodic mode. A value of 1 will set the percussion mode. (Refer to table on p. 18.)

This function will reset all relative volumes to 100%; change all voices to a piano; and reset the pitch to normal (pitch = 0). Note that the mode will affect the subsequent use of Function 12 (Set Active Voice).

Function 7: Get mode of ALMSC

Input: ---

Output: PercussionMode: integer

Function 7 returns a value corresponding to the current mode in which the ALMSC is set. A value of 0 indicates Melodic mode or 1 for percussion mode.

Function 8: Set relative volume

Input: VolNum, VolDen, TimeNum, TimeDen: unsigned integer

Output: Ok: boolean

Function 8 enables you to change the relative volume ($\text{VolNum}/\text{VolDen}$) of the active voice at the specified time ($\text{TimeNum}/\text{TimeDen}$). The relative volume must be a fraction smaller than or equal to 1. Both denominators, VolNum and TimeNum, must be different from 0. This function returns a boolean value that indicates if the queue is full (Result = 0) or if the operation was done successfully (Result = 1). ($\text{VolDen} = 1$ to 255, $\text{VolNum} = 0$ to 255).

Function 9: Set tempo

Input: Tempo, TimeNum, TimeDen: unsigned integer

Output: Ok: boolean

Function 9 sets the specified Tempo at TimeNum/TimeDen. It returns a boolean value of 1 if the operation was successful, if not it returns a 0 to indicate that the queue is full.

Function 10: Set keyboard transpose

Input: KeyTranspose: integer
Output: ---

Function 10 will offset all notes by KeyTranspose which is expressed as semitones. You can transpose your keyboard up or down depending on the sign of KeyTranspose (negative = down, positive = up). This Function is executed immediately, it is a global parameter affecting all voices.

$$\text{EffectiveNote} = \text{KeyTranspose} + \text{Note}$$

Function 11: Get keyboard transpose

Input: ---
Output: KeyTranspose: integer

This function returns the value stored by the last Set Keyboard Transpose call (Function 10).

Function 12: Set active voice

Input: ActVoice: unsigned integer
Output: ---

Function 12 changes the active voice to ActVoice. The ActVoice parameter can take values 0 through 8 for melodic mode and 0 through 10 in percussion mode. This function must be called before you start sending parameters to the Sound Driver with functions that refer to ActVoice.

Function 13: Get active voice

Input: ---

Output: ActVoice: unsigned integer

Function 13 returns the present active voice. This function returns the value stored by the last Set Active Voice call (Function 12).

Function 14: Play note with delay

Input: Pitch: integer,
LengthNum, LengthDen: unsigned integer,
DelayNum, DelayDen: unsigned integer

Output: Ok: boolean

Function 14 will play the note Pitch of length LengthNum/LengthDen and will set the delay until the next note starts to DelayNum/DelayDen. The note will be played when the previous note's delay has expired. This function returns a 1 when the operation was done successfully and a 0 when the queue is full. (Pitch = -48 to +47, LengthDen = 1 to 255, DelayDen = 1 to 255.)

Function 15: Play note without delay

Input: Pitch: integer,
LengthNum, LengthDen: unsigned integer
Output: Ok: boolean

This function is identical to Function 14 but with the delay being equal to the length (LengthNum/LengthDen) of the note.
Function 15 returns 1 if successful; 0 if not. (Pitch = -48 to +47, LengthDen = 1 to 255.)

Function 16: Set voice timbre parameters

Input: Timbre[26]: far pointer to an integer,
TimeNum, TimeDen: unsigned integer
Output: Ok: boolean

Function 16 changes the timbre of the active voice at the specified Time (TimeNum/TimeDen). Parameters for the timbre are sent via a pointer to a 26 element array that must remain valid until the actual data is sent (until Time has gone by).

Function 17: Set pitch

Input: DeltaOctave: integer must always be 0,
DeltaNum, DeltaDen: integer,
TimeNum, TimeDen: unsigned
Output: Ok: boolean

Function 17 will change the pitch in the range of -1 to +1 semitone. This change will occur at TimeNum/TimeDen. Function 17 returns a 1 for a successful operation. Note that DeltaOctave should always be 0, DeltaDen = 1 to 100, and DeltaNum = -100 to 100.

Function 18: Set ticks per beat

Input: TickBeat: unsigned integer
Output: ---

This is a low level function call which affects the computer's timer interrupt. TickBeat specifies the smallest division allowed within a beat .

