SOUND COMMANDER

The IBM PC was probably one of the most important reasons why computers have come to be what they are today. One reason for the wide consumer acceptance of the IBM PC was its open architecture and standard interfaces. However, IBM left one thing out: high-quality sound. The PC does have a speaker that clicks and beeps, but I would have rather no sound than that atrocious little speaker.

Before long, a few companies realized that millions of people have PCs and many other companies were writing games for the PC that made great use of sound effects. Therefore, it wasn't long before sound cards started emerging (with the Adlib architecture being the most accepted in the early days). So much for history.

We aren't going to build anything as complex as an Adlib card, but we are going to make a sound synthesizer that can be used for serious applications such as arcade games and music synthesis. As a bonus, we will design an analog and a digital joystick interface into the card.

The sound card will be based on the Commodore 6581 SID (Sound Interface Device) chip. This is the same chip that the Commodore 64 uses. If you've ever heard a C64 make sound, you have to admit it's not bad. The chip is capable of both music and voice synthesis.

I selected the 6581 for three reasons. First, the SID has three separate sound generators, each with its own ADSR (Attack-Decay-Sustain-Release) envelope, filtering, and feedback modulation hardware, making complex sound synthesis easy. Second, its hardware interface is very simple and requires very few discrete support components. Finally, the SID also has two analog inputs that can be used to read a potentiometer (which we will use for the analog joystick).

System Overview

First I am going to describe the basic philosophy behind the hardware and software of our sound system. Let's begin with the card itself.

Most plug-in cards use I/O ports for communication. Our card will follow this trend. We will communicate with the card via six I/O ports adjacent to each other in I/O space. The SID has a very simple architecture from a programming point of view. It consists of 29 registers (refer to Figures 1 and 2 to see register descriptions), most of which are write only (four are readable).

To control the SID, we need to address the register of interest and either read from or write to it with the proper timings. The first problem we must overcome is simulating the proper timings. The SID is supposed to be connected to a 65*xx* or 68*xx* series processor system. The PC's timings for an I/O write and read are so fast they barely meet the constraints of these older (slower) processors. Therefore, to be safe, I created a Finite State Machine that will simulate slower timings that will meet the SID's requirements.

Since we are using I/O ports and not memory-mapped I/O, we can't directly access the SID. A buffer must be created between the SID and our I/O-mapped communications channel.

In order to communicate with the chip, we simply place an address on its address bus and either write to or read from the chip via the data bus (driving the appropriate control signals). Therefore, our buffer will consist of three latches: one to hold the address of the register, one to hold data during a write operation, and one to hold the data retrieved during a read operation. These latches are controlled via I/O port selection pulses.

We will decode the PC bus such that our card will exist in I/O space at some base address, say 300H (we will

Andre' Lamothe

André holds degrees in mathematics, computer science, and electrical engineering. He is the president of Andromeda Industries in San Jose and is currently doing research and development in virtual reality.

	Address				Reg #	Data									
	A4	A3	A1	A0	(Hex)	D7	D6	D5	D4	D3	D2	D1	D0	Reg Name	Reg Type
														VOICE 1	
0	0	0	0	0	00	F7	F6	F5	F4	F3	F2	F1	F0	Freq Lo	Write-only
1	0	0	0	1	01	F15	F14	F13	F12	F11	F10	F9	F8	Freq Hi	Write-only
2	0	0	1	0	02	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	PW LO	Write-only
3	0	0	1	1	03	—	—	—	—	PW11	PW10	PW9	PW8	PW HI	Write-only
4	0	0	0	0	04	NOISE				TEST		SYNC	GATE	Control Reg	Write-only
5	0	0	0	1	05	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/Decay	Write-only
6	0	0	1	0	06	STN3	STN2	STN1	STN0	RLS3	RLS2	RLS1	RLS0	Sustain/Release	Write-only
														VOICE 2	
7	0	0	1	1	07	F7	F6	F5	F4	F3	F2	F1	F0	Freq Lo	Write-only
8	0	1	0	0	08	F15	F14	F13	F12	F11	F10	F9	F8	Freq Hi	Write-only
9	0	1	0	1	09	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	PW LO	Write-only
10	0	1	1	0	0A	—	—	—	—	PW11	PW10	PW9	PW8	PW HI	Write-only
11	0	1	1	1	0B	NOISE				TEST		SYNC	GATE	Control Reg	Write-only
12	0	1	0	0	0C	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/Decay	Write-only
13	0	1	0	1	0D	STN3	STN2	STN1	STN0	RLS3	RLS2	RLS1	RLS0	Sustain/Release	Write-only
														VOICE 3	
14	0	1	1	0	0E	F7	F6	F5	F4	F3	F2	F1	F0	Freq Lo	Write-only
15	0	1	1	1	0F	F15	F14	F13	F12	F11	F10	F9	F8	Freq Hi	Write-only
16	1	0	0	0	10	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	PW LO	Write-only
17	1	0	0	1	11	—	—	—	—	PW11	PW10	PW9	PW8	PW HI	Write-only
18	1	0	1	0	12	NOISE				TEST		SYNC	GATE	Control Reg	Write-only
19	1	0	1	1	13	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/Decay	Write-only
20	1	0	0	0	14	STN3	STN2	STN1	STN0	RLS3	RLS2	RLS1	RLS0	Sustain/Release	Write-only
														Filter	
21	1	1	0	1	15	—	—	—	—	—	FC2	FC1	FC0	FC LO	Write-only
22	1	1	1	0	16	FC10	FC9	FC8	FC7	FC6	FC5	FC4	FC3	FC HI	Write-only
23	1	1	1	1	17	RES3	RES2	RES1	RES0	Filt EX	Filt 3	Filt 2	Filt 1	RES/Filt	Write-only
24	1	0	0	0	18	3 OFF	HP	BP	LP	VOL3	VOL2	VOL1	VOL0	Mode/Vol	Write-only
														Misc	
25	1	0	0	1	19	PX7	PX6	PX5	PX4	PX3	PX2	PX1	PX0	POTX	Read-only
26	1	0	1	0	1A	PY7	PY6	PY5	PY4	PY3	PY2	PY1	PY0	POTY	Read-only
27	1	0	1	1	1B	07	06	05	04	03	02	01	00	OSC3/Random	Read-only
28	1	1	0	0	1C	E7	E6	E5	E4	E3	E2	E1	E0	ENV3	Read-only

