APPENDIX C

COLECOVISION
SOUND USERS' MANUAL

Version 1.1
May 3, 1982

COLECOVISION SOUND USERS' MANUAL

*** CONTENTS ***

The Colecovision operating system includes routines which allow the cartridge programmer to store "songs" and simple sound effects in tabular form in cartridge ROM, and play them on request during the game. More complex, special sound effects can also be created and played within the same data structure and procedural format. This Sound Users' Manual describes the data structures expected by and the use of the operating system sound routines.

SUMMARY OF FEATURES

* six song note types: combinations of fixed or variable frequency and attenuation, "noise" (percussion) notes, and rests

* hierarchical structure of local data areas assigned to each sound channel, which allows the temporary "overwriting" of lower priority songs (i.e., lower priority songs are not truncated by higher priority songs or sound effects that use the same channel, but continue unheard until the higher priority songs finish)

* the ability to easily include a special sound effect (say, a cymbal crash) as part of a song composed primarily of musical tones

* both songs and special, independent sound effects (e.g., an explosion sound) can utilize the same data structures and output procedures

* sweep routines, which automatically create frequency or attenuation sweeps, simplify note data storage and can be used by special sound effects

* song end codes allow songs to be played once, or automatically restarted upon completion (repeat forever)

GENERAL DESCRIPTION

* Song data areas, SxDATA: RAM map mode of sound chip operation

The Colecovision operating system provides sound routines which output frequency, attenuation, and control data to the TI 76489 sound chip. Data to be sent to a particular sound generator channel is expected to be stored within a ten byte block of CPU RAM called a "song data area". A song data area, then, contains a RAM record of the current values "playing" on a sound channel.

Each song data area can also contain timing and descriptive information which allows for simple generation of musical notes. A "song" can be created by storing in CART ROM a list of note parameters which specify note duration, frequency, and attenuation. When a song is started, O/S routines are provided to load the data describing the first note of the song into a song data area. Each song data area is then processed at regular intervals by routines which modify and output the area's data to the sound chip. When a note is completed, the next note in the song is automatically loaded and the process continues.

O/S routines also exist which facilitate the creation of "special effects": sound routines written by the cartridge programmer which algorithmically generate data to be sent to the sound chip (as opposed to the table look-up, song approach).

RAM space for at least four song data areas must be reserved by the cartridge programmer: one each to describe the current status of the four sound chip channels. More than four song data areas will be required if the ability to "overwrite" lower priority songs is desired, and some songs may share the same data area (see the following discussion, "Hierarchy of song data areas: priority, truncation, overwriting"). The first byte in each song data area, byte 0 (its offset from the beginning of the data block = 0), contains the channel number upon which the song is to be played (0: noise generator, 1 to 3: tone generator) and the song's identification number (SONGNO: 1 to 61). A song data area is referred to throughout the rest of this manual as "SxDATA", where x = the song's SONGNO. For a detailed description of each byte in a song data area, see the following discussion, "NOTES", and refer to Figure 1.

* Note list storage and note headers

A note list is a sequential list of frequency and timer data stored in cartridge ROM that, when processed and output to the sound chip, create the notes that comprise a song. Each block of 1 to 8 bytes of data that describes a note in the list must begin with a one byte header which contains information (bit flags or values) that indicate (see Figure 2):

1) The number of the channel upon which to play the note
2) The note type (one of 4 combinations of fixed or swept frequency and attenuation, plus a rest).

The single byte header can also be used as an end-of-song marker, a repeat-song indicator, or an indicator that a "note" is to be determined algorithmically by a special sound effect routine (the starting address of which immediately follows).

A 16 bit pointer to the location of the header of the next note to be played (NEXT_NOTE_PTR) is maintained by the O/S routine SND_MANAGER in each song's data area (SxDATA, offsets 1 and 2).

* LST_OF_SND_ADDRS and PTR_TO_LST_OF_SND_ADDRS

The O/S routines expect the ten byte long song data areas to be stored contiguously in CPU RAM, starting with the data area used by song number one. The beginning addresses of each of these data areas, as well as the addresses of the headers of the first note in each song, are stored in a ROM table called LST_OF_SND_ADDRS (see Figure 3): The cartridge programmer may place this table wherever desired in ROM. The O/S routines know its starting address through a dedicated CPU RAM location, PTR_TO_LST_OF_SND_ADDRS, which must be loaded with the 16 bit address of the table by the cartridge program before calling any O/S routines which use it (see description of INIT_SOUND).

* Hierarchy of song data areas: truncation, priority, overwriting

The routine that does the processing of the note data stored in the song data areas, SND_MANAGER, and the routine that outputs the modified data to the sound chip, PLAY_SONGS, are designed to be called by the cartridge program every Video Display Processor (VDP) interrupt (every 16.7 ms). Starting with the data area for song number one, SND_MANAGER processes the appropriate timer and sweep counters and modifies the frequency and attenuation data accordingly. If the data area is assigned to a special effect, SND_MANAGER calls that effect. When a note is finished, SND_MANAGER, using the data area's next note pointer, moves data for the next note of the song into the area.

After the operations upon a data area have been performed, the sound chip channel number (CH#) stored in byte 0 of that data area is consulted and the appropriate "channel data area pointer" (PTR_TO_S_ON_x) is set to point to the beginning of the data area just processed (four of these pointers exist at dedicated 16 bit locations in CPU RAM, one for each of the sound generator channels). The following data areas are then processed in the same fashion, in order of occurence, until the end of data area code, 00, is reached. If a data area is inactive, i.e., if the song(s) that use it aren't playing at the moment, SND_MANAGER simply passes it over, doing no processing or channel data area pointer modification.

PLAY_SONGS, usually called immediately prior to SND_MANAGER, outputs data to the sound chip from the four song data areas pointed to by the channel data area pointers. Thus, a channel output priority is established on the basis of ordinal position within the data area block: the last data area processed that uses a given channel is the one that will be played on that channel. E.g.,

order of data area within data block containing all song data areas	songs that use this data area: SONGNO/CH# song is to be played on	
1st	1/CH#x	(remember: although other songs may use this data area also , song 1 MUST use it)
...		
5th	6/CH#2	
6th	3/CH#2	
7th	11/CH#2	
...		
10th	2/CH#3; 4/CH#3; 7/CH#3	

First, consider channel 2. Let's say that the only songs which use channel 2 are assigned three contiguous song data areas, 5th through 7th (grouping songs which use the same channel isn't necessary as far as the code is concerned, but it makes it simpler to think about). SND_MANAGER, as it makes its way through the song data areas in order, will process the 5th data area (which "belongs" to song 6) and set PTR_TO_S_ON_2 to the address of byte 0 in the 5th data area. Then, the 6th area will be processed, resulting in resetting PTR_TO_S_ON_2 to the 6th data area (song number 3). ' Likewise, the 7th data area will be processed, finally leaving PTR_TO_S_ON_2 pointing to the 7th data area (song number 11). The next time PLAY_SONGS is called, it will send to sound chip channel 2 frequency and attenuation data in the data area pointed to by PTR_TO_S_ON_2, namely, the data for song number 11.

Note that although only song 11 will be heard on channel 2 this pass through PLAY_SONGS, the timers and data for songs 6 and 3 were nonetheless modified, regardless of the existence of the higher priority song 11. That is, all songs "keep going", whether or not their data will be output during PLAY_SONGS. Songs 6 and 3 are said to have been "overwritten" by song 11. Should song 11 become inactive (end) before song 6 and/or song 3, then the highest priority of the remaining active songs (i.e., the last data area within the block of data areas to use a given channel) will be heard.

Thus, assigning several data areas to songs which use the same channel allows the creation of "background" songs which can be momentarily interrupted (overwritten) by a higher priority song or sound effect (e.g., an explosion, or a bonus song) and continue on after the overwriting song is over.

Now examine the 10th song data area. Again, let's say that the only songs that use channel 3 are shown here and that they all share the 10th data area. In this case, the programmer may have arranged things such that songs 2, 4, and 7 are never active simultaneously: i.e., there was no reason to assign three different data areas for songs which never overlap. If, however, this is not the case and, say, song 4 may be started before song 7 is finished, the O/S routines would stop song 7 in favor of song 4. That is, for songs that share the same data area, the most recent song started is the song heard, and interrupted songs do not continue: i.e., songs sharing the same data area truncate each other. In many cases this may be both acceptable and desirable, as it saves RAM space.

NOTE: The preceeding description states that a channel data area pointer is updated every time SND_MANAGER processes a song data area: this is actually not the case. To save processing time, a routine which updates all the pointers is called only when, after loading the next note in a song, SND_MANAGER detects that the data area's CH# or SONGNO has changed. This happens whenever the next note: 1) uses a different channel (see "Noise notes: special case Type 2 notes), 2) is a special effect note, or 3) is an end-of-song indicator. It is only necessary to update the channel data area pointers in these cases, and when a new song is started (in JUKE_BOX). See "Pseudo code versions of main routines".

* The four basic routines, briefly

The following four O/S routines are the only ones that need be called to create songs which use the six standard note types (more complete descriptions of each routine can be found in the "OPERATING SYSTEM ROUTINES" section):

INIT_SOUND: This routine should be called immediately after power on, before any sound processing can occur. It turns off the sound generators, initializes the CART RAM locations to be used as song data areas, sets up the four channel data area pointers, and initializes PTR_TO_LST_OF_SND_ADDRS.

INPUT: n
TYPE: 8 bit constant
PASSED: in B
DESCRIPTION: number of song data areas used by the game

INPUT: LST_OF_SND_ADDRS
TYPE: 16 bit address
PASSED: in HL
DESCRIPTION: LST_OF_SND_ADDRS is the base address of a list of the starting addresses of each song's data area and note list.

OUTPUT: 1) turns off all sound generators
2) initializes PTR_TO_LST_OF_SND_ADDRS
3) writes the inactive code (0FFH) to byte 0 of the n song data areas
4) stores 00 at end of song data areas
5) sets the 4 channel song pointers to a dummy inactive area
6) sets SAVE_CTRL to 0FFH (see "Noise notes" discussion)

JUKE_BOX: JUKE_BOX is called to start a song. Using a song number passed in B, JUKE_BOX loads the data for the song's first note into the appropriate song data area, thereby truncating whatever song had been "playing" in that data area. (The address of the appropriate area is found by using the song number as an index into the LST_OF_SND_ADDRS table). It also formats the data area's header and sets up the next note pointer. If the song is a special sound effect, its next note pointer is set to the address of the special effect routine. The next time PLAY_SONGS is called, that song's first note will be played.

If JUKE_BOX is called with a song number of a song already in progress, it returns immediately (i.e., it doesn't restart the song).