All notes should fall on multiples of 1/TickBeat and every voice should obey this rule. The value should also obey the following formula:

$$18.2 \leq (\text{TickBeat} * \text{Tempo} / 60)$$

The number of interrupt per seconds will be equal to:

$$\text{FInt} = \max(60, \text{TickBeat}) * \text{Tempo} / 60$$

Function 19: Direct note on

Input: Voice: unsigned integer,
 Pitch: integer

Output: ---

Function 19 is a low level call which bypasses all queues and outputs directly to the ALMSC. The note will played until Function 20 is called. (Pitch = -48 to +47, Voice = 0 to 8 or 10 depending on the mode.)

Function 20: Direct note off

Input: Voice: unsigned integer
Output: ---

This turns off the note which was turned on by Function 19.

Function 21: Direct set timbre parameters

Input: Voice: unsigned integer,
 Timbre[26]: far pointer to integer
Output: ---

This function is identical to Function 16, but it sets the specified voice immediately and your program has to support all timings. You do not have to keep the array valid after the call because the parameters are sent directly to the ALMSC.

Programming Guidelines

Programs which use the Sound Driver to play melodies will all be similarly structured, although they may vary in details.

Programs which play one short melody on one voice are the easiest to deal with.

General Strategy for Playing Short Melodies

1. Initialize
2. Load note queues
3. Start playing notes
4. Wait until all notes have been played

The example programs (actually one program in two different languages) in Appendices A and B use this type of strategy. Note that in these programs the notes are not played the instant the ALMSC receives them because its clock is turned off in the initialization phase. Instead it will store the notes in a queue and begin playing them later when the clock is turned on. Sending notes to the ALMSC is referred to as "playing notes" regardless of whether or not the ALMSC is making the notes sound at that time. Once the clock has been turned on, which causes the ALMSC to start playing the notes in its queue, the application program must wait for the ALMSC to finish. The sound driver will be using the array of parameters it received earlier and exiting before the melody is finished may destroy these parameters.

Programming Guidelines

General Strategy for Playing Long Melodies

General Strategy for Playing Long Melodies

1. Initialize
2. Load events until the queues are full
3. Start the ALMSC playing
4. Wait until there is a space in a queue
5. Load another event in the queue
6. Repeat Steps 4 and 5 until there are no more notes
7. Wait until all notes have been played

Playing a long melody is more complicated than a short melody in that it must take care of the case where the number of notes to play exceeds the space available to store them ("buffer overflow"). There is one queue for every voice, the size of which varies dynamically. However, there is a limit on the total space available for all of the queues. In order to avoid filling up this space, commands should be sent in the order in which they will occur (event driven). For example, if we are using four voices at one time, we should send only a small number of notes to one voice, then send the equivalent number of notes (same number of beats worth) to the next voice and so on. We wish to avoid the case where we send so many notes to one voice that it leaves no room for the others . (See Appendix B of the installation guide for more information on the buffer.)

Note that we usually shut off the clock when first loading up the queues. We could have left the clock turned on and started sending the notes immediately. In this case, the ALMSC will be playing notes while they are being sent and this will work as long as the CPU is dedicated to this one task.

If the CPU has other tasks running concurrently, they might interfere with the task sending notes to the ALMSC. For this reason, it is best to load up the ALMSC's queues while the clock is stopped so that, when the clock is enabled, the ALMSC has a buffer full of notes to work with if the CPU goes off to work on another task.

With short melodies, it is feasible to code the note and timbre data within the program. This quickly becomes tedious with longer programs. Longer and more complex applications will need to read data from external files. These files can be structured as the programmer wishes or the file structures (and the actual files) of Ad Lib's Visual Composer (TM) and Instrument Maker (TM) software can be used. How to use these structures and interface with ready made files is explained in Appendix C.

Other Types of Programs

Some programs may not play melodies, but instead act on input from a user interface: the computer keyboard or a midi keyboard. Such a program could be even simpler than playing a melody because as little as three functions could be used (Functions 0, 19 and 20). Other programs will vary in the number and variety of functions they use according to their degree of complexity. Note that all programs must begin by initializing the ALMSC (Function 0).