figure 1: The 6581 SID chip's 25 write-only registers and just four read-only registers make it appear as if programming the part is difficult. It doesn't have to be.

design the decoder logic so that the card can be addressed anywhere in the I/O space on a base-8 boundary). Then at locations 300h–305h, we would access the buffer latches as needed. Once the address and data levels have been placed in these buffer latches, we start the Finite State Machine via a write to one of the ports (the data doesn't matter during this write operation). The Finite State Machine will then control the SID pins appropriately to accomplish the desired operation.

Let's Talk About the Software

I began by writing the drivers needed to access any register in the SID. However, I was having so much fun programming the chip that I got a little carried away and wrote a fairly complete interface.

This interface has three different layers. The lowest layer allows the programmer full control of all aspects of the card and all of the associated hardware. This includes the buffers, the Finite State Machine, the joysticks, and the SID chip itself. With this layer alone you could create any sound you wish.

The next layer of software is based on the first. This layer has higher-level control functions that are designed to insulate the programmer from controlling the hardware directly. Instead of having to deal with I/O ports, buffer latches, and finite state machines, the programmer can make calls like `Write_To_Register()`, `Read_From_Register()`, and `Reset_Card()`.

The final layer allows the programmer to synthesize notes and music in a more natural manner. When we use this layer, the programmer can think of the card as a music synthesizer instead of a piece of hardware with a bunch of abstract registers. This "high-level" software allows the programmer to concentrate on the characteristics of the sound he wants to make instead of the mechanics of controlling the card.

As an example, one of the functions in this layer is called `Play_Song()`. To use this function, the pro-

Freq Lo	Register Numbers: 0, 7, 14 Set the lower 8 bits of the oscillator to control the frequency of a channel.
Freq Hi	Register Numbers: 1, 8, 15 Set the upper 8 bits of the oscillator to control the frequency of a channel.
PW Lo	Register Numbers: 2, 9, 16 Set the lower 8 bits of the pulse width (duty cycle) of an oscillator.
PW Hi	Register Numbers: 3, 10, 17 Set the upper 4 bits of the pulse width (bits 4–7 are not used).
Control Reg	Register Numbers: 4, 11, 18 Set the following options on the oscillators: Gate (bit 0): Triggers the ADSR cycle. Sync (bit 1): Synchronizes oscillator 1 with oscillator 3. RingMod (bit 2): Replaces triangle wave with ring-modulated wave. Test (bit 3): Resets and locks the oscillators. Tri (bit 4): Selects triangular waveform output. Saw (bit 5): Selects sawtooth waveform output. Sqr (bit 6): Selects pulse waveform output. Noise (bit 7): Selects noise waveform output.
Attack/Decay	Register Numbers: 5, 12, 19 Bits 4–7 select attack rates. Bits 0–3 select decay rates.
Sustain/Rel	Register Numbers: 6, 13, 20 Bits 4–7 select sustain levels. Bits 0–3 select release rates.
FC Lo	Register Number: 21 Bits 0–2 control the filter's cutoff freq. Bits 3–7 aren't used.
FC Hi	Register Number: 22 Along with FC Lo, controls the filter's cutoff frequency. The approximate range is 30 Hz–12 kHz.
Res/Filt	Register Number: 23 Bits 4–7 control the resonance (peaking effect) of the filter. Bits 0–3 determine whether the signal for each of the voices will be sent directly to the audio output or altered according to the selected filter parameters.
Mode/Vol	Register Number: 24 Bits 0–3 select volume levels for the audio output. Bits 4–7 select various filter mode and output options: LP (bit 4): Selects the low-pass output of the filter. BP (bit 5): Selects the band-pass output of the filter. HP (bit 6): Selects the high-pass output of the filter. 3 Off (bit 7): Disconnects voice 3 from audio path.
POTX	Register Number: 25 Allows the processor to read the position of the potentiometer tied to POTX (pin 24). Values range from 0 (minimum resistance) to 255 (maximum resistance).
POTY	Register Number: 26 Allows the processor to read the position of the potentiometer tied to POTY (pin 23). Values range from 0 to 255.
OSC3/ Random	Register Number: 27 Allows the processor to read the upper 8 output bits of oscillator 3. Selecting the noise waveform for oscillator 3 will generate random numbers.
ENV 3	Register Number: 28 Allows the processor to read the output of the voice 3 envelope generator.

figure 2: Each register plays a part in defining what the final output will sound like. Note that there are three channels with corresponding registers that perform identical functions.

grammer creates the notes of the song, places them in a string, and passes the string to the function. The music is then played in the instrument, tempo, and volume that the programmer selected.