INPUT: song number to be started
TYPE: 8 bit constant, 1 to 61
PASSED: in B

CALLS: PT_IX_TO_SxDATA, LOAD_NEXT_NOTE_PTR, UP_CH_DATA_PTRS

OUTPUT: 1) moves the song's first note data to the appropriate song data area
1) formats byte 0 header of the song's data area
2) points next note pointer in data area (bytes 1&2) to address of first note in song, or address of special sound effect routine

SND_MANAGER: SND_MANAGER should be called every VDP interrupt (every 16.7 ms). For each data area, SND_MANAGER processes the appropriate timer and sweep counters and modifies the frequency and attenuation data accordingly. If the data area is assigned to a special effect, SND_MANAGER calls that effect. When a note is finished, SND_MANAGER, using the data area's next note pointer, moves data for the next note of the song into the area. If SND_MANAGER reads a header byte (in CART ROM) that has bits 3&4 set, indicating repeat song, it will start the song again by reloading the first note in the song.

After the operations upon a data area have been performed, if necessary, the channel data area pointers (PTR_TO_S_ON_x) are updated. The following data areas are processed in the same fashion, in order of occurence, until the end of data area code, 00, is reached.

SND_MANAGER does not output the modified frequency and attenuation data. PLAY_SONGS is called just before SND_MANAGER to do this.

Special codes in byte 0 of the song data area indicate:

255: data area inactive, do no processing
62: a special effect is to be played; SND_MANAGER calls the effect routine
0: end of song data areas (SND_MANAGER processes data areas until it sees 0 in byte 0)

NOTE: Song number 1 MUST use the first area in the block of song data areas.

INPUT: none

CALLS: PT_IX_TO_SxDATA, PROCESS_DATA_AREA

OUTPUT: Calls routines which:
1) decrement song duration and sweep timers
2) modify swept frequency and attenuation values
3) call special effects routines where necessary
4) update the channel data area pointers if necessary
5) restart the song if indicated

PLAY_SONGS: PLAY_SONGS takes the frequency and attenuation data pointed to by the four channel data area pointers (PTR_TO_S_ON_x) and outputs it to the four sound chip generators.

INPUT: none

CALLS: TONE_OUT, UPATNCTRL

OUTPUT:
1) current freq and atn data is output to each tone generator, if song/effect on that channel is active; if song on that channel is inactive, that generator is turned off
2) noise generator is sent current atn data, and control data, if new
3) modifies SAVE_CTRL if necessary

These four routines would normally be called as follows:

```
power on inits done by O/S
cartridge program receives control:
LD B, # of song data areas used in the game
LD HL, address where LST_OF_SND_ADDRS is stored in ROM
CALL INIT_SOUND to initialize song data areas
whatever other power on inits you want to do
start game:
.
.
.
LD B, # of song you want to start
CALL JUKE_BOX to set up for start of song
.
.
VDP interrupt occurs:
CALL PLAY_SONGS to output data
CALL SND_MANAGER to process song data
whatever else you want to do during VDP interrupts
RETN to game
```

NOTES

* Terminology

Each note in a song has an associated 10 bit frequency (except for noise notes) and 4 bit attenuation which is output to the sound chip every time PLAY_SONGS is called. The initial frequency and attenuation values (stored in 2 bytes) are part of a block of 4 to 8 bytes that describe a single note within a song's ROM note list. The remaining bytes are used to indicate sound channel, note type, duration, and various timers and values associated with swept notes.

The following are explanations of names and symbols used throughout this manual to refer to bytes, or segments of bytes, within both a note's ROM note list and a RAM song data area (see Figure 1):

!: ! is a symbol used to graphically separate bits or nibbles within a byte

Bx: means bit x of a byte, bit 7 being the most significant bit

byte x: refers to the offset of a byte within a data block, byte 0 being the first byte in the block

MSN, LSN: MSN means the most significant nibble of a byte, LSN is the least significant nibble

CH#: the sound channel upon which a note is to be played; 0 = noise generator, 1 to 3 = the 3 tone generators

SONGNO: the song number of the song playing in a song data area; 1 to 61; SONGNO 62 means a special sound effect is using the data area

NEXT_NOTE_PTR: 2 byte address of the ROM location of the data block for the next note to be played in a song

F0 - F9: the 10 bit frequency data to be sent to a sound chip tone generator; F0 is the most significant bit; see data sheets, TI 76489

ATN: 4 bit attenuation data to be sent to any of the four sound generators

CTRL: 3 bit control data for sound chip noise genrator; FB NF0 NF1 (called SHIFT in this manual); see data sheets for details

NLEN: 1 byte that directly or indirectly determines the duration of a note

FPS: 4 bit frequency prescaler, used with NLEN to determine the length of a frequency sweep

FPSV: 4 bit temporary storage location for FPS; this variable gets decremented every VDP interrupt, and is reloaded from FPS

FSTEP: the size of a step in a frequency sweep, an 8 bit two's complement signed value that is added to the current 10 bit frequency at a rate determined by NLEN and FPS; 1 to 127, -1 to -128

ALEN: 4 bit number of steps in an attenuation sweep

ASTEPS: ASTEPS is the signed, 4 bit size of a step in an attenuation sweep; can take values 1 to 7, -1 to -8 (MSB = 1 means negative)

APS: 4 bit attenuation prescaler, used with ALEN to determine the length of a frequency sweep

APSV: 4 bit temporary storage location for APS; this variable gets decremented every VDP interrupt, and is reloaded from APS

* Frequency sweeps and note duration

The time duration of a note = the number of passes by PLAY_SONGS through the note times 16.7ms (the VDP interrupt period). (note that the time the note is HEARD could be shorter: see "Attenuation sweeps" discussion) The number of PLAY_SONGS passes is always determined directly or indirectly by NLEN. NLEN, however, has two meanings, depending upon whether or not a note's frequency is swept:

Fixed frequency notes - In this case, NLEN is decremented every VDP interrupt and therefore directly determines the length of a note:

duration = NLEN * 16.7ms

NLEN should have values in the range 0 to 255 (0 => 256), giving a maximum duration of ~ 4.25 seconds.

Swept frequency notes - Here, the prescaler variable, FPSV, is decremented until zero before NLEN is decremented. Once FPSV goes to zero, it's reloaded from FPS; however, an initial value for FPSV, which enters into the calculation of the the length of the first step in the sweep, is stored in ROM along with the rest of the note's initial data, and it may be different from the reload value, FPS. For a frequency swept note:

note duration = [((NLEN -1) * FPS) + initial FPSV] * 16.7ms

So, NLEN again determines note duration, but in an indirect fashion (in concert with FPS and the initial FPSV).

In the case of a frequency swept note, NLEN can be thought of as one of four parameters that describe the sweep: 1) the starting frequency, 2) FSTEP, a signed step size, i.e., a delta frequency that is periodically added to the current frequency, 3) the prescaler value (FPS) which determines the length of time at any one frequency step, and 4) NLEN, the number of steps in the sweep.

The duration of each step in the sweep is given by the following:

duration 1st step = initial FPSV * 16.7ms
duration all others = FPS * 16.7ms

Frequency sweep parameter ranges:

FSTEP - signed 8 bit two's complement number: 1 to 127, -1 to -128; an FSTEP of 0 tells FREQ_SWEEP that the note is not frequency swept, and the note's duration is determined by directly decrementing NLEN (prescaler is disregarded)

FPS - 4 bit frequency prescaler, used with NLEN to determine the length of a frequency sweep: 0 to 15 (0 => 16)

FPSV - 4 bit temporary storage location for FPS; this variable gets decremented every VDP interrupt, and is reloaded from FPS: 0 to 15 (0 => 16)

NLEN - 8 bit note duration for a fixed frequency note: 0 to 255 (0 => 256)
8 bit number of steps for a swept frequency note: 2 to 255 (0 =>256)

Note durations:

Fixed frequency -	NLEN * 16.7ms
Swept frequency -	[(NLEN -1) * FPS) + initial FPSV] * 16.7ms
duration 1st step =	initial FPSV * 16.7ms
duration all others =	FPS * 16.7ms

* Attenuation sweeps

Volume attacks and decays can be thought of as attenuation sweeps: a sweep from low to higher volume is an attack, a sweep in the other direction is a decay. Attenuation sweeps are created in a similiar fashion to the frequency sweeps described above, the primary difference being that attenuation sweep parameters don't take on 8 bit values. The full volume range for the attenuation registers on the 76489 chip is 0 (ON) to 15 (OFF), so step sizes and number of sweep steps greater than 4 bits aren't generally useful.

Just as in the case of a frequency swept note, attenuation sweeps have four parameters that describe the sweep: 1) the starting attenuation, 2) ASTEP, a signed step size, i.e., a delta attenuation that is periodically added to the current attenuation, 3) the prescaler value (APS) which determines the length of time at any one attenuation step, and 4) ALEN, the number of steps in the sweep.

The prescaler parameters, APS and APSV, are the same size (4 bits) and mean exactly the same thing as their frequency counterparts. ALEN, the number of steps in the sweep, is only 4 bits (compared to an FLEN of 8 bits), but 15 steps of even the smallest step size (+/-1) can sweep a generator from full on to full off. ASTEP, in order to squeeze it into a nibble, has been limited to a 4 bit signed number (3 bits data, 1 bit sign). This gives a range of step values from 1 to 7, -1 to -8. This shouldn't be too limiting, since most attenuation sweeps are implemented with the smallest step size.

NOTE: Recall that, as far as SND_MANAGER is concerned, the length of a note is determined, directly or indirectly, only by NLEN, a timer/counter that is decremented during FREQ_SWEEP; i.e., the duration of a note is independent of what is happening to its attenuation. Therefore, the programmer should take care to see that an attenuation sweep isn't inadvertently created that ramps the volume down to off before SND_MANAGER, through NLEN, has decided that the note is over. However, since ATN_SWEEP simply leaves the attenuation alone once it's finished a sweep, the independence of attenuation sweep length and note length may be put to good use: e.g., a sforzando can be accomplished by making an attenuation sweep (to a still audible volume) end before the rest of the note.

ALEN, like NLEN for a frequency sweep, is the number of steps in an attenuation sweep and can take on values from 0 to 15. However, since a "step" consists of a tone (which may be frequency swept) played at a fixed attenuation level, a sweep of 1 step doesn't make sense. ALEN values, then, should range from 2 to 15. An ALEN value of 0 causes a sweep of sixteen steps (NOTE: ATN_SWEEP "wraps around" at 0 and 15, i.e., subtracting 1 from 0 results in 15, and adding 1 to 15 results in 0).

The duration of an attenuation sweep can be calculated as follows:

duration entire sweep = [((ALEN -1) * APS) + initial APSV] * 16.7ms
duration 1st step = initial APSV * 16.7ms
duration all others = APS * 16.7ms

Attenuation sweep parameter ranges:

ALEN: 4 bit number of steps in an attenuation sweep; can take values from 2 to 15 (0 => 16 steps).

ASTEP: ASTEP is the 4 bit signed (two's complement) size of a step in an attenuation sweep; can take values from 1 to 7, -1 to -8.

APS: 4 bit attenuation prescaler, used with ALEN to determine the length of a frequency sweep: 0 to 15 (0 => 16).