Programming the Synthesizer

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

The Ad Lib Music Synthesizer Card

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

A white noise generator is used to create rhythm sounds. This white noise generator uses voices 7 and 8 (melodic voices), frequency information (Block, F-Number, Multi), and the proper phase output. Various rhythm sounds are produced by combining this output signal with white noise. The resulting signal is then sent to the operators. Experience has shown that the best ratio for the 2 frequencies is 3:1 (melodic voice 7 frequency = 3 times melodic voice 8 frequency). Finally, envelope information is multiplied with the wave table output. As the envelope is set for one operator which corresponds to a single rhythm instrument, the values which express that instrument's characteristics are set in the parameter registers in the same manner as for melody instruments.

Programming the Synthesizer

Operators

Operators

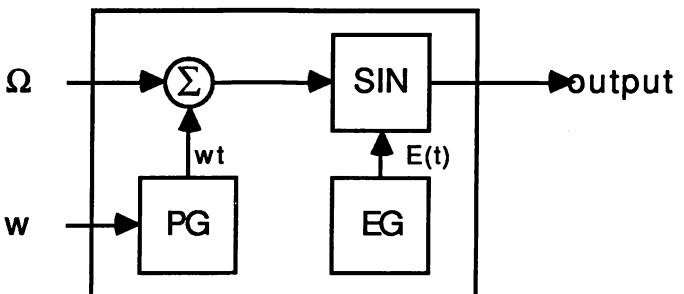
The ALMSC uses pure sine waves that interact together to produce the full harmonic spectrum for any voice. Each digital sine wave oscillator is combined with its own envelope generator to form an "operator".

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

The operator can be expressed as a mathematical expression:

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

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



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

- **Additive synthesis**

Additive synthesis connects two operators in parallel,
adding both outputs together. This method of synthesis
is not very interesting because you can only generate
organ type sounds.

The simplified formula for the additive synthesis is:

$$F(t) = E_1(t) \sin(\omega t + \Omega_1) + E_2(t) \sin(\omega t + \Omega_2)$$

- **FM synthesis**

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

$$F_m(t) = E_m(t) \sin(\omega_m t + \beta F_m(t)) \quad \text{Modulator and feedback}$$
$$F_c(t) = E_c(t) \sin(\omega_c t + F_m(t)) \quad \text{Carrier and Modulator}$$

- **Composite sine wave synthesis**

Composite sine wave synthesis (CSW) may be used to
generate speech or other related sounds by playing all
voices simultaneously: when using this mode the card
can't generate any other sounds. As no experiments
have yet been done in this direction, this feature remains
a theoretical possibility.

Programming the Synthesizer

ALMSC input/output map

ALMSC Input / Output Map

The ALMSC is located at address 388H in the i/o space. The card decodes two addresses 388H and 389H. The first address is used for selecting the register address and the second is used for writing data to the selected register. Because of the nature of the card, you must wait 3.3msec after a register select write and 23msec for a data write. Only the status register located at address 388H can be read. Here is a register map of the ALMSC:

REG	D7	D6	D5	D4	D3	D2	D1	D0
01								TEST
02								TIMER-1
03								TIMER-2
04	RST	T1	mask	T2				start/stop T1 T2
08	CSM	SEL						
20-35	AM	VIB	EG	KSR				MULTI
40-55		KSL						TL
60-75			AR					DR
80-95			SL					RR
A0-A8						F-NUMBER (L)		
B0-B8			KON		BLOCK		F-NUM (H)	
BD	DEP AM	DEP VIB	R	BD	SD	TOM	TC	HH
C0-C8						FB		C
E0-F5							WS	

When addressing a operator's register care should be taken because of holes in the addressing MAP so the correct offset to add to the register is as follows:

	Operator Address Offset								
Opr.	1	2	3	4	5	6	7	8	9
Off. (hex)	00	01	02	03	04	05	08	09	0A

Opr.	10	11	12	13	14	15	16	17	18
Off. (hex)	0B	0C	0D	10	11	12	13	14	15

Registers Reference

Test Register

This register must be initialized to zero before taking any action whatsoever.

Timers

Timer-1 is an upward 8 bit counter, it has a resolution of 80usec. If an overflow occurs, the status register flag FT1 is set, and the preset value (address = 02) is loaded into Timer-1. Timer-1 is also used for control of composite speech synthesis. When an overflow occurs in this mode, all voices are set to KEY-ON and then immediately back to KEY-OFF. Timer-2 (address = 03) is an upward 8 bit counter just like timer-1 except that the resolution is 320usec.