Finally, I have included a couple code samples so you can see how the software interface is used to make working programs.

Hardware Description

Now I'm going to explain in detail how the sound card works, along with the motivation behind each circuit. Before we begin talking about the sound card itself, let's take a closer look at the SID. For the following explanations, refer to the register descriptions in Figure 2.

The SID consists of three independent sound generators, each with its own amplitude modulation hardware. With the amplitude modulation hardware, you can control the way the sound's amplitude varies as a function of time. This is commonly known as the *envelope modulation*. See Figure 3 for a list of SID oscillator values for generating standard musical notes.

Musicians have found that four parameters are enough to give reasonable control over a sound's envelope. These parameters make up what is called the *ADSR envelope* (see Figure 4). The Attack phase is the portion of the envelope where the sound gains its initial amplitude. The Decay phase allows the sound to fall off to a lower level. The Sustain phase holds this amplitude. Finally, the Release phase allows the amplitude to die back down.

The SID is capable of generating standard waveforms such as square, triangular, and sawtooth. These waveforms can then be amplitude modulated with a given envelope. That's all we need to synthesize any sound—even human voice.

Next I'll describe the other support hardware a subsystem at a time.

See Figure 5 for the complete schematic. There are three subsystems on the card with some extra logic for reading the digital joystick and resetting the system. The three main systems are:

- Decoder Logic—allows the user to communicate with the card's buffer latches and finite state machine with a specific set of I/O addresses.
- Buffer Logic—sits between the PC bus and the SID bus. The buffers allow us to read or write data at our leisure.
- Finite State Machine—simulates 65*xx* timings. It is the FSM that carries out our commands by driving the SID's control lines.

The Decoder Logic

Whenever building a plug-in card, you must allow the card to be accessed at some specific set of addresses in memory space or in I/O space. Our card will occupy I/O space only. The card needs six contiguous I/O ports to operate. Asking for six I/O ports to be free and adjacent is quite a request, so I designed the card so you can place the card's base address anywhere in I/O space.

The DIP switches are used to set the base address of the card. After building the card, you will set them to an area of I/O space that you know is open and won't conflict (I suggest starting with 300H).

Now, let's get into the hardware of the Decoder logic. We want to activate a set of signals whenever a certain I/O address is placed on the PC bus with the proper control signals. To accomplish this, we only need the address bus, *IOR (I/O read pulse, active low), *IOW (I/O write pulse, active low), and AEN (DMA address enable, active high).

The high-order bits of the address bus are fed into U1 and U2, 4-bit magnitude comparators. The address is compared against what is set as the card's base address (determined by the settings of the DIP switches). Also, you will notice that AEN is an input to the comparators. AEN must always be low for an I/O transaction to occur, so to save gates I used one of the free inputs to the comparator to make sure that AEN is low. Therefore, one of the DIP switches will always be low.

Following the signal flow, the output of the comparators is sent as input to AND gate U9b. The output of U9b is then sent to one of the chip enables

Musical Note	Freq. (Hz)	Osc. Fn (Decimal)	Musical Note	Freq. (Hz)	Osc. Fn (Decimal)
0 C0	16.35	274	48 C4	261.63	4389
1 C0#	17.32	291	49 C4#	277.18	4650
2 D0	18.35	308	50 D4	293.66	4927
3 D0#	19.44	326	51 D4#	311.13	5220
4 E0	20.60	346	52 E4	329.63	5530
5 F0	21.83	366	53 F4	349.23	5859
6 F0#	23.12	388	54 F4#	370.00	6207
7 G0	24.50	411	55 G4	392.00	6577
8 G0#	25.96	435	56 G4#	415.30	6968
9 A0	27.50	461	57 A4	440.00	7382
10 A0#	29.14	489	58 A4#	466.16	7821
11 B0	30.87	518	59 B4	493.88	8286
12 C1	32.70	549	60 C5	523.25	8779
13 C1#	34.65	581	61 C5#	554.37	9301
14 D1	36.71	616	62 D5	587.33	9854
15 D1#	38.89	652	63 D5#	622.25	10440
16 E1	41.20	691	64 E5	659.25	11060
17 F1	43.65	732	65 F5	698.46	11718
18 F1#	46.25	776	66 F5#	740.00	12415
19 G1	49.00	822	67 G5	783.99	13153
20 G1#	51.91	871	68 G5#	830.61	13935
21 A1	55.00	923	69 A5	880.00	14764
22 A1#	58.27	978	70 A5#	932.33	15642
23 B1	61.74	1036	71 B5	987.77	16572
24 C2	65.41	1097	72 C6	1046.50	17557
25 C2#	69.30	1163	73 C6#	1108.73	18601
26 D2	73.42	1232	74 D6	1174.66	19709
27 D2#	77.78	1305	75 D6#	1244.51	20897
28 E2	82.41	1383	76 E6	1318.51	22121
29 F2	87.31	1465	77 F6	1396.91	23436
30 F2#	92.50	1552	78 F6#	1479.98	24830
31 G2	98.00	1644	79 G6	1567.98	26306
32 G2#	103.83	1742	80 G6#	1661.22	27871
33 A2	110.00	1845	81 A6	1760.00	29528
34 A2#	116.54	1955	82 A6#	1864.65	31234
35 B2	123.47	2071	83 B6	1975.53	33144
36 C3	130.81	2195	84 C7	2093.00	35115
37 C3#	138.59	2325	85 C7#	2217.46	37203
38 D3	146.83	2463	86 D7	2349.32	39415
39 D3#	155.56	2610	87 D7#	2489.01	41759
40 E3	164.81	2765	88 E7	2637.02	44242
41 F3	174.61	2930	89 F7	2793.83	46873
42 F3#	185.00	3104	90 F7#	2959.95	49660
43 G3	196.00	3288	91 G7	3135.96	52613
44 G3#	207.65	3484	92 G7#	3322.44	55741
45 A3	220.00	3691	93 A7	3520.00	59056
46 A3#	233.08	3910	94 A7#	3729.31	62567
47 B3	246.94	4143	95 B7	3951.06	66288