APSV: 4 bit temporary storage location for APS; this variable gets decremented every VDP interrupt, and is reloaded from APS: 0 to 15 (0 => 16)

Descriptions of each of the six note types follow:

* Rests

See Figure 4.

byte 0: B5 set indicates a rest, to be played on CH# in B7 - B6
duration = (B4 - B0) * 16.7ms, can take values 1 to 31

* Type 0: Fixed frequency, fixed attenuation

See Figure 4.

byte 0: header, CH# in B7 - B6, note type = 0 in B1 - B0
byte 1: least significant 8 bits of the 10 bit frequency data (constant)
byte 2: MSN = 4 bit ATN data (constant throughout the note)
LSN = 0 0 F0 F1, the top 2 bits of the frequency data (constant)
byte 3: NLEN, duration of the note = NLEN * 16.7ms

* Type 1: Swept frequency, fixed attenuation

See Figure 5.

byte 0: header, CH# in B7 - B6, note type = 1 in B1 - B0
byte 1: least significant 8 bits of the initial 10 bit frequency data
byte 2: MSN = 4 bit ATN data (constant throughout the note)
LSN = 0 0 F0 F1, the top 2 bits of the initial frequency data
byte 3: NLEN, number of steps in the frequency sweep, 1 to 255 (0 => 256)
byte 4: FPS : FPSV, prescaler reload value and initial FPSV
byte 5: FSTEP, sweep step size, 1 to 127, -1 to -128

* Type 2: Fixed frequency, swept attenuation

See Figure 6.

byte 0: header, CH# in B7 - B6, note type = 2 in B1 - B0
byte 1: least significant 8 bits of the 10 bit frequency data (constant)

byte 2: MSN = 4 bit ATN data (initial value)
LSN = 0 0 F0 F1, the top 2 bits of the frequency data (constant)
byte 3: NLEN, duration of the note = NLEN * 16.7ms
byte 4: ALEN ! ASTEP
ALEN = number of steps in the attenuation sweep
ASTEP = step size, 1 to 7, -1 to -8
byte 5: APS ! APSV, prescaler reload value and initial APSV, 1 to 15 (0 => 16)

* Type 3: Swept frequency, swept attenuation

See Figure 7.

byte 0: header, CH# in B7 - B6, note type = 3 in B1 - B0
byte 1: least significant 8 bits of the initial 10 bit frequency data
byte 2: MSN = 4 bit ATN data (initial value)
LSN = 0 0 F0 F1, the top 2 bits of the initial frequency data
byte 3: NLEN, number of steps in the frequency sweep, 2 to 255 (0 => 256)
byte 4: FPS ! FPSV, prescaler reload value and initial FPSV, 0 - 15 (0 => 16)
byte 5: FSTEP, sweep step size, 1 to 127, -1 to -128
byte 6: ALEN ! ASTEP
ALEN = number of steps in the attenuation sweep
ASTEP = step size, 1 to 7, -1 to -8
byte 7: APS ! APSV, prescaler reload value and initial APSV, 1 to 15 (0 => 16)

* Noise notes: special case Type 2 notes

See Figure 8.

Noise notes are notes that are played on the sound chip noise generator (CH#0). They are stored in ROM as a special case of a Type 2 note, fixed frequency and swept attenuation. They consist of white noise with superimposed attenuation decay, which creates a percussive effect, such as a snare drum note.

Instead of frequency information, noise notes are stored with three bits of noise control data: FB NF0 NF1 (see TI data sheets 76489), which remain constant throughout the note. Experimentation with various values can result in credible percussion effects.

NLEN, as is the case for a regular Type 2 note, directly determines the duration of a noise note.

The sound chip noise generator is unlike the other generators in that sending it redundant data (i.e., the same data that it has stored in its internal registers) has an audible effect on its output. In particular, whenever control data, redundant or not, is sent to the noise generator, its internal shift register is reset, causing a short pop or click to be heard. This isn't annoying on an occasional basis, or when a new noise starts, but remember: PLAY_SONGS is sending data to all four channels every 16.7ms. This would cause a noticeable lack of "whiteness" in the noise generator's output.

PLAY_SONGS avoids this problem by referencing a dedicated CART RAM location, SAVE_CTRL (see Figure 9) each time before it sends control data to the noise generator. SAVE_CTRL contains the control data that was output to the noise generator the last time through PLAY_SONGS. If the noise control data in the pointed to song data area = SAVE_CTRL, PLAY_SONGS doesn't send it out again. If there is new data to be sent, that data is output and SAVE_CTRL is updated.

byte 0: header, CH#0 in B7 - B6, note type = 2 in B1 - B0
byte 1: MSN = 4 bit ATN data (initial value)
LSN = 0 FB NF0 NF1, noise control data
byte 2: NLEN, duration of note: NLEN * 16.7ms
byte 3: ALEN ! ASTEP
ALEN = number of steps in the attenuation sweep
ASTEP = step size, 1 to 7, -1 to -8
byte 4: APS ! APSV, prescaler reload value and initial APSV, 1 to 15 (0 => 16)

OPERATING SYSTEM ROUTINES

INIT_SOUND

Contains ENTRY POINT: ALL_OFF

INIT_SOUND, usually called right after power on, turns off the sound generators, initializes the CART RAM locations to be used as song data areas, and sets up the four channel data area pointers. Specifically, it:

1) directly turns off all four sound generators.
2) initializes PTR_TO_LST_OF_SND_ADDRS, a dedicated 16 bit CPU RAM pointer which other sound routines expect to contain the base address of a list in CART ROM (called LST_OF_SND_ADDRS) of the starting addresses of each song's data area and note list. The address of LST_OF_SND_ADDRS is passed to INIT_SOUND in HL.
3) stores the sound-inactive code (0FFH) into byte 0 of n song data areas. n is passed in B and = the total number of song data areas used by the game.
4) stores an end of data area code (00) following the last data area.
5) sets the four pointers to the data areas for the songs to be played on each channel, PTR_TO_S_ON_x (x = 0-3), to a dummy inactive area (DUM_AREA, which is actually a single 0FFH byte within INIT_SOU).
6) sets SAVE_CTRL to an initial value of 0FFH

INPUT: n
TYPE: 8 bit constant
PASSED: in B
DESCRIPTION: number of song data area used by the game

INPUT: LST_OF_SND_ADDRS
TYPE: 16 bit address
PASSED: in HL
DESCRIPTION: LST_OF_SND_ADDRS is the base address of a list of the starting addresses of each song's data area and note list.

OUTPUT: 1) turns off all sound generators
2) initializes PTR_TO_LST_OF_SND_ADDRS
3) writes inactive code to byte 0 of n song data areas
4) stores 00 at end of song data areas
5) sets the 4 channel song pointers to the inactive DUM_AREA
6) sets SAVE_CTRL to 0FFH

ALL_OFF

ALL_OFF directly turns off all four sound generators, but does nothing to any song data areas or the 4 channel data pointers.

INPUT: none

OUTPUT: turns off all sound generators

JUKE_BOX

JUKE_BOX is called to start a song. Using a song number passed in B, JUKE_BOX loads the data for the song's first note into the appropriate song data area (the address of the area is found by using the song number as an index into the LST_OF_SND_ADDRS table). It also formats the data area's header and sets up the next note pointer. If the song is a special sound effect, its next note pointer is set to the address of the special effect routine. The next time PLAY_SONGS is called, that song's first note will be processed (thereby truncating whatever song had been "playing" in that data area), and the song will have started.

Since starting a new song may have altered the priority structure within the song data areas, JUKE_BOX also calls UP_CH_DATA_PTRS to modify the channel data pointers accordingly.

If JUKE_BOX is called with a song number of a song already in progress, it returns immediately (i.e., it doesn't restart the song).

INPUT: song number to be started
TYPE: 8 bit constant, 1 to 61
PASSED: in B

CALLS: PT_IX_TO_SxDATA, LOAD_NEXT_NOTE, UP_CH_DATA_PTRS

OUTPUT:
1) moves the song's first note data to the appropriate song data area
1) formats byte 0 header of the song's data area
2) points next note pointer in data area (bytes 1&2) to address of first note in song, or address of special sound effect routine
3) updates the channel data pointers on basis of song priorities

SND_MANAGER

SND_MANAGER should be called every VDP interrupt (every 16.7 ms). It assumes that the song data areas are stored contiguously in a data block beginning with the data area assigned to song number one. For each data area, SND_MANAGER, or routines which it calls, processes the appropriate timer and sweep counters and modifies the frequency and attenuation data accordingly. If the data area is assigned to a special effect, SND_MANAGER simply calls that effect, and doesn't modify any data. When a note is finished, SND_MANAGER, using the data area's next note pointer, moves data for the next note of the song into the area and fills in keys bytes within the area to allow proper processing of the data area by the sweep routines it calls (FREQ_SWEEP and ATN_SWEEP). (SND_MANAGER considers a note finished when its frequency duration timers have timed out; see the descriptions of the FREQ_SWEEP and ATN_SWEEP routines) A special effect is responsible for deciding when its over and initiating the next note in the song.

After the operations upon a data area have been performed, the channel data area pointers (PTR_TO_S_ON_x) may be updated (see description of UP_CH_DATA_PTRS in "UTILITIES" section).

If SND_MANAGER reads a header byte (in CART ROM) that has bits 3&4 set, indicating repeat song, it will start the song again by reloading the first note in the song, using the SONGNO portion (B5-B0) of byte 0 in the song's data and the LST_OF_SND_ADDRS to find it.

SND_MANAGER does not output the modified frequency and attenuation data. PLAY_SONGS is usually called just before SND_MANAGER to do this.

Special codes in byte 0 of the song data area indicate:

0FFH	-	data area inactive, do no processing, do not modify channel data area pointer
B5-B0 = 62	-	a special effect is to be played; SND_MANAGER calls the effect routine
00H	-	end of song data areas (SND_MANAGER processes data areas until it sees 0 in byte 0)

NOTE: Song number 1 MUST use the first data area in the block of song data areas.

INPUT: none

CALLS: PROCESS_DATA_AREA, PT_IX_TO_SxDATA

OUTPUT:
1) decrements song duration and sweep timers
2) modifies swept frequency and attenuation values
3) calls special effects routines where necessary
4) restarts the song if indicated
5) may update the channel data area pointers (PTR_TO_S_ON_x)

PLAYSONGS

PLAY_SONGS takes the frequency and attenuation data pointed to by the four channel data area pointers (PTR_TO_S_ON_x) and outputs it to the four sound chip generators. Action is taken on the basis of the each data area's byte 0:

1) If the pointed to data area is active, the frequency and attenuation data are sent to the channel indicated by B7-B6 (CH#) of byte 0 of the pointed to data area.
2) If byte 0 is 0FFH (inactive), the channel to which that pointer is dedicated is sent the OFF attenuation code.
3) If CH# = 0 (noise), the attenuation data is output. If there is no new noise control data to be output (determined by checking dedicated CART RAM location SAVE_CTRL), no control data is sent out. Otherwise, the new control data is output and SAVE_CTRL is updated.