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

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

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

CSM/Keyboard Split

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

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

NOTE-SEL = 0

BLOCK/OCT	0	1	2	3	4	5	6	7
FNUM(MSb)	1	1	1	1	1	1	1	1
FNUM(8)	0	1	0	1	0	1	0	1
Split Num.	0	1	2	3	4	5	6	7

NOTE-SEL = 1

BLOCK/OCT	0	1	2	3	4	5	6	7
FNUM(MSb)	0	1	0	1	0	1	0	1
FNUM(8)	X	X	X	X	X	X	X	X
Split Num.	0	1	2	3	4	5	6	7

X = Don't care

AM/VIB/EG-TYP/KSR/Multiple

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

The MULTI 4 bit field determines the multiplication factor applied to the input pitch frequency in the PG section. The multiplication factor is given in the next table:

MULTI	Factor
0	1/2
1	1
2	2
3	3
4	4
5	5
6	6
7	7

MULTI	Factor
8	8
9	9
10	10
11	10
12	12
13	12
14	15
15	15

The operator formula, with the multiplication factor included; where " ∂ " is the multiplication factor.

$$F(t) = E_c(t) \sin(\partial_c w_{ct} + E_m \sin(\partial_m w_{mt}))$$

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

$$\text{Actual rate} = 4 * \text{Rate} + \text{KSR offset}$$

The KSR offset is specified in the following table:

Rate	KSR=0	KSR=1
0	0	0
1	0	1
2	0	2
3	0	3
4	1	4
5	1	5
6	1	6
7	1	7

Rate	KSR=0	KSR=1
8	2	8
9	2	9
10	2	10
11	2	11
12	3	12
13	3	13
14	3	14
15	3	15

Programming the Synthesizer

Registers Reference

The EG-Type activates the sustaining part of the envelope when the EG-Type is set ("1"). For more information on this feature, check documentation of the Instrument Maker software; most items discussed here are included.

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

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

KSL/Total Level

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

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

The Total Level (TL) attenuates the operator's output. Varying the output level of an operator functioning as a carrier results in a change in the overall level of the sound contributed to the voice by that operator. Attenuating the output from a modulator will change the frequency spectrum produced by the carrier. The TL value has a range of 0 through 63 (6 bits). To convert this value into an attenuation, apply the following formula:

$$\text{Attenuation} = (63 - \text{TL}) * 0.75\text{dB}$$

ADSR

These values change the envelope shape of the specified operator by changing the rates or the levels. The attack (AR) and the decay (DR) rates are at addresses 60H to 75H (1 per operator). The Sustain Level (SL) and Release Rate (RR) are located at addresses 80H to 95H. All these values are 4 bits in length (range 0 to 15). These parameters are well explained in the Instrument Maker's documentation.

BLOCK/F-Number

These parameters determine the pitch of the note played. The Block parameter determines the octave in which the note will be played. The F-Number (10 bits) will specify the scale. The following formula will help to determine the value to put into F-Number and Block:

$$F\text{-Num} = F_{\text{mus}} * 2^{(20-b)} / 50\text{kHz}$$

Programming the Synthesizer

Registers Reference

In this formula, F_{mus} is the desired frequency (Hz) and "b" is the block value (0 to 7). The D5 bit in the register that contains the BLOCK information is called KEY-ON (KON) and determines if the specified voice (0 to 8) is enable ("1") or disable ("0"). The lower bits of F-Number are at location A0H through A8H (1 per voice) and the 2 MSb are at position D0 and D1 of the addresses B0H to B8H.

REG	D7	D6	D5	D4	D3	D2	D1	D0
A0H-		F-Number						
A8H	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
B0H-		Block						
B8H		KEY ON	2^2	2^1	2^0	2^9	2^8	

FeedBack/Connection

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

	0	1	2	3	4	5	6	7
β	0	$\pi/16$	$\pi/8$	$\pi/4$	$\pi/2$	π	2π	4π

Rhythm/AM Dep/VIB Dep

This register allows for control over AM and VIB depth, selection of rhythm mode and ON/OFF control for various rhythm instruments. Bit D5 is used to change mode from melodic ("0") to percussive ("1") as explained at the beginning of this document (voices). Bits D0 through D4 allow for ON/OFF control of the various rhythm instruments. This means that registers B6H, B7H and B8H KON bit must always be "0".

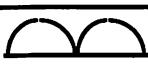
The AM Depth is 4.8dB when D7 is set ("1") and 1dB when "0". The VIB Depth is 14 cents when D6 ="1", and 7 cents when zero.