figure 3: SID oscillator values to produce musical notes.

of U3, which is a 3-to-8 decoder. The chip select of this decoder is from the logical ANDing of the *IOR and *IOW signals.

If the computer performs an I/O read or write at the base address, then one of the outputs of U3 (determined by the lower three bits of the address bus, A0–A2) will become active. This allows the PC to access one of eight different I/O ports on our card. Currently, I am only using the first six.

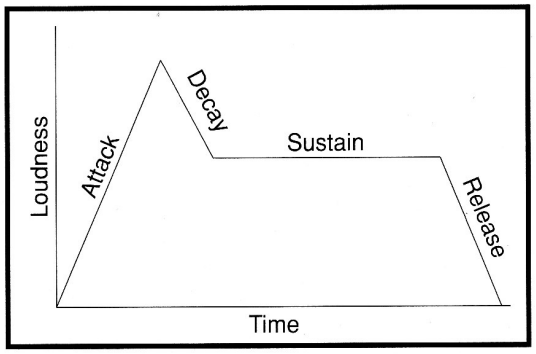

figure 4: The amplitude of a signal is typically described using four parameters: attack, decay, sustain, and release (ADSR).

The Buffer Logic

The buffer logic is used (as it name suggests) to buffer data. All transactions between the PC and the SID are filtered through three buffers. One buffer (U4) is used to latch addresses, another (U5) to latch data to be written, and a third (U6) to latch data that has been read from the SID. These buffers make the control logic less complex and allow us to use I/O port addressing instead of memory-mapped I/O.

The Finite State Machine (FSM)

The FSM used in our card is composed of three flip-flops (U8b, U14a, and U14b), some AND gates (U10a, U9c, and U9d), and a few inverters (U12b, U12d, and U12e). When Y3 of U3 (74LS138) goes low (meaning the user wants to do an operation), flip-flop U8b is set high to synchronize the timings to the SID clock. Flip-flop U14a will go high when phase *PHI2 of the biphase clock goes low to start the FSM off at a known clock state.