INPUT: none

OUTPUT: through SOUND_PORT,
1) current freq and atn data is output to each tone generator, if song/effect on that channel is active
2) noise generator is sent current atn data, and control data, if new
3) modifies SAVE_CTRL if necessary

FREQ_SWEEP

FREQ_SWEEP is used by SND_MANAGER and special effects routines to create frequency sweeps. It operates upon frequency data stored within a song data area, and is normally called (by SND_MANAGER or a special effect routine) once every VDP interrupt (16.7ms). The start of the data area (address of byte 0) is passed in IX.

FREQ_SWEEP assumes data has been stored as follows (names which may be used to describe the various bytes or byte segments within the data area are indicated; see Figure 1):

byte 3: the least significant 8 bits of that note's frequency (F2 - F9)

byte 4: top 2 bits of that note's frequency: B1 = F0, B2 = F1

byte 5: NLEN - determines the note's duration:
1) if frequency is to be swept, NLEN = number of steps in the sweep: 2 to 255 (0 => 256)
2) if fixed frequency, NLEN * 16.7 ms = duration of the note: 1 to 255 (0 => 256)

byte 6: FPS ! FPSV - frequency sweep duration prescaler:
FPS = prescaler reload value: 0 to 15 (0 => 16)
FPSV = temp storage nibble for FPS: init ROM value, 0 to 15 (0 => 16)
duration of sweep (& note) = [((NLEN-1) * FPS) + initial FPSV] * 16.7ms
duration 1st step = initial FPSV * 16.7ms
duration all other steps = FPS * 16.7ms

byte 7: FSTEP - frequency sweep step size: signed 8 bit number, two's complement: 1 to 127, -1 to -128
if FSTEP = 00, frequency is not to be swept, but NLEN is decremented each time called

Parameter limitations:
1) In a frequency sweep, a "step" consists of a single fixed frequency tone; therefore, the minimum number of steps a frequency sweep can have is two (otherwise the frequency wouldn't have "swept").
2) If a note is to be frequency swept, FSTEP must not be 0.
3) The minimum length fixed frequency note has NLEN = 1.
4) Maximum NLEN 0, which is equivalent to 256.

FREQ_SWEEP returns with the Z flag SET if the note (swept or fixed) is over, RESET if the note is not over. (PROCESS_DATA_AREA decides that a note is over when FREQ_SWEEP returns with the Z flag set)

INPUT: 16 bit address of a song data area in CPU RAM
PASSED: in IX
DESCRIPTION: FREQ_SWEEP operates upon frequency data within this song data area

OUTPUT: 1) duration and sweep counters are decremented
2) freq data in bytes 3&4 is modified if note is freq swept
3) returns with Z flag SET if note over, RESET if note not over

ATN_SWEEP

ATN_SWEEP is used to create attenuation sweeps. It operates upon attenuation data stored within a song data area, and is normally called (by PROCESS_DATA_AREA or a special effect routine) once every VDP interrupt (16.7ms). The start of the data area (address of byte 0) is passed in IX.

ATN_SWEEP assumes data has been stored as follows (see Figure 1):

byte 4: ATN - the MSN = 4 bit attenuation

byte 8: ALEN ! ASTEP - no-sweep code or sweep length and step size:
1) if byte 8 = 00, ATN is not to be swept and counters aren't changed
2) if byte 8 non zero, attenuation is to be swept:
   a) ALEN = number of steps in the sweep: 1 to 15 (0 => 16)
   b) ASTEP = sweep step size: 1 to 7, -1 to -8 (signed, 4 bit two's complement)

byte 9: APS ! APSV - attenuation sweep duration prescaler:
1) if attenuation is not swept, byte 9 is not used by ATN_SWEEP
2) if attenuation is to be swept:
APS = prescaler reload value: 1 to 15 (0 => 16)
APSV = temp storage nibble for APS: init ROM value, 1 to 15 (0 => 16)
duration of swept attenuation part of note =
[((ALEN - 1) * APS) + initial APSV] * 16.7 ms
  duration 1st step = initial APSV * 16.7ms
  duration all other steps = APS * 16.7ms

Parameter limitations:
1) In an attenuation sweep, a "step" consists of a tone (swept or not) played at a fixed attenuation level; so, the minimum number of steps an attenuation sweep can have is two (otherwise the attenuation wouldn't have "swept"). Therefore, the minimum ALEN value is 2 (0 => 16)
2) If a note is to be attenuation swept, byte 8 must not be 00.
3) The absolute value of ASTEP must be > = 1.

If byte 8 is 00, ATN_SWEEP returns immediately with Z flag SET (the sweep is over or the note was never swept), and doesn't modify any counters. When a sweep finishes, ATN_SWEEP sets byte 8 to 00 and returns with the Z flag SET. If a sweep is in progress, ATN_SWEEP returns with the Z flag RESET. (NOTE: PROCESS_DATA_AREA decides that a note is over when FREQ_SWEEP returns with Z set: the length of a note has nothing to do with when its attn sweep is over)

INPUT: 16 bit address of a song data area in CPU RAM
PASSED: in IX
DESCRIPTION: ATN_SWEEP operates upon frequency data within this song data area

OUTPUT: 1) duration and sweep counters are decremented if sweep in progress
2) atn data in byte 4 is modified if note is atn swept
3) RETs C SET, byte 8 = 0 if sweep is over or note was never swept
   RETs C RESET if sweep in progress

PROCESS_DATA_AREA

PROCESS_DATA_AREA is called by SND_MANAGER. For an active data area (address of byte 0 passed in IX), PROCESS_DATA_AREA modifies the timers, sweep counters, frequency, and attenuation data by calling FREQ_SWEEP and ATN_SWEEP. If a note finishes during the current pass through PROCESS_DATA_AREA, the next note in the song is examined and its data is loaded into the data area (calls LOAD_NEXT_NOTE). Then, in order to maintain the song data area priority structure, the CH# : SONGNO of the newly loaded note is compared to the CH# : SONGNO of the previous note: if there is a difference, UP_CH_DATA_PTRS is called to adjust the channel data area pointers in response to the change caused by loading the next note.

If the data area is being used by a special sound effect, PROCESS_DATA_AREA calls the sound effect routine whose address is stored in bytes 1&2 of the data area (the actual address called is routine + 7: see discussion of special sound effects).

If the data area is inactive, PROCESS_DATA_AREA returns immediately (no processing occurs).

INPUT: address of byte 0 of a song data area
PASSED: in IX

CALLS: ATN_SWEEP, FREQ_SWEEP, LOAD_NEXT_NOTE, UP_CH_DATA_PTRS, AREA_SONG_IS

OUTPUT:
1) if active, modifies song data area's timer, freq, and atn data
2) loads the next note's data when a note is finished
3) if special sound effect routine using data area, calls it
4) when necessary, updates the channel data area pointers

LOAD_NEXT_NOTE

Called by PROCESS_DATA_AREA and JUKE_BOX, LOAD_NEXT_NOTE examines the next note to be played in a data area (address byte 0 passed in IX) and moves its data into the area. It fills in bytes (e.g., to indicate swept or not swept) where appropriate, based upon note type. If the next "note" is a special sound effect, its address is saved in bytes 1&2 and the address of the routine + 0 is called, with the address of the note to follow the effect passed in HL and SONGNO passed in A. This will cause the special effect routine to save both these values. Then, the special effect routine + 7 is called, which allows the routine to initialize the song data area for the first pass through PLAY_SONGS. (see discussion of special sound effects)

Prior to moving the next note data, LOAD_NEXT_NOTE saves the data area's byte 0 (CH# ! SONGONO) and stores the song inactive code (0FFH) there. The last thing LOAD_NEXT_NOTE does is restore byte 0, loading CH# with the CH# ! SONGNO of the new note (usually the same as the old note). If the new note is a special sound effect, 62 is returned as the SONGNO part of byte 0.

INPUT: address of byte 0 of a song data area
PASSED: in IX

OUTPUT:
1) sets up song data area with data from next note to be played
2) for next note = special sound effect, calls the effect twice, first with the address of the following note in the song and the song's SONGNO, and then once more to allow the effect to initialize the song data area
3) if next note is "normal", loads CH# ! SONGNO in byte 0 with CH# ! SONGNO of new note
4) returns with byte 0 = 0FFH if song over, SONGNO = 62 if next note is a sound effect

UTILITIES

The following are O/S utility routines, used by the main O/S sound programs, that may be of use to the cartridge programmer:

*** AREA_SONG_IS ***

The address of byte 0 of a song data area is passed in IX. The song number of the song using that area is returned in A (0FFH if inactive). If a special effect was using that area, 62 is returned in A and HL is returned with the address of the special sound effect routine.

*** UPATNCTRL ***

Perform single byte update of the snd chip noise control register or any attenuation register. IX is passed pointing to byte 0 of a song data area, MSN register C = formatted channel attenuation code.

*** UPFREQ ***

Perform double byte update of a sound chip frequency register. IX is passed pointing to byte0 of a song data area, MSN register D = formatted channel frequency code.

*** DECLSN ***

Without affecting the MSN, decrement the LSN of the byte pointed to by HL. HL remains the same.
RET with Z flag set if dec LSN results in 0, reset otherwise.
RET with C flag set if dec LSN results in -1, reset otherwise.

*** DECMSN ***

Without affecting the LSN, decrement the MSN of the byte pointed to by HL. HL remains the same.
RET with Z flag set if dec MSN results in 0, reset otherwise.
RET with C flag set if dec MSN results in -1, reset otherwise.

*** MSNTOLSN ***

Copy MSN of the byte pointed to by HL to the LSN of that byte. HL remains the same.

*** ADD816 ***

Adds 8 bit two's complement signed value passed in A to the 16 bit location pointed to by HL. Result is stored in the 16 bit location.

*** PT_IX_TO_SxDATA ***

A SONGNO is passed in B. PT_IX_TO_SxDATA returns with IX pointing to the song data area which is used by that SONGNO.

*** UP_CH_DATA_PTRS ***

UP_CH_DATA_PTRS adjusts each channel data pointer to point to the highest priority (ordinal last) song data area that uses that channel. It is called whenever a change has been made to a song data area that requires modification of the channel data pointers.

All 4 channel data pointers (PTR_TO_S_ON_x) are initially pointed to a dummy inactive area, DUM_AREA. Then, moving in order from the first data area to the last, CH# in byte 0 of each data area is examined, and the corresponding channel data pointer is pointed to that data area. Thus, by the time the routine is done, each channel data pointer is pointing to the last active data area that contains data to be sent to that channel. If none of the active data areas used a particular channel, then that channel remains pointing to DUM_AREA (and therefore its generator will be turned off next time through PLAY_SONGS).