Wave Select

The WS parameter enables the card to generate other kinds of wave shapes. This is done by changing the sine function of the specified operator. The addresses of this feature are at E0H to F5H. The following figure gives the corresponding wave form:

Programming the Synthesizer

Registers Reference

D1	D0	Waveform
0	0	
0	1	
1	0	
1	1	

Appendix A

Basic Example and Interface

```
10 REM                               SOUND DRIVER BASIC INTERFACE EXAMPLE
20 REM
30 REM
40 DEFINT A-Z
50 REM
60 FSDINIT      = 0
70 FSDRELTIMESTART = 2
80 FSDSETSTATE   = 3
90 FSDGETSTATE   = 4
100 FSDFLUSH     = 5
110 FSDSETMODE    = 6
120 FSDGETMODE    = 7
130 FSDSETRELVOLUME = 8
140 FSDSETTEMPO   = 9
150 FSDSETTRANPOSE = 10
160 FSDGETTRANPOSE = 11
170 FSDSETACTVOICE = 12
180 FSDGETACTVOICE = 13
190 FSDPLAYNOTEDEL = 14
200 FSDPLAYNOTE   = 15
210 FSDSETTIMBRE  = 16
220 FSDSETPITCH   = 17
230 FSDSETTICKBEAT = 18
240 FSDNOTEON     = 19
250 FSDNOTEOFF    = 20
260 FSDTIMBRE     = 21
270 REM
280 FIN = 100
290 S0 = 0: S1 = 0: S2 = 0: S3 = 0: S4 = 0: S5 = 0: REM RESET ARGUMENTS
300 I = 0: MEM = 0: BYTE = 0
310 FUNCsize      = 66:           REM CODE SIZE
320 DIM FUNC%( FUNCsize /2 +1):  REM CODE ARRAY
330 DIM INSTRUM( 26):          REM TIMBRE DATA
340 REM
```

Appendix A

Basic Example and Interface

```
350 REM           LOAD TIMBRE INTO ARRAY, FROM DATA
360 REM
370 MEM = VARPTR( INSTRUM(0))
380 FOR I = 1 TO 52: READ BYTE: POKE MEM, BYTE: MEM = MEM +1: NEXT I
390 REM
400 REM           LOAD CODE (SOUNDBAS.ASM) INTO FUNC% ARRAY
410 REM
420 MEM = VARPTR( FUNC%(0))
430 FOR I = 1 TO FUNCSIZE
440 READ BYTE
450 POKE MEM, BYTE
460 MEM = MEM +1
470 NEXT I
480 REM
490 REM           SET UP SOUND DRIVER
500 REM
510 S0 = FSDINIT: GOSUB 730:           REM INITIALIZE SD
520 S0 = FSDRELTIMESTART: S1 = 0: S2 = 1: GOSUB 730:   REM START TIME
530 S0 = FSDSETTEMPO: S1=100: S2=0: S3=1: GOSUB 730:   REM TEMPO = 100
540 S0 = FSDSETACTVOICE: S1 = 0: GOSUB 730:           REM SET ACTIVE VOICE
550 S0 = FSDSETTIMBRE: MEM = VARPTR( FUNC(0)):        REM SET INSTRUMENT
560 CALL MEM( S0, INSTRUM(0), S2, S3, S4, S5)
570 GOSUB 630:                         REM LOAD SONG INTO SD
580 S0 = FSDSETSTATE: S1 = 1: GOSUB 730:   REM START PLAYING SONG
590 S0 = FSDGETSTATE: GOSUB 730:         REM TEST IF FINISH
600 IF S0 <> 0 GOTO 590
610 STOP
620 REM           PLAY SONGS
630 S0 = FSDPLAYNOTE
640 READ S1: READ S2: READ S3
650 IF S1 = FIN THEN RETURN
660 GOSUB 730
670 GOTO 630
680 REM
690 STOP
700 REM
710 REM           SOUND DRIVER CALL
```