Next, the outputs of U14a are logically combined with the current

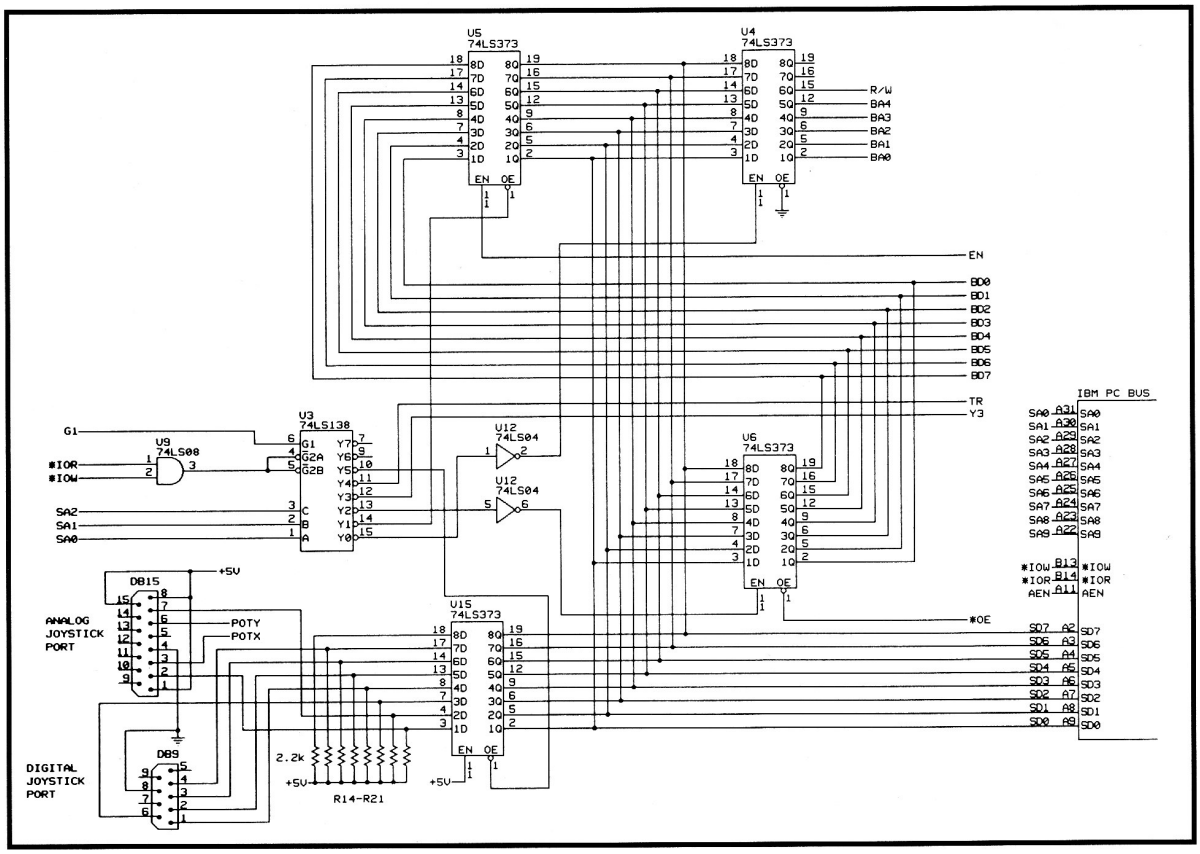

figure 5a: The Sound Commander connects to the PC bus (lower right) through four buffer latches. The board also supports both analog and digital joysticks.

Assembling the Software Pieces

I used the Microsoft 5.1 C compiler to create this project. If you don't have MS C, then you can manually change the special functions in MS C to whatever their analogous functions are in the compiler you are using. In any case, follow the same basic steps outlined here.

Step 1—Create the sound library

Create a library named SOUND.LIB from SOUND.C and SOUND.H. To do this, first create an object from SOUND.C and SOUND.H using the following:

```
CL -C -AM SOUND.C
```

Now, create a library with the newly created object SOUND.OBJ. To do this, invoke the library manager named LIB. Then create a library called SOUND.LIB by inserting SOUND.OBJ. Once, you have this library, you will always link to it.

Step 2—Compile the debugging software

The first step in creating the two executables, SIO.EXE and SSID.EXE, is to compile each program:

```
CL -C -AM SIO.C
CL -C -AM SSID.C
```

Now, you will have two objects, SIO.OBJ and SSID.OBJ.

Step 3—Create the executables

Next, invoke the linker to link together the objects with the library you previously created (SOUND.LIB) using the following commands:

```
LINK SIO.OBJ,,,SOUND.LIB
LINK SSID.OBJ,,,SOUND.LIB
```

You will now have two executables named SIO.EXE and SSID.EXE.

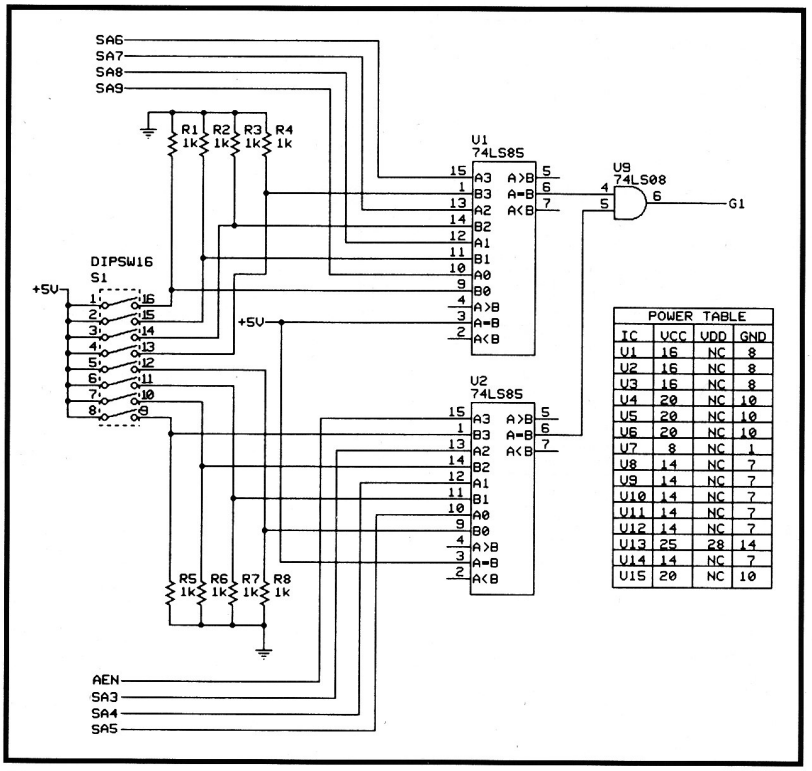

figure 5b: A set of DIP switches selects the base address for the board, which is decoded by a pair of comparators.

operation (i.e., a read or a write) and the result is sent to the control pins of the U4, U5, and U6. After one clock period, the FSM activates *CS (chip select) on the SID, allowing the transaction to occur.