*** LEAVE_EFFECT ***

LEAVE_EFFECT, called by a special sound effect routine when it's finished, restores the SONGNO of the song to which the effect belongs to B5 - B0 of byte 0 in the effect's data area, and loads bytes 1 & 2 with the address of the next note in the song. The address of the 1 byte SONGNO (saved by the effect when it was first called) is passed in DE. The 2 byte address of the next note in the song, also saved by the effect, is passed in HL. IX is assumed to be pointing to byte 0 of the data area to which the song number is to be restored. Bits 7 & 6 of the saved SONGNO byte are not stored into byte 0, and therefore may be used during the course of the effect to store any useful flag information.

SPECIAL SOUND EFFECTS

* Sound effects as notes within a song

Sounds which do not fit one of the six categories of "normal" musical notes can be created and played throughout the course of a song as "special effect" notes. Unlike normal musical notes, which are stored in ROM as tables of frequency/control and attenuation data, a special effect's data are determined algolrithmically by a custom routine written by the cartridge programmer. Special effect notes can also be used to generate sounds that could have been comprised of many normal notes, but which are more efficiently (in terms of ROM space used) computed by a short program.

These notes use the same song data area as the song within which they are contained, and they are stored in the song's ROM note list with a one byte header as are normal notes. However, the bytes following the ROM header do not contain data to be directly loaded into the song data area. The header (see Figure 2), which specifies the channel upon which to play the effect (which is usually the same as the channel used by the rest of the notes in the song), is followed by a two byte address of a routine written by the cartridge programmer which will be called every 16.7ms by PROCESS_DATA_AREA. When called, this special effect routine should compute data values and store them at the appropriate locations within the song data area. (In fact, many effect routines may call the O/S routines FREQ_SWEEP or ATN_SWEEP, which also require that data be ordered appropriately within a song data area) This computed data will then be output on the next pass through PLAY_SONGS (assuming that this song data area has the highest priority of any data area using the same channel).

Variables required by the effect which will not be output may be stored wherever the programmer desires. Free locations within the song's data area might as well be used for effect variable storage, since the entire ten byte area is reserved for the song anyway. If no free locations exist within a data area, which would be the case if an effect required both frequency and attenuation to be swept, the effect can store the remaining needed variables wherever convenient.

In order to interact properly with the O/S sound routines, each special effect routine must conform to a certain format. A description of that format, and how an effect interacts with the O/S routines, follows:

WHEN AN EFFECT BEGINS - When loading a new note, if LOAD_NEXT_NOTE sees that the note to be loaded is a special effect:

1) It stores in byte 0 of the song's data area the effect's CH# and a SONGNO of 62. SONGNO = 62 is used later by PROCESS_DATA_AREA to detect the fact that an effect is using the data area.

2) It then takes the address of the special effect routine (lets's call it SFX) from ROM and puts it into bytes 1&2 (NEXT_NOTE_PTR).

3) LOAD_NEXT_NOTE then calculates the ROM address of the header of the next note in the song, stores that address in HL, puts the song's SONGNO in A, and calls SFX + 0. In every special effect routine at SFX + 0, there MUST be the following code which saves the two passed values (see Figure 9):

```
SFX:    LD (SAVE_x_NNP),HL
        LD (SAVE_x_SONGNO),A
        RET
SFX+7:  code for sound effect starts here
        ...
```

where SAVE_x_NNP is a two byte location used by all the sound effect notes in the current song to save the address of the next note in the song, and SAVE_x_SONGNO is the address of a byte where the song number is saved. The programmer may put SAVE_x_NNP and SAVE_x_SONGNO wherever desired, including somewhere within the song data area.

Thus, calling SFX + 0 allows each effect routine to save the next note's address and the song's SONGNO.

1ST PASS THROUGH EFFECT - After calling SFX + 0, LOAD_NEXT_NOTE calls SFX + 7 for the first pass through the body of the routine. At this location, there should be code which initializes the appropriate bytes within the song data area, as the next pass through PLAY_SONGS, subject to the data area priority system, will output this initial data in normal fashion.

As will be seen below, this same location (SFX + 7) will be called every 16.7ms by PROCESS_DATA_AREA to modify the data within the area. Therefore, the code at SFX + 7 must know which pass is in effect, so that the song data area will be initialized only on the first pass. A convenient way of doing this is to test bit 7 of SAVE_x_SONGNO, the byte which contains the saved song number. On the 1st pass through the effect, bit 7 (and bit 6) will be zero, since the largest possible SONGNO (62) would not set this bit. If bit 7 is zero, then, code to initialize the data area can be executed and bit 7 reset to prevent re-initialization. I.e.,

```
SFX+7:    LD HL,SAVE_x_SONGNO
          BIT 7,(HL)
          JR NZ,NOT_PASS_1
            SET 7,(HL)          ;to prevent further passes thru inits
            ...                 ;initialize bytes within the data area here
            RET                 ;to LOAD_NEXT_NOTE
NOT_PASS_1: ....                ;code for pass 2 or greater starts here
```

PRIORITY UPDATE - After calling SFX + 0 and SFX + 7, LOAD_NEXT_NOTE will return to PROCESS_DATA_AREA, which checks to see if loading a new note has caused a change in either the channel used by the song (this happens with noise notes within a musical song) or the song number. If a change has occured, UP_CH_DATA_PTRS will be called, which updates the data pointers on the basis of priority within the block of song data areas (see description of this routine in the preceeding "UTILITIES" section). Since a special effect note will cause a change in the song number (from whatever it was to 62), UP_CH_DATA_PTRS will always be called whenever an effect note is loaded.

SECOND PASS OR GREATER - The next time PROCESS_DATA_AREA is called (from SND_MANAGER), which will be 16.7ms after PLAY_SONGS has sent out the effect's initial data, it will detect the fact that an effect is using the song data area (by seeing a SONGNO of 62) and will JUMP to SFX + 7, rather than calling the frequency and attenuation sweep routines as it would for a normal note. This will result in the first pass through the part of the body of the effect routine that actually does computation and adjusts the data values within the data area. When the effect routine has completed its modifications to the data area and performs a RET, control is transferred back to SND_MANAGER, which then moves on to the next song data area to be processed.

This process will be repeated every 16.7ms until the effect routine itself decides that it's over and takes action to load the next note in the song.

WHEN AN EFFECT IS OVER - Prior to performing a RET, the effect routine must decide whether the effect note has finished. If it has, NEXT_NOTE_PTR within the data area must be set to the address of the next note in the song and SONGNO must be restored to byte 0. This can be done by calling the O/S routine LEAVE_EFFECT which does this. The address of SAVE_x_NNP must be passed in HL and the address of SAVE_x_SONGNO must be passed in DE. Finally, the effect should JUMP to EFXOVER, a location within PROCESS_DATA_AREA which would normally be reached once a note is over. The code there takes care of loading the next note in the song. Thus, the final code of each effect routine will look as follows:

```
RET if effect not over
  LD HL,(SAVE_x_NNP)        ;HL := addr next note in song
  LD DE,SAVE_x_SONGNO       ;DE := addr saved song number
  CALL LEAVE_EFFECT         ;to restore them to bytes 0 - 2 in data area
  JP EFXOVER                ;in PROCESS_DATA_AREA to load song's next note
```

The entire above described sequence is summarized in Figure 9.

* A sound effect as a single sound

A stand alone sound effect can be implemented within the previously mentioned structures simply by creating a single note song. The single note is the effect

and would be followed by an end of song code (or repeat code if you wish the effect to go on forever).

Many stand alone effects may want to use more than one tone generator channel: e.g., a special laser zap that momentarily requires all three tone generators, or, as is often the case, a white noise effect of particular character that requires the noise channel shift rate to be modified by channel three (see TI data sheets). In these cases, the effect's routine will have to modify data in several data areas whenever called. The song data areas used by such effects are subject to the normal priority structure. E.g., if you wish a two channel effect to temporarily overwrite the harmony and bass lines of a repeating song, the effect must have been assigned two data areas of higher priority (ordinally later in the block of song data areas). If it is not necessary to maintain any underlying songs, an effect can share data areas to conserve RAM space, with the understanding that, as usual, songs or sounds that share the same song data area truncate each other. A multi-channel effect (a chord note, say) may be included as a note within a song, but, again, the song data area priority structure determines what will finally be heard.

Providing for a typical game's sound generation needs might require eight song data areas: four for an underlying, repeating song(s) (three areas for the three tone generators and one for the noise generator used for percussion notes), and four for higher priority, occasional sound effects (which would temporarily overwrite the repeating songs, but truncate each other).

* Pseudo code listings of main routines

The following two pages contain pseudo code descriptions of most of the O/S sound routines. Some computational details are not shown, but all jumps, calls, returns, pushes and pops are listed.

Terminology:
":=" is used as the assignment statement, and "(xx)" means the contents of the memory location pointed to by xx, where "xx" is HL, IX, etc.