Appendix A

Basic Example and Interface

```
720 REM
730 MEM = VARPTR( FUNC%(0))
740 CALL MEM( S0, S1, S2, S3, S4, S5)
750 RETURN
760 REM
770 REM instrument marimba3
780 DATA &H01, &H00, &H05, &H00, &H05, &H0D, &H00, &H01
790 DATA &H00, &H00, &H00, &H0A, &H00, &H05, &H00, &H0E, &H00, &H01, &H00
800 DATA &H00, &H00, &H00, &H01, &H00, &H02, &H00, &H01, &H00, &H00
810 DATA &H00, &H0F, &H00, &H01, &H00, &H00, &H00, &H09, &H00, &H03, &H00
820 DATA &H00, &H00, &H01, &H00, &H00, &H00, &H00, &H01, &H00
830 REM
840 REM MACHINE CODE (SOUNDBAS.ASM)
850 DATA &H06, &H56, &H57, &H8B, &HEC, &H8B, &H5E, &H14, &H8B, &H37, &H8B
860 DATA &H5E, &H0A, &HFF, &H37, &H8B, &H5E, &H0C, &HFF, &H37, &H8B, &H5E
870 DATA &H0E, &HFF, &H37, &H8B, &H5E, &H10, &HFF, &H37, &H83, &HFE, &H10
880 DATA &H75, &H07, &H1E, &HFF, &H76, &H12, &HEB, &H06, &H90, &H8B, &H5E
890 DATA &H12, &HFF, &H37, &H16, &H07, &H8B, &HDC, &HCD, &H65, &H8B, &HE5
900 DATA &H8B, &H5E, &H14, &H89, &H07, &H5F, &H5E, &H07, &HCA, &H0C, &H00
910 REM
920 REM           SONG #1
930 DATA 0, 1, 2
940 DATA 2, 1, 2
950 DATA 4, 3, 4
960 DATA 7, 1, 4
970 DATA 7, 1, 1
980 DATA 4, 3, 4
990 DATA 0, 1, 4
1000 DATA 4, 3, 2
1010 DATA 2, 1, 2
1020 DATA 0, 2, 1
1030 REM
1040 REM           SONG #2
1050 DATA 0, 1, 2, 4, 1, 2, 7, 1, 2, 12, 3, 2, 2, 3, 4, 5, 1, 4, 9, 1, 2, 12, 3, 2
1060 DATA 11, 3, 4, 9, 1, 4, 7, 1, 2, 5, 1, 1, 2, 1, 2, 4, 1, 1, 9, 1, 2, 7, 3, 2
1070 DATA 100, 100, 100
```

Appendix A

Basic Example and Interface

SOUNDBAS.ASM

Marc Savary, Editions Ad Lib., 3-06-87

assembler interface to the Sound Driver for Basic programs

This code should be included in a Basic array and used with the CALL command : func = VARPTR (array(0)): CALL func(s0,s1,s2,)

sound_driver_int	equ	101	;SD interrupt number
fSDInit	equ	0	
fSDRelTimeStart	equ	2	
fSDSetState	equ	3	
fSDGetState	equ	4	
fSDFlush	equ	5	
fSDSetMode	equ	6	
fSDGetMode	equ	7	
fSDSetRelVolume	equ	8	
fSDSetTempo	equ	9	
fSDSetTranspose	equ	10	
fSDGetTranspose	equ	11	
fSDSetActVoice	equ	12	
fSDGetActVoice	equ	13	
fSDPlayNoteDel	equ	14	
fSDPlayNote	equ	15	
fSDSetTimbre	equ	16	
fSDSetPitch	equ	17	
fSDSetTickBeat	equ	18	
fSDNoteOn	equ	19	
fSDNoteOff	equ	20	
fSDTimbre	equ	21	

Code SEGMENT BYTE
assume cs:code

Appendix A

Basic Example and Interface

```
; SoundBasic( s0, s1, s2, s3, s4, s5)
; int * s0, * s1, * s2, * s3, * s4, * s5;
;
; s0:      function number
; s1 - s5: arguments
;
; return result in s0.
;
SoundBasic PROC FAR
    PUBLIC SoundBasic
frames STRUC
    dw    ?          ; old di
    dw    ?          ; old si
old_es dw    ?          ; old ES
    dd    ?          ; return addr
    s5    dw    ?          ; ptrs to s5 argument
    s4    dw    ?          ; ... s4
    s3    dw    ?
    s2    dw    ?
    s1    dw    ?
    s0    dw    ?          ; ... ptr to function number
frames ENDS

    push  es
    push  si
    push  di
    mov   bp, sp
    mov   bx, [bp].s0
    mov   si, [bx]        ; get function number

    mov   bx, [bp].s5
    push  [bx]
    mov   bx, [bp].s4
    push  [bx]
    mov   bx, [bp].s3
    push  [bx]
```

Appendix A

Basic Example and Interface

```
        mov    bx, [bp].s2
        push   [bx]
        cmp    si, fSDSetTimbre ; s1 is a pointer ??
        jne    suite
        push   ds               ; we need a Far prt ...
        push   [bp].s1
        jmp    ok
suite:  mov    bx, [bp].s1
        push   [bx]
ok:     push   ss               ; set ES:BX to point to list of arg.
        pop    es
        mov    bx, sp
        int    sound_driver_int
        mov    sp, bp

        mov    bx, [bp].s0      ; store return value in S0
        mov    [bx], ax
        pop    di
        pop    si
        pop    es
        ret    12              ; 6 words arguments
SoundBasic ENDP

Code  ENDS
END
```

Appendix B

C Language Example and Interface

*/

16/06/87

Sound-driver demonstration program.