Finally, the FSM resets itself and waits to be started again. Note that during the FSM state transitions, the FSM will disregard another "Go" command via Y3 of the decoder (U3).

The Digital Joystick Logic

The sound card has two joystick ports. One is digital such as used on an Atari or a C64, and the other is analog such as on a PC or Apple II. The analog joystick is implemented using the SID's built-in A/D converters and is explained more fully later, but I'll cover the digital joystick here.

The digital joystick is simply a port (base + 5) that can have an Atari-type joystick connected to it. When the PC reads this port, it will get the switch status of the joystick and

(through software) can find the direction the joystick is being moved and the status of the fire button.

The System Reset Logic

The SID is the only thing in the system that needs to be reset. The strobe generated by accessing one of the remaining unused I/O ports is used to do the reset, however the PC is too fast and pulse is too short to be tied directly to the SID's reset input. Therefore, a one-shot pulse stretcher (U7) is placed between the I/O port and the SID chip to extend the reset pulse to approximately 10 µs, which is long enough to reset the SID.

Software Description

Let's begin with the system software that controls the sound card, then I'll cover the diagnostics and demonstration software. Finally, I will explain how to compile and link the software into executables.

Hardware Control (Low Level)

The low-level functions consist of calls that control the hardware directly by accessing the I/O ports of the card and include:

- `Scd_Write_Addr(address, read_write_flag)`
- `Scd_Write_Data(data)`
- `Scd_Read_Data()`
- `Scd_Run()`
- `Scd_Reset(type_of_reset)`

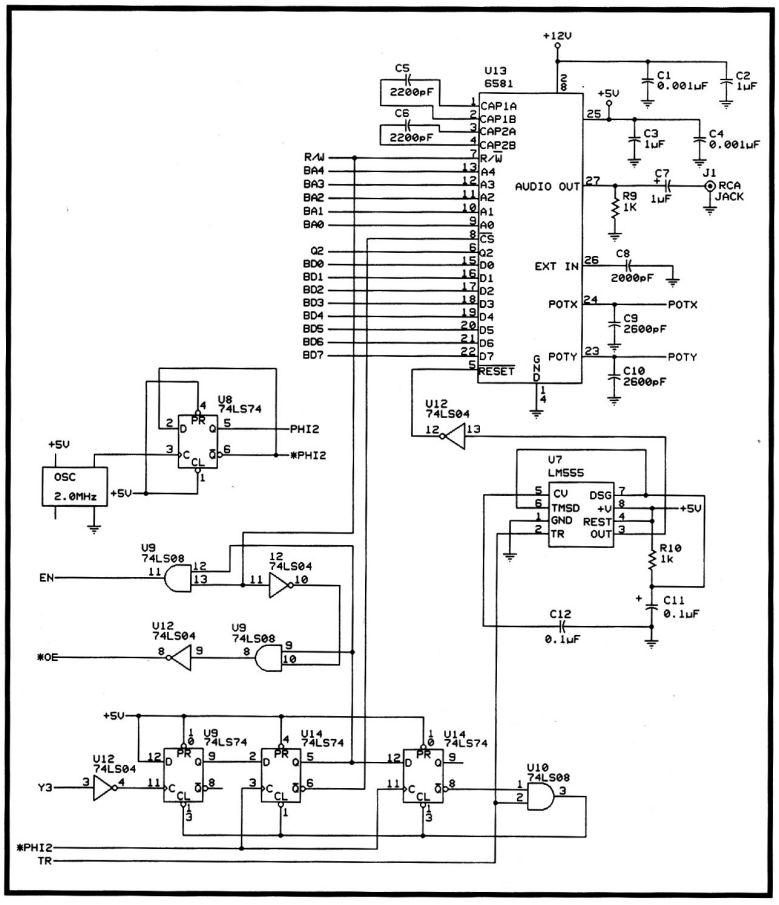

figure 5c: The heart of the Sound Commander is the 6581 SID chip. In order to adapt the chip's timing to the PC bus, a finite state machine (lower left) is used. The LM555 is used to extend the reset pulse to the SID chip.

To access the SID, we must address one of its registers and perform a read or a write. This process is done in three steps:

First, set the register address using `Scd_Write_Addr (address)`. You also need set bit 5 of the register address to indicate whether you're doing a read or a write operation (high for read, low for write).

Next, if you're doing a write operation, you must latch the write data into the write buffer using `Scd_Write_Data(data)`. Skip this step if you're doing a read.

Finally, you activate the finite state machine to perform the function using `Scd_Run()`. If you did a read operation, you may use `Scd_Read_Data()` to retrieve the data read from the SID. See Listings 1a and 1b for samples of code that can be used to accomplish these steps.

Finally, to reset the hardware, the function `Scd_Reset()` is used. This function can do either a hardware reset or a software reset. The hardware reset actually resets the SID chip, whereas the software reset zeros each register. See Listing 1c for a sample of this call.

As you can see, this level of software is very rudimentary and not very powerful. Based on these low-level functions, I created a more palatable method of reading and writing registers.

Register Control (Medium Level)

The medium-level software consists of calls based on the low-level software. These calls insulate the programmer from the details of the control registers and finite state machine. It consists of three functions:

- Scd_Write_Reg(register, data)
- Scd_Read_Reg(register)
- Delay(duration)

Scd_Write_Reg(register, data) is used to write to one of the SID chip's registers as shown in Listing 1d.