The structure of each description is as follows:

```
*** name of routine ***
the value expected for passed parameters (if any)

pseudo code description
of the routine, uninterrupted
by blank lines
RET
```

```
zzz JUKE_BOX zzz
B = SONGNO to be started

PUSH BC
CALL PT_IX_TO_SxDATA (IX set)
POP BC
RET if song in progress
  byte 0 := SONGNO
  set NEXT_NOTE_PTR to 1st note in song
  CALL LOAD_NEXT_NOTE
  CALL UP_CH_DATA_PTRS
  RET

zzz PLAY_SONGS zzz

set A for CH1 OFF code
set MSN C for CH1 attenuation
set MSN D for CH1 frequency
IX := (PTR_TO_S_ON_1)
(i.e., pt IX to byte 0 data area of song for CH1)
CALL TONE_OUT
set A for CH2 OFF code
set MSN C for CH2 attenuation
set MSN D for CH2 frequency
IX := (PTR_TO_S_ON_2)
(i.e., pt IX to byte 0 data area of song for CH2)
CALL TONE_OUT
set A for CH3 OFF code
set MSN C for CH3 attenuation
set MSN D for CH3 frequency
IX := (PTR_TO_S_ON_3)
(i.e., pt IX to byte 0 data area of song for CH3)
CALL TONE_OUT
set A for CH0 OFF code
set MSN C for CH0 attenuation
IX := (PTR_TO_S_ON_0)
(i.e., pt IX to byte 0 data area of song for CH0)
IF area INACTIVE
  turn off CH0
ELSE
  CALL UPATNCTRL (send current atn)
  set LSN A for current ctrl data
  IF current ctrl data diff from last
    reload SAVE_CTRL
    CALL UPATNCTRL (send new ctrl)
  ENDIF
ENDIF
RET

zzz SND_MANAGER zzz

CALL PT_IX_TO_SxDATA, song 01
LOOP
  RET if end of song data areas
    CALL PROCESS_DATA_AREA
    pt IX to byte 0 next song data area
REPEAT LOOP

zzz PROCESS_DATA_AREA zzz
IX = addr byte 0 of a song data area

CALL AREA_SONG_IS
RET if area INACTIVE
  IF SONGNO = 62
    JP SFX+7 (RET from SFX)
  ENDIF
    CALL ATN_SWEEP
    CALL FREQ_SWEEP
    IF note over
EFXOVER: PUSH CH0 ! SONGNO note just over
      CALL LOAD_NEXT_NOTE
      POP CH0 ! SONGNO note just over
      IF CH0 ! SONGNO newly loaded note not =
      CH0 ! SONGNO note just over
        CALL UP_CH_DATA_PTRS
      ENDIF
    ENDIF
    RET

zzz LOAD_NEXT_NOTE zzz
IX = addr byte 0 of a song data area
byte 0 = CH0 (or 00) ! SONGNO
(SFX = addr of a special effect note's routine)

PUSH 0 0 ! SONGNO (CH0 not pushed)
deactivate area (byte 0 := FF)
A := (NEXT_NOTE_PTR) (header new ROM note)

CASE header type of
1)rest:
  PUSH header of new ROM note
  set NEXT_NOTE_PTR for next note in song
  (i.e., the note after this new note)
  set bytes in song data area:
    ATN := off
    NLEN := 5 bit rest duration
    FSTEP := 0 (i.e., no freq sweep)
    ASTEP := 0 (no atn sweep)
  JP MODB0
2)end of song:
  IF end repeat
    POP BC (B := SONGNO)
    CALL JUKE_BOX to reload 1st note of this song
    RET (to PROCESS_DATA_AREA)
  ENDIF
    PUSH inactive code
    JP MODB0
3)special effect:
  POP IY (IY := SONGNO)
  PUSH IY to put SONGNO back on stack
  PUSH header new ROM note
  set NEXT_NOTE_PTR, bytes 1&2, to SFX
  (address special effect routine)
  set DE to SFX
  HL := addr next note in song
  (i.e, the note after this new effect note)
  PUSH IY, POP AF (A := SONGNO)
  PUSH DE, POP IY (IY := SFX)
  DE := PASS1, PUSH DE
  JP (IY), i.e., "CALL (IY)", RET to PASS1
  (SFX saves SONGNO & addr next note)
  PASS1: IY := IY + 7
  DE := MODB0, PUSH DE
  JP (IY), i.e., "CALL (IY+7)", RET to MODB0
  (SFX+7 loads initial effect data)
4)normal note:
  CASE note type
    0)NEXT_NOTE_PTR := adr song's nxt note
      move 3 bytes ROM note data to RAM
      FSTEP := 0 (no freq sweep)
      ASTEP := 0 (no atn sweep)
      JR MODB0
    1)NEXT_NOTE_PTR := adr song's nxt note
      move 5 bytes ROM note data to RAM
      ASTEP := 0 (no atn sweep)
    2)NEXT_NOTE_PTR := adr song's nxt note
      move 5 bytes ROM note data to RAM
      FSTEP := 0 (no freq sweep)
      JR MODB0
    3)NEXT_NOTE_PTR := adr song's nxt note
      move 7 bytes ROM note data to RAM
  ENDCASE
ENDCASE

MODB0: PUSH IX
POP HL to point to byte 0
POP AF (A := header new note)
POP BC (B := SONGNO)
RET if header is inactive (i.e., song is over)
  IF header is for a special effect
    B := 62, the SONGNO for all effect notes
  ENDIF
  byte 0 := CH0 (from header new note) ! SONGNO (from B)
  RET
```

```
*** INIT_SOUND ***
HL = addr of LST_OF_SND_ADDRS
B = number of song data areas used by game

set RAM word PTR_TO_LST_OF_SND_ADDRS to value passed in HL
pt HL to byte 0 in 1st song data area
B1: (HL) := inactive code (0FFH)
HL := HL + 10
DJNZ B1 (set B areas inactive)
(HL) := end of data area code (0)
load all 4 channel data area pointers with the
addr of a dummy inactive area (DUMAREA)
SAVE_CTRL := 0FFH
ALL_OFF: turn off all 4 generators directly
RET

DUMAREA DEFB 0FFH


  *** UP_CH_DATA_PTRS ***

PUSH IX to save it
set all 4 ch data ptrs to a dummy, inactive area
CALL PT_IX_TO_SxDATA, song 0 1
LOOP
  IF byte 0 indicates the end of the song data areas JR DONE
    IF byte 0 indicates an active area
      set HL to address of this area's channel data pointer
      (i.e., HL := addr PTR_TO_S_ON_0 + (CH# this area  x 2))
      PUSH IX
      POP DE (DE := addr byte 0 this area)
      (HL) := E, (HL+1) := D
    ENDIF
    IX := IX + 10
  ENDIF
REPEAT LOOP
DONE  POP IX to restore it
RET
```

```
*** FREQ_SWEEP ***
IX = addr byte 0 of a song data area

IF FSTEP = 0 note is not to be swept
  A := NLEN
  DEC A
  RET Z (leave if note over, Z flag SET)
    NLEN := A (store decremented NLEN)
    RET (note not over, Z flag RESET)
ENDIF
PUSH IX, POP HL (pt HL to byte 0)
HL := HL + offset within data area of FPSV
CALL DECLSN to decrement FPSV
IF Z flag SET, FPSV has timed out
  CALL MSNTOLSN to reload FPSV
  A := NLEN
  DEC A
  RET Z (leave if sweep over with Z flag SET)
    NLEN := A (store decremented NLEN)
    point HL to FREQ
    A := FSTEP
    CALL ADD816 to add FSTEP to FREQ
    RESET bit 7 in hi byte FREQ
    (in case of overflow from addition)
    OR 0FFH to RESET Z flag
ENDIF
RET


*** ATN_SWEEP ***
IX = addr byte 0 of a song data area

RET with Z flag SET if byte 8 = 0
(i.e., note atn not to be swept)
  PUSH IX, POP HL (pt HL to byte 0)
  HL := HL + offset within data area of APSV
  CALL DECLSN to decrement APSV
  IF Z flag SET, APSV has timed out
    CALL MSNTOLSN to reload APSV
    pt HL to ALEN (# of steps in atn sweep)
    CALL DECLSN to decrement ALEN
    IF Z flag RESET, sweep not over yet
      ATN := ATN + ASTEP
      (4 bit add, overflow ignored)
      OR 0FFH to RESET Z flag
    ELSE Z flag is SET (sweep is over)
      byte 8 := 0 to indicate sweep over
    ENDIF
  ENDIF
RET
```

```
*** TONE_OUT ***

IX = (PTR_TO_S_ON_x), i.e., IX pts to byte 0 data area of song for CHx
A set for CHx OFF code
MSN C set for CHx attenuation
MSN D set for CHx frequency

IF area INACTIVE
  turn off CHx
ELSE
  CALL UPATNCTRL (send out attenuation)
  CALL UPFREQ (send out frequency)
ENDIF
RET


*** PT_IX_TO_SxDATA ***
B = a song number

HL := addr of LST_OF_SND_ADDRS
HL := HL - 2
BC := 4 * SONGNO
HL := HL + BC (i.e., HL now points to SxDATA's entry in LST_OF_SND_ADDRS)
E := (HL), D := (HL+1)
PUSH DE
POP IX (IX := addr byte 0 of this song's data area)
RET
```

SxDATA

FIGURE 1

Description: Storage area for the various timers and output data for song number x. The song data areas MUST be stored in a contiguous block of CPU RAM and the data area used by song number one MUST be the first data area in the block. Song data area storage is allocated according to addresses stored in LST_OF_SND_ADDRS, a table stored in CART ROM.

Byte 0 of each data area, in addition to giving CH# and song number, can indicate two special conditions:
- byte 0 = 0FFH: song(s) using this data area are inactive
- byte 0 = 00H: indicates end of song data areas

If SONGNO = 62, the address of a special sound effect routine is stored in bytes 1 and 2.

Length in bytes: 10
Location: CPU RAM
Beginning address: pointed to by a 16 bit entry in LST_OF_SND_ADDRS

Offset	Contents B7 B6 B5 B4 B3 B2 B1 B0	Description
0	CH# | SONGNO	B7 - B6: song channel number, 0 to 3 B5 - B0: song number, 1 to 61 SONGNO = 62, snd effect adr in next 2 bytes
1	the LSB of an address...	usually, the addr of the next note in song; if SONGNO = 0FEH, this is the LSB of the addr of the special sound.effect routine
2	the MSB of an address...	usually, the addr of the next note in song; if SONGNO = 0FEH, this is the MSB of the addr of the special sound effect routine
3	F2 F3 F4 F5 F6 F7 F8 F9	bottom 8 bits of 10 bit freq data
4	ATN|CTRL or ATN|0 0 F0 F1	if CH# = 0 (noise): ATN | 0 FB SHIFT if CH# = 1 - 3 (tone): MSN = 4 bit ATN, LSN = top 2 bits freq (0 0 F0 F1)
5	NLEN	determines duration of note: if freq swept, = # of steps in the sweep if not, NLEN * 16.7ms = duration of note
6	FPS | FPSV	freq sweep duration prescaler: FPS = prescaler reload value FPSV = temp FPS variable storage
7	FSTEP	freq sweep step size: 1 to 127, -1 to -128 if FSTEP = 0, freq is not to be swept

8	ASTEP | ALEN	ALEN = # steps in atnswp: 2 - 15 (0 => 16)
		ASTEP = step size: 1 to 7, -1 to -8
		if whole byte = 00, atn not to be swept

9	APS | APSV	atn duration prescaler: APS = prescaler reload value APSV = temp APS variable storage

DURATIONS:

```
fixed frequency =       NLEN * 16.7ms
frequency sweep =       [((NLEN - 1) * FPS) + initial FPSV] * 16.7ms
duration 1st step =     initial FPSV * 16.7ms
duration all others =   FPS
  FPS: 0 to 15 (0 => 16)
  FPSV: 0 to 15 (0 => 16)
  NLEN: 0 to 255 (0 => 256)
attenuation sweep =     [((ALEN - 1) * APS) + initial APSV] * 16.7ms
duration 1st step =     initial APSV * 16.7ms
duration all others =   APS
  APS: 0 to 15 (0 => 16)
  APSV: 0 to 15 (0 => 16)
  ALEN: 0 to 15 (0 => 16)
```

Note Header

FIGURE 2

Length in bytes: 1
Location: begins each block of 1 to 10 bytes of note data in CART ROM

Offset	Contents B7 B6 B5 B4 B3 B2 B1 B0	Description
		*** REST
0	CH# 1 duration	if B7 = 1, header describes a rest: B7 -B6 = channel number, 0 - 3 B4 - B0 = duration, 1 to 30
		or, if B5 = 0, header preceeds note data or is special indicator:
		*** NOTE
0	CH# 0 0 0 0 type	note data follows: B7 - B6 = channel number, 0 - 3 B1 - B0 = note type, 0 -3.
		or
		*** END OF SONG / REPEAT SONG
0	CH# 0 1 R 0 0 0	if B4 = 1, end of song on channel in B7-B6: if B3 = 1, repeat song forever if B3 = 0, don't repeat
		or
		*** SPECIAL EFFECT
0	CH# 0 0 0 1 0 0	this note is to be "played" by a special sound effect routine whose addr is contained in the following 2 bytes

REST DURATION = duration * 16.7ms
duration: 1 to 31

FIGURE 3

LST_OF_SND_ADDRS

Description: LST_OF_SND_ADDRS is a list of the starting addresses of each song's data area and note list. They are used by JUKE_BOX as source (note list) and destination (song data area) pointers. Each song's entries are stored as follows:

Byte 1: LSB of the address of the start of that song's note list
Byte 2: MSB of the address of the start of that song's note list
Byte 3: LSB of the address of the start of that song's data area
Byte 4: MSB of the address of the start of that song's data area

The beginning address of LST_OF_SND_ADDRS is stored in a dedicated CPU RAM 16 bit word, PTR_TO_LST_OF_SND_ADDRS (xxxxH), allowing the cartridge programmer to place LST_OF_SND_ADDRS wherever desired.