*/

```
/* Define function numbers *.
#define fSDInit          0
#define fSDRelTimeStart  2
#define fSDSetState       3
#define fSDGetState       4
#define fSDFlush          5
#define fSDSetMode         6
#define fSDGetMode         7
#define fSDSetRelVolume   8
#define fSDSetTempo        9
#define fSDSetTranspose    10
#define fSDGetTranspose   11
#define fSDSetActVoice    12
#define fSDGetActVoice    13
#define fSDPlayNoteDel    14
#define fSDPlayNote       15
#define fSDSetTimbre      16
#define fSDSetPitch        17
#define fSDSetTickBeat    18
#define fSDNoteOn          19
#define fSDNoteOff         20
#define fSDTimbre          21
```

```
#define END             100 /* indicate the end of 'melodie' array */
```

```
/* Timbre data array */
```

Appendix B

C Language Example and Interface

```
int marimba3[] =
0x0001, 0x0005, 0x0005, 0x000d, 0x0001, 0x0000, 0x000a, 0x0005, 0x000e,
0x0001, 0x0000, 0x0000, 0x0001, 0x0002, 0x0001, 0x0000, 0x000f, 0x0001,
0x0000, 0x0009, 0x0003, 0x0000, 0x0001, 0x0000, 0x0000, 0x0001

};

/* Array of notes to be played */
char melody[] = {
0, 1, 2,
2, 1, 2,
4, 3, 4,
7, 1, 4,
7, 1, 1,
4, 3, 4,
0, 1, 4,
4, 3, 2,
2, 1, 2,
0, 2, 1,

0,1,2,
4,1,2,
7,1,2,
12,3,2,
2,3,4,
5,1,4,
9,1,2,
12,3,2,
11,3,4,
9,1,4,
7,1,2,
5,1,1,
2,1,2,
4,1,1,
9,1,2,
7,3,2,
100, 100, 100
};
```

```
extern char SoundCall();           /* interface to sound-driver */  
  
main()  
{  
    int i;  
  
    if( !GetSoundDrvVersion() ) {      /* Is sound-driver installed? */  
        printf( "\n Sound-driver not installed!" );  
        exit( 1 );  
    }  
    SoundCall( fSDInit);             /* reset sound-driver */  
    SoundCall( fSDSetState, 0);       /* make sure driver is off */  
    SoundCall( fSDSetMode, 0);        /* set to melodic mode */  
    SoundCall( fSDSetTickBeat, 4);    /* 4 ticks per beat */  
    SoundCall( fSDReTimeStart, 0, 1); /* start of music piece */  
    SoundCall( fSDSetTempo, 100, 0, 1); /* set tempo to 100 */  
    SoundCall( fSDSetActVoice, 0);    /* use voice 0 */  
    SoundCall( fSDSetTimbre, &marimba3[ 0], 0, 1); /* set timbre voice */  
/*  
 * Play all notes of 'melody' array....  
*/  
    i = 0;  
    while(END != melody[ i])  
    {  
        SoundCall ( fSDPlayNote, (unsigned)melody[ i], (unsigned)melody[ i+1]  
                  (unsigned)melody[ i+2]);  
        i += 3;  
    }  
    SoundCall( fSDSetState, 1);        /* turn on the sound-driver */  
    while( SoundCall( fSDGetState))    /* wait until the last note */  
        ;  
    printf( "\nDone!");  
    exit( 0);  
}
```

Appendix B

C Language Example and Interface

CSOUND.ASM

interface to resident sound-driver for LATTICE C compiler, LARGE model.