Scd_Read_Reg(register) is used to read data from a register (Listing 1e). This function only makes sense when reading from registers 25–28. Reading from a register not in that range will give unpredictable results.

Finally, Delay(duration) causes the computer to wait for a period of time (in milliseconds). Since this function uses simple delay loops, you may need to adjust it to run on different speed machines.

Music Synthesis (High Level)

This layer of software is by far the most complex, and the most interesting. In its simplest form, it allows you to make a single tone with characteristics such as waveform, envelope, filtering, and volume. It also lets you create entire songs composed of notes (which you put in a special format) in a selected musical instrument. The high-level software consists of the following functions:

- Scd_Set_Envelope(channel, attack, decay, sustain, release)
- Scd_Set_Instrument(channel, instrument)
- Scd_Set_Waveform(channel, waveform, pulse width)
- Scd_Set_Volume(volume)
- Scd_Set_Filter(frequency, filter type, resonance)
- Scd_Sound_On(channel, frequency, volume)
- Scd_Sound_Off(channel)
- Scd_Play_Song(channel, "the notes of the song")

```
/* A: Read an SID register and save it in a variable */

    /* send address of register along with READ bit set*/
Scd_Write_Addr(27 | SCD_R);
    /* tell the state machine to "Go"*/
Scd_Run();
    /* retrieve the data from the buffer*/
data = Scd_Read_Data();
```

```
/* B: Write data to an SID register */

    /* send address of register along with WRITE bit set*/
Scd_Write_Addr(24 | SCR_W);
    /* send data to be written to data latch buffer*/
Scd_Write_Data(15);
    /* tell the state machine to "Go" */
Scd_Run();
```

```
/* C: Reset the SID chip (both hard and soft) */

     /* reset the SID chip via pin 5 (reset) */
Scd_Reset(SCD_HARD_RESET);

     /* or if you wanted only a software reset...*/
     /* this fragment will clear the registers, but not */
     /* reset the SID chip */
Scd_Reset(SCD_SOFT_RESET);
```

```
/* D: Medium-level routine to write to an SID register */
/*     in one step */

       /* write a 50 to register 0 (channel 1's low */
       /*  word of frequency) */
Scd_Write_Reg(0, 50);
```

```
/* E: Medium-level routine to read an SID register in */
/*     one step */

       /* read data from register 25 (potentiometer x) */
unsigned int x_pot;
x_pot = Scd_Read_Reg(25);
```

listing 1: Low- and medium-level routines make it possible to access the SID's registers directly.

- Scd_Read_Analog_Stick(pointer to analog stick record)
- Scd_Read_Digital_Stick(pointer to digital stick record)
- Scd_Random()

Many of these functions have parameters that are hard to remember, so I have defined many useful constants in the header file, such as register, instrument, and waveform names. Using the defined names will make your code more readable and easier to debug.

Some of the functions do overlap. Instead of explaining each one separately, I am going to give you a few examples so you can see how

```
/* A: Play a middle C on channel 0 using a piano */

#include <stdio.h>
#include <sound.h>
main()
{
    Scd_Reset(SCD_FULL_RESET);
    /* set channel to zero and instrument to piano */
    Scd_Set_Instrument(0,SCD_PIANO);
    /* clear out filters */
    Scd_Set_Filter(0,0,0);
    /* play the sound-turn on channel 0 with full volume */
    /* the parameter 1097 is decimal value for C */
    Scd_Sound_On(0,1097,15);
}
```

```
/* B: Play an A using a flute */

#include <stdio.h>
#include <sound.h>
main()
{
    Scd_Reset(SCD_FULL_RESET);
    /* set the envelope to guitar */
    Scd_Set_Envelope(0,9,4,4,0);
    /* set waveform for a guitar */
    Scd_Set_Waveform(0,1,0);
    /* clear out filters */
    Scd_Set_Filter(0,0,0);
    /* play the sound-turn on channel 0 with full volume */
    /* the parameter 1845 is decimal value for A */
    Scd_Sound_On(0,1845,15);
}
```

```
/* C: Play a scale using an organ */

#include <stdio.h>
#include <sound.h>
main()
{
    Scd_Reset(SCD_FULL_RESET);
    /* set the instrument */
    Scd_Set_Instrument(0,SCD_ORGAN);
    /* turn all filtering off */
    Scd_Set_Filter(0,0,0);
    /* set the volume to half */
    Scd_Set_Volume(8);
    /* play the scale in the 3rd octave with whole notes */
    /* until the keyboard is hit */
    while (!kbhit()) {
        Scd_Play_Song(0,"CW3 DW3 EW3 FW3 GW3 AW3 BW3 CW4");
    }
}
```

listing 2: At the highest level there are calls for setting channel parameters and even to play complete songs.

they are used in context. As always, you should refer to the actual code and header file for clarification of parameters and associated data types. See Listing 2 for examples of how to use these functions.

If you execute the program in Listing 2a, you will hear a single piano stroke that will decay to silence. Now, let's do something a bit more complex. Instead of using the function `Scd_Set_Instrument` to set the envelope and waveform, let's do it ourselves with the functions `Scd_Set_Envelope` and `Scd_Set_Waveform`. But instead of a piano, I'll make a flute play an A. See Listing 2b for the code.