NOTE: In other data structures, six bits are allocated for the song number (SONGNO). However, song numbers 0, 62, and 63 are used as special indicators, leaving song numbers 1 - 61 available. Therefore, the first entry in LST_OF_SND_ADDRS is for song number 1.

Length in bytes: 4 * total number of songs
Location: CART ROM
Beginning address: pointed to by CPU RAM word $LST_OF_SND_ADDRS (xxxxH)

Offset	Contents B7 B6 B5 B4 B3 B2 B1 B0	Description
0	LSB	of starting adr of the note list for song number 1
1	MSB	of starting adr of the note list for song number 1
2	LSB	of starting adr of the data area for song number 1
3	MSB	of starting adr of the data area for song number 1
4	LSB	of starting adr of the note list for song number 2
etc...		
4 * n	MSB	of starting adr of the data area for song number n (n = total number of songs)

FIGURE 4

Rests

Length in bytes: 2
Location: CART ROM
Beginning address: pointed to by bytes 1&2 in that song's data area

Offset	Contents B7 B6 B5 B4 B3 B2 B1 B0	Description
0	CH# 1 duration	if B7 = 1, header describes a rest: B4 - B0 = duration, 1 to 30

REST DURATION = duration * 16.7ms
duration: 1 to 30

Note Type 0: fixed frequency, fixed attenuation

Length in bytes: 4
Location: CART ROM
Beginning address: pointed to by bytes 1&2 in that song's CPU RAM data area

Offset	Contents B7 B6 B5 B4 B3 B2 B1 B0	Description
0	CH# 0 0 0 0 0 0	header
1	F2 F3 F4 F5 F6 F7 F8 F9	least significant 8 bits of 10 bit frq data
2	ATN 0 0 F0 F1	ATN = 4 bit atn data, LSN = top 2 bits freq
3	NLEN	NLEN * 16.7ms = duration of note

NOTE DURATION = NLEN * 16.7ms
NLEN: 1 to 255 (0 => 256)

FIGURE 5

Note Type 1:
swept frequency, fixed attenuation

Length in bytes: 6
Location: CART ROM
Beginning address: pointed to by bytes 182 in that song's CPU RAM data area

Offset	Contents B7 B6 B5 B4 B3 B2 B1 B0	Description
0	CH# 0 0 0 0 0 1	header
1	F2 F3 F4 F5 F6 F7 F8 F9	least sig 8 bits of initial 10 bit frq.data
2	ATN 0 0 F0 F1	ATN = 4 bit atn data, LSN.= top 2 bits freq
3	NLEN	NLEN = number of steps in the sweep
4	FPS FPSV	freq sweep duration prescaler: FPS = prescaler reload value FPSV = initial FPSV
5	FSTEP	freq sweep step size: 1 to 127, -1 to -128

NOTE DURATION = [(NLEN - 1) * FPS) + initial FPSV] * 16.7ms
duration 1st step = initial FPSV * 16.7ms
duration all others = FPS
FPS: 0 to 15 (0 => 16)
FPSV: 0 to 15 (0 => 16)
NLEN: 0 to 255 (0 => 256)

FIGURE 6

Note Type 2:
fixed frequency, swept attenuation

Length in bytes: 6
Location: CART ROM
Beginning address: pointed to by bytes 1&2 in that song's CPU RAM data area

Offset	Contents B7 B6 B5 B4 B3 B2 B1 B0	Description
0	CH# 0 0 0 0 1 0	header
1	F2 F3 F4 F5 F6 F7 F8 F9	least significant 8 bits of 10 bit frq data
2	ATN 0 0 F0 F1	ATN = init atn data, LSN = top 2 bits freq
3	NLEN	NLEN * 16.7ms = duration of note

4	ASTEP / ALEN	ALEN = # steps in atnswp: 2 - 15 (0 => 16)
		ASTEP = step size: 1 to 7, -1 to -8
		if whole byte = 00, atn not to be swept

5	APS / APSV	atn duration prescaler: APS = prescaler reload value APSV = initial APSV

NOTE DURATION = NLEN * 16.7ms
  NLEN: 1 to 255 (0 => 256)

ATN SWEEP DURATION = [((ALEN - 1) * APS) + initial APSV] * 16.7ms
duration 1st step = initial APSV * 16.7ms
duration all others = APS
  APS: 0 to 15 (0 => 16)
  APSV: 0 to 15 (0 => 16)
  ALEN: 0 to 15 (0 => 16)

FIGURE 7

Note Type 3:
swept frequency, swept attenuation

Length in bytes: 8
Location: CART ROM
Beginning address: pointed to by bytes 1&2 in that song's CPU RAM data area

Offset	Contents B7 B6 B5 B4 B3 B2 B1 B0	Description
0	CH# 0 0 0 0 1 1	header
1	F2 F3 F4 F5 F6 F7 F8 F9	least sig 8 bits of init 10 bit frq data
2	ATN 0 0 F0 F1	ATN = init atn data, LSN = top 2 bits freq
3	NLEN	NLEN = number of steps in the sweep
4	FPS FPSV	freq sweep duration prescaler: FPS = prescaler reload value FPSV = initial FPSV
5	FSTEP	freq sweep step size: 1 to 127, -1 to -128

6	ASTEP ALEN	ALEN = # steps in atnswp: 2 - 15 (0 => 16)
		ASTEP = step size: 1 to 7, -1 to -8
		if whole byte = 00, atn not to be swept

7	APS APSV	atn duration prescaler: APS = prescaler reload value APSV = initial APSV

NOTE DURATION = [((NLEN - 1) * FPS) + initial FPSV] * 16.7ms
duration 1st step = initial FPSV * 16.7ms
duration all others = FPS
  FPS: 0 to 15 (0 => 16)
  FPSV: 0 to 15 (0 => 16)
  NLEN: 0 to 255 (0 => 256)

ATN SWEEP DURATION = [((ALEN - 1) * APS) + initial APSV] * 16.7ms
duration 1st step = initial APSV * 16.7ms
duration all others = APS
  APS: 0 to 15 (0 => 16)
  APSV: 0 to 15 (0 => 16)
  ALEN: 0 to 15 (0 => 16)

FIGURE 8

Noise notes:
special case type 2 notes

Length in bytes: 5
Location: CART ROM
Beginning address: pointed to by bytes 1&2 in that song's CPU RAM data area

Offset	Contents B7 B6 B5 B4 B3 B2 B1 B0	Description
0	0 0 0 0 0 0 1 0	header (CH# = 0, indicates noise note)
1	ATN | 0 FB SHIFT	MSN = 4 bit noise ATN data (init if swept) LSN = noise control data (SHIFT = NF0 NF1)
2	NLEN	NLEN * 16.7ms = duration of note

3	ASTEP | ALEN	ALEN = # steps in atnswp: 2 - 15 (0 => 16)
		ASTEP = step size: 1 to 7, -1 to -8
		if whole byte = 00, atn not to be swept

4	APS | APSV	atn duration prescaler: APS = prescaler reload value APSV = temp APS variable storage

NOTE DURATION = NLEN * 16.7ms
NLEN: 1 to 255 (0 => 256)

ATN SWEEP DURATION = [((ALEN - 1) * APS) + initial APSV] * 16.7ms
duration 1st step = initial APSV * 16.7ms
duration all others = APS
APS: 0 to 15 (0 => 16)
APSV: 0 to 15 (0 => 16)
ALEN: 0 to 15 (0 => 16)

FIGURE 9

Dedicated cartridge RAM locations and Special Effect format

Length in bytes: 11
Location: CPU RAM
Beginning address: 7020H

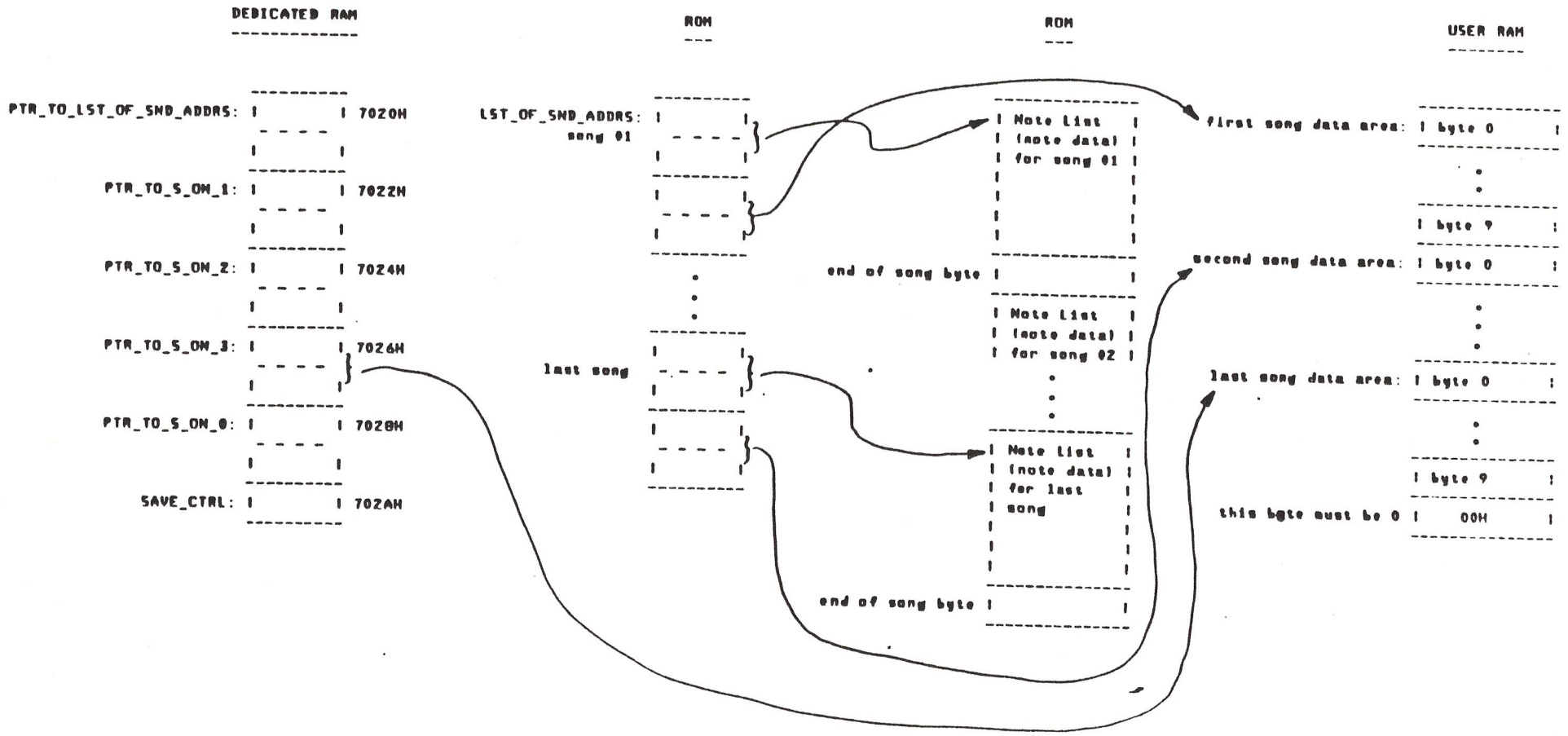
```
PTR_TO_LST_OF_SND_ADDRS   DS 2  ;pointer to start of LST_OF_SND_ADDRS

PTR_TO_S_ON_1    DS 2    ;pointer to data area of song to be played on CH# 1
PTR_TO_S_ON_2    DS 2    ;pointer to data area of song to be played on CH# 2
PTR_TO_S_ON_3    DS 2    ;pointer to data area of song to be played on CH# 3
PTR_TO_S_ON_0    DS 2    ;pointer to data area of song to be played on CH# 0

SAVE_CTRL        DS 1    ;LSN = last control data sent to noise generator
```