87/03/18 Ad Lib.

```
INCLUDE DOS.MAC      ; memory models ...
INCLUDE DEFS.MAC     ; equates & sound-driver version proc.
```

PSEG

```
INCLUDE VERSION.MAC   ; sound-driver signature
```

```
public GetSoundDrvVersion
```

```
; unsigned GetSoundDrvVersion()
; if the sound-driver is charged in memory, return his
; version number, else 0.
```

```
DrvVersionProc GetSoundDrvVersion
```

```
; int SoundCall( functionNumber, arg_list)
; int functionNumer;
; any... arg_list
```

```
; Generate interrupt to sound-driver with parameter's address
; in ES:BX and function number in SI
```

```
BEGIN SoundCall
```

```
        IF LDATA EQ 0
sframe STRUC
    dw    ?          ; old ES
    dw    ?          ; old BP
    db    CPSIZE DUP (?) ; return addr
args  dw    ?
sframe ENDS
ELSE
sframe STRUC
    dw    ?          ; old BP
    db    CPSIZE DUP (?) ; return addr
args  dw    ?
sframe ENDS
ENDIF

        push   bp
IF LDATA EQ 0
push   es
ENDIF
mov    bp, sp
mov    si, [bp].args      ; get function number
lea    bx, [bp].args+2    ; get pointers to others args...
push   ss
pop    es
int    sound_driver_int  ; call sound-driver ...
IF LDATA EQ 0
pop    es
ENDIF
pop    bp
ret
SoundCall ENDP

ENDPS
end
```

Appendix C

File Structures

Files containing instrument timbre information are suffixed with ".INS".

Structure of .INS files:

field#	size (bytes)	type	description
1	1	char	mode: 0 => melodic 1 => percussive
2	1	char	if percussive, voice number (6 - 10)

Modulator (operator 0):

3	2	int	KSL
4	2	int	frequency multiplier
5	2	int	feed back
6	2	int	attack rate
7	2	int	sustain level
8	2	boolean	sustaining sound
9	2	int	decay rate
10	2	int	release rate
11	2	int	output level
12	2	boolean	AM
13	2	boolean	VIB
14	2	int	KSR
15	2	boolean	FM

Appendix C

File Structures

Carrier: (operator 1), significant for melodic voices and percussive voice 6 (Bass Drum) only:

16	2	int	KSL
17	2	int	frequency multiplier
18	2	int	unused
19	2	int	attack rate
20	2	int	sustain level
21	2	boolean	sustaining sound
22	2	int	decay rate
23	2	int	release rate
24	2	int	output level
25	2	boolean	AM
26	2	boolean	VIB
27	2	int	KSR
28	2	int	unused

Files containing note information (i.e. songs) are suffixed with ".ROL". ("File version" and "editing scale" are non-musical information used by Visual Composer (TM).)

Structure of .ROL files:

field#	size (bytes)	type	description
1	2	int	file version, major
2	2	int	file version, minor
3	40	char	unused
4	2	int	ticks per beat
5	2	int	beats per measure
6	2	int	editing scale (Y axis)
7	2	int	editing scale (X axis)
8	1	char	unused
9	1	char	mode: 0 => melodic 1 => percussive
10	90	char	unused
11	38	char	filler
12	15	char	filler
13	4	float	basic tempo

Field 14 indicates the number of times to repeat fields 15 and 16:

14	2	int	number of tempo events
15	2	int	time of event, in ticks
16	4	float	tempo multiplier (0.01, 10.0)

Appendix C

File Structures

The remaining fields (17 to 34) are to be repeated for each of 11 voices:

17	15	char	filler
18	2	int	time (in ticks) of last note +1

Repeat the next two fields (19 and 20) while the summation of field 20 is less than the value of field 18:

19	2	int	note number: 0 => silence from 12 to 107 => normal note (you must subtract 60 to obtain the correct value for the sound- driver)
20	2	int	note duration, in ticks
21	15	char	filler

Field 22 indicates the number of times to repeat fields 23 to 26:

22	2	int	number of instrument events
23	2	int	time of event, in ticks
24	9	char	instrument name
25	1	char	filler
26	2	int	unused
27	15	char	filler

Field 28 indicates the number of times to repeat fields 29 and 30:

28	2	int	number of volume events
29	2	int	time of event, in ticks
30	4	float	volume multiplier (0.0, 1.0)

Field 32 indicates the number of times to repeat fields 33 and 34:

32	2	int	number of pitch events
33	2	int	time of event, in ticks
34	4	float	pitch variation (0.0, 2.0, nominal is 1.0)

ISBN 2-920858-10-6



AdLib™

Personal Computer Music System

Ad Lib Inc.
220 Grande-Allée East, Suite 960
Québec, QC, Canada G1R 2J1

50 Staniford Street, Suite 800
Boston, MA 02114