Next, I'll use `Scd_Play_Song` to play a series of notes. To keep it simple, I'll play the musical scale with an organ (Listing 2c). I have included a few short songs in Figure 6. After listening to the scales, try substituting the other song data and see if they sound like you expect. I think one of them needs a little help!

The string of notes that `Scd_Play_Song` takes as a parameter deserves a bit of explanation. Each note consists of a group of characters that define the note, length of note, octave of note, and whether the note is dotted. Each note block must be separated from the others by at least one space may not contain any white space (see Figure 7).

Each note group must have at least the note, the duration, and the octave. Optionally, each group may have a sharp designator or a dotted rhythm control designator. Tempo varies depending on your computer's speed, so you must adjust the `cpu_reference` variable to the proper value experimentally (but only if you want to use `Scd_Play_Song`). The function is limited since it only allows one channel to play at once.

There are two more functions to support the analog and digital joystick

interfaces. See the code samples in Listing 3.

The last feature of the SID is a pseudorandom number generator. The value of the current random number can be accessed by reading register 27. The chip uses sound channel 3 to generate the number, so that channel must be active for this feature to work correctly.

I hope I've described the driver code in enough detail so you can use it productively, but definitely read the source code and comments for further understanding. As far as the high-level layer of code goes, I don't think Scd_Play_Song will be giving a Yamaha synthesizer much competition because the function is so limited. I include the high-level functions as more of a starting point than a final solution. The lower-level functions should make it easier for you to write your own code, though.

Now, that I've explained how the code works, I want to describe some software I wrote for testing and debugging purposes.

Diagnostic Software

The diagnostic software can be helpful in debugging and experimenting with the sound card and the SID chip. The first piece of software, SIO, allows you to talk directly to any port in the computer and is most useful during the debugging phase of your project. By reading and writing the Sound Commander's ports manually in conjunction with a logic probe or oscilloscope, you should be able to track down problems on the board.

The second piece of software, SSID, is most useful for experimenting rather than debugging. Once you get the card working (or think it's working), you will want to get right in there and make some sound. To do this, SSID allows you to easily access the 6581's registers. The program is self explanatory and menu driven.

"Row, Row, Row Your Boat"

```
CQ3 R4 CQ3 R4 CI3 DI3 EQ3 EQ3 DI3 EI3 FI3 GQ3.1 CS4 CS4 CS4
GS3 GS3 GS3 ES3 ES3 EI3 CS3 CS3 CS3 R3 GQ3.1 FQ3 EQ3.1 DI1
CW3
```

"Surprise Symphony" (Haydn)

```
CW4 CW4 EW4 EW4 GW4 GW4 EW4 EW4 FW4 FW4 DW4 DW4 BW3 BW3 GW3
CW4 CW4 EW4 EW4 GW4 GW4 EW4 EW4 CW5 CW5 FW4# FW4 GW4 GW4 GW3
```

figure 6: Sample song strings for the Play_Song function.

Note|Duration|Octave|[#]|[.num]

where "|" means concatenate and "[ ]" means optional

Each parameter may take on the following values:

Note	A,B,C,D,E,F,G.
Duration	W (whole note), H (half note), Q (quarter note), I (eighth note), S (sixteenth note).
Octave	0,1,2,3,4,5,6,7
#	Optional parameter. Makes the note sharp.
num	Optional parameter. Allows for dotted rhythms. "num" can be 1–4. Final note duration = Duration + (Duration × 0.5^{num}).

Examples:

AW2#.1	Represents octave-two A sharp with duration of one whole note plus half a whole note.
AW2#.2	Represents octave-two A sharp with duration of one whole note plus a quarter of a whole note.

figure 7: The Play_Song function can handle virtually all notes and rhythms.

```
#include <stdio.h>
#include <sound.h>
main() {
d_stick_mess digital; /*holds a digital joystick message*/
a_stick_mess analog;  /*holds a analog joystick message*/

    while (!kbhit()) {
        /* read the joysticks (/
        Scd_Read_Analog_Stick((a_stick_mess_ptr)&analog);
        Scd_Read_Digital_Stick((d_stick_mess_ptr)&digital);

        /* print the results */
        _settextposition(0,0);
        printf("Digital Joystick...");
        printf("\n Direction = %d  ",digital.direction);
        printf("\n Button  = %d  ",digital.button);
        printf("\n\nAnalog Joystick...");
        printf("\n Xpos  = %d  ",analog.xpos);
        printf("\n Ypos  = %d  ",analog.ypos);
        printf("\n Buttons = %d  ",analog.buttons);
    }
}
```

listing 3: The sound library routines make it easy to access the digital and analog joystick data.

See Listings 4–7 for all the source code. The sidebar gives some helpful hints on how to create the final sound programs.

Have Fun

That about covers it. I would recommend using PC prototyping board that contains a ground plane to minimize noise. Try to keep the audio circuits away from the digital circuits. Wire-wrapping can be used to minimize the time spent building the board. Finally, don't forget to include a bypass capacitor next to each chip.

Now you're ready to include high-quality music and sound in your next project. Have fun!

Contact

Commodore Semiconductor Group
950 Rittenhouse Rd.
Norristown, PA 19403
(215) 666-7950