Special Effect format

All special effect routines should be written in the following format (SFX = the address of the effect routine, stored in ROM after the effect's header, IX is passed pointing to the song's data area):

```
SFX:     LD (SAVE_x_NNP),HL       ;save address of next note in song
         LD (SAVE_x_SONGNO),A     ;save song's SONGNO
         RET                      ;to LOAD_NEXT_NOTE
SFX+7:   LD HL,SAVE_x_SONGNO      ;test for 1st pass through effect
         BIT 7,(HL)
         JR NZ,NOT_PASS_1
           SET 7,(HL)      ;to prevent further passes thru inits
           ...             ;initialize bytes within the data area here
           RET             ;to LOAD_NEXT_NOTE
NOT_PASS_1: ...            ;code for pass 2 or greater starts here,
         .                 ;which algorithmically modifies freq, atn,
         .                 ;or control data within song data area
         .                 ;pointed to by IX
       RET (to PROCESS_DATA_AREA) if effect not over
         ;if here, effect is over, so restore SONGNO and addr nxt note in song
         LD HL,(SAVE_x_NNP)     ;HL := addr next note in song
         LD DE,SAVE_x_SONGNO    ;DE := addr saved song number
         CALL LEAVE_EFFECT      ;to restore them to bytes 0 - 2 in data area
         JP EFXOVER             ;in PROCESS_DATA_AREA to load song's next note
```

FIGURE 10


DEDICATED RAM
ROM
ROM
USER RAM
PTR_TO_LST_OF_SND_ADDRS:
702OH
PTR_TO_S_ON_1:
702ZH
PTR_TO_S_ON_2:
7024H
PTR_TO_S_ON_3:
7026H
PTR_TO_S_ON_0:
7028H
SAVE_CTRL:
702AH
LST_OF_SND_ADDRS:
song 01
last song
Note List
(note data)
for song 01
end of song byte
Note List
(note data)
for song 02
Note List
(note data)
for last
song
end of song byte
first song data area:
byte 0
byte 9
second song data area:
byte 0
byte 9
last song data area:
byte 0
byte 9
this byte must be 0
00H

December 7, 1982

NOTES AND ERRATA
for ColecoVision Sound Users' Manual
Version 1.1

The following is a list of known errors in the User's Manual and a set of explanatory notes which hopefully serve to clarify some of the concepts presented therein. It is suggested that, prior to reading the Manual, corrections be made according to the following list and references to the explanatory notes be marked at the appropriate passages.

*** ERRATA ***

page 2 about half way down the page, starting near the right margin:
" where x = the song's SONGNO"
- should read -
delete - There is no necessary relationship between a SONGNO and a song data area. As becomes clear later, several songs (with different SONGNOs) can share the same data area. This error also occurs in the first sentence of FIGURE 1.

page 12 next to last line:
- add -
"0 = no sweep"

page 16 first line:
"PLAYSONGS"
- should read -
"PLAY_SONGS"

page 18 last two lines:
"RETs C SET..."
"RETs C RESET..."
- should read -
"RETs Z SET..."
"RETs Z RESET..."

page 21 the following utilities are not generally useful, are not available as entry points, and therefore can not be used:
AREA_SONG_IS
UPATNCTRL
UPFREQ
PT_IX_TO_SxDATA
UP_CH_DATA_PTRS (on page 22)

FIG 2 last line of REST description:
"B4 - B0 = duration, 1 to 31"
- should read -
"B4 - B0 = duration, 1 to 30"

FIG 3 about one third down and half way down the page:
"xxxxH"
- should read -
"7020H"

USER RAM

This is the area in CRAM that the cartridge programmer has chosen to hold the ten byte song data areas which contain sound and timing information to be processed by the OS sound routines. These data areas must be stored as contiguous blocks of ten bytes each. In all cases but one, the programmer may choose to "play" a song in any data area; however, song #1 MUST use the FIRST song data area for the OS routines to work properly. Also, the byte immediately following the last byte in the last data area MUST be zero (this code tells the OS routine SND_MANAGER to stop looking for more song data areas; see bottom of page 5, Users' Manual). This byte will automatically be set to zero by proper invocation of the INIT_SOUND routine before any other OS sound routines are called (see pages 5 and 13).

2) The ColecoVision OS entry point names of some of the sound routines and dedicated locations are different from their names given in the Sound Users' Manual. They are:

Entry Point	Sound Users' Manual
PLAY_IT	JUKE_BOX
SOUND_MAN	SND_MANAGER
SOUND_INIT	INIT_SOUND
NOTES	PTR_TO_LST_OF_SND_ADDRS

You should use the entry point names.

3) Page 22, last paragraph: It is mentioned that a special effect routine may want to call the OS sound routines FREQ_SWEEP and ATN_SWEEP to operate upon data within the effect's data area, which "require that data be ordered appropriately within a song data area". This means:

Whether or not the special effect uses FREQ_SWEEP or ATN_SWEEP: bytes 3 and 4 (see FIGURE 1) MUST contain the frequency and attenuation data as specified. This is because PLAY_SONGS (called every interrupt) will output

For FREQ_SWEEP used by itself - in addition to bytes 3 and 4, bytes 5, 6, and 7 must contain data as specified. Bytes 8 and 9 may be used for whatever (since FREQ_SWEEP doesn't look at them).

For ATN_SWEEP used by itself - in addition to bytes 3 and 4, bytes 5, 8, and 9 must contain data as specified. Bytes 6 and 7 may be used for whatever (since ATN_SWEEP doesn't look at them).

If both FREQ_SWEEP and ATN_SWEEP are used, all bytes in the data area must look as specified in FIGURE 1.

PTR_TO_S_ON_3:
7026-27H - As above, for tone generator #3.

PTR_TO_S_ON_0:
7028-29H - As above, for tone generator #0 (the noise generator).

The final byte at 702AH, SAVE_CTRL, is used by the OS sound routines to store data necessary for smooth operation of the noise generator (see bottom of page 11 in the Users' Manual).

All 11 bytes should be initialized before the OS routines which operate upon them are called: this is done by calling INIT_SOUND and passing the appropriate cartridge-dependent information (see Users' Manual pages 5 and 13).

ROM

Cartridge ROM to be used by OS sound routines is divided into two sections: LST_OF_SND_ADDRS and the Note List.

LST_OF_SND_ADDRS:
A contiguous list of 4 bytes per each song (or special effect) used by the game. In each 4 byte section, the first 2 bytes are a pointer to the beginning of the song's note list (also in ROM). The second two bytes are a pointer to the song data area in RAM to be used by that song (review pages 2 and 3 in the Manual). Of course, another song may also use the same data area, since there can be (and usually are) more songs than there are data areas. NOTE, however, that Song #1 MUST be the first entry in LST_OF_SND_ADDRS for the OS routines to operate properly.

-- EXAMPLE -- The first two bytes in this list are a pointer to song #1's note list, as they MUST be. The second two bytes are a pointer to the song's data area in RAM, shown here pointing to the first song data area as also MUST be the case (see later). As can also be seen from the figure, the last song happens to use the second song data area.

In summary, there is a note list for every note list pointer in LST_OF_SND_ADDRS, but, since songs may share RAM data areas, there are almost always more data areas than there are data area pointers.

NOTE LIST:

The Note List is a contiguous block of ROM containing the data which comprise the notes of each song. The number of bytes per song of course varies with the length of the song. The first byte of the note list for a song is pointed to by a two byte entry in LST_OF_SND_ADDRS (see above). The last byte of each song's note list is a single byte end of song/repeat code (see page 2 and Figure 2 in the Users' Manual).

also, half way down the page:
"...pointed to by CPU RAM word LST_OF_SND_ADDRS"
- should read -
"...pointed to by CPU RAM word PTR_TO_LST_OF_SND_ADDRS"

FIG 4 second line:
"Length in bytes: 2"
- should read -
"Length in bytes: 1"

also, about 6 lines down from that:
"B4 - B0= duration, 1 to 31"
- should read -
"B4 - B0= duration, 1 to 30"

*** NOTES ***

1) After reading through page 6 in the Users' Manual, it may be helpful to review the following summary of dedicated pointers and data structures (discussion refers to Figure 10, included as part of these notes):

DEDICATED RAM

Prior to calling any OS sound routines (except INIT_SOUND), the 11 CRAM locations 7020H through 702AH must be initialized to meaningful data. Ten of the locations are two byte pointers:

PTR_TO_LST_OF_SND_ADDRS:
7020-21H - Points to the start of a list of pointers in cartridge ROM, LST_OF_SND_ADDRS (see later). The OS sound routines know where the cartridge-dependent LST_OF_SND_ADDRS is stored through this pointer. It is shown pointing to the first byte in this ROM list (as it must).

PTR_TO_S_ON_1:
7022-23H - This and the following three pointers are used by OS sound routines to store the addresses of the four song data areas which currently contain the sound data to be modified/output to the four sound generators in the TI sound chip. This pointer stores the address for the song currently playing on tone generator #1.

-- EXAMPLE -- PTR_TO_S_ON_3 is shown pointing to the second song data area. I.e., data for the song currently playing on tone generator #3 happens to be stored in the second song data area. The second data area is used for purposes of illustration only: other songs may very well require that data for tone generator #3 be stored in a different song data area.

PTR_TO_S_ON_2:
7024-25H - As above, for tone generator #2.