1-1-1994

3D Sound Generation For Virtual Environments, Convolvotron Technical Documentation And MIDIator MIDI Library: Summary Report

Kimberly Abel

Michael Goslin

INSTITUTE FOR SIMULATION AND TRAINING

Contract Number
N61339-92-C-0076

May 1994

SUMMARY REPORT

3D Sound Generation for Virtual Environments
Convolvotron Technical Documentation and MIDIator MIDI Library

Prepared by:	Michael Tedder
Project Manager:	Jacquelyn Ford Morie
Technical Direction:	Michael Goslin
Principal Investigator:	Kimberly C. Abel

Contract Activity for the
Army Research Institute (ARI)
Virtual Environment Testbed (VETB) Project

Visual Systems Laboratory
VSL Document Number IST.VSL.VET.RPT.0001

SUMMARY REPORT

3D Sound Generation for Virtual Environments

Convolvotron Technical Documentation and MIDIator MIDI Library

Kimberly Abel, Jacquelyn Ford Morie, Michael Tedder, Michael Goslin

May 1994

Research Funded By
Army Research Institute (ARI)
Virtual Environment Testbed (VETB) Project
(Contract # N61339-92-C-0076)

Visual Systems Laboratory
VSL Document Number VSL94.xx

Institute for Simulation and Training
3280 Progress Drive
Orlando, FL 32826

University of Central Florida
Division of Sponsored Research

Contents

Overview: 3D Sound in Virtual Worlds

Adding realistic sound to the visual elements of a virtual environment greatly enhances the believability and immersive qualities of the simulation. The most convincing audio is fully three dimensional, re-creating sound as it is experienced in real life, with qualities of direction, distance, intensity, and depth.

Various types of dimensional sound systems were explored for their potential inclusion in future ARI projects which would require sound. Initial investigations were made with the technical people at SoundDeluxe and Q Sound, and also with audio engineers from United Arts. This led to the conclusions that, while these systems provided a certain level of realism, they were not capable of providing directional sound, necessary for effective training in Virtual Worlds. Several methods have been devised in the audio industry to approximate dimensional sound. Simple stereo sound gives two channels separated by some distance, but does nothing to recreate those sounds that would ordinarily be heard between, above, in front of, or behind these sounds sources. Surround Sound systems basically increase the number of speakers as well as maximizing their placement . However, these systems do not faithfully recreate sounds which are moving between one source and the other. Q Sound is an example of a recent effort by the audio industry to rectify this situation. Utilizing a simple phase-shifting technique, Q Sound gives the illusion of sound traveling from one audio source (speaker) to another. These and other similar methods do not provide directional sound which can be placed anywhere in a virtual space.

To more closely approximate the human hearing system, two things are needed: a way to *precisely* position a sound in 3D space, and a way to recreate the complex filtering mechanism the human ear provides. Additionally, sounds in the real world rarely reach our ears in a "pure" form; they are reflected and absorbed by the various surfaces and materials in an environment, much like light is. Only one of the systems for sound which were explored – the Convolvotron™ – was found to effectively address these aspects of realistic sound. In addition to the dimensional nature of sounds, the Convolvotron also provides mechanisms to include the human ear filter as well as some means to specific acoustic qualities of an environment.

The Convolvotron™

1. Introduction

The Convolvotron is a three dimensional (referenced from hereon as 3D) sound convolution engine, made and marketed by Crystal River Engineering of Groveland, California. Its basic purpose is to accept as many as four audio input sources, locate the sources and the listener in a 3D world, and output a stereo analog signal through a pair of headphones. An audio source sent through the Convolvotron engine will appear as if it is either above, below, in front of, or behind the listener, depending on where it is located. The Convolvotron does this by combining (or mathematically by convolving) the audio signal with information about a location in 3D space, as well as with specific mathematical functions that duplicate the acoustic properties of the human ear.

The Convolvotron is head specific. A sound which seems to be located behind one listener may give the impression that it is located in front of another. The Convolvotron uses **Head Related Transfer Functions** (HRTF) to correct this problem. A HRTF is a binary data file that can be uploaded to the Convolvotron. The Convolvotron will then process the data and create a new phase shifting formula from the data it just received. This data is obtained through a process of "measuring" a specific set of ears through a sequence of audio sounds. Tiny microphones are placed deep in the ears of a person seated in an anechoic chamber. The measurements of the sounds picked up by these microphones are compared to a baseline measurement of the sounds signals. The difference is the HRTF. The HRTF consists of two parts: the scattering caused by the outer ear, or pinna, and the damping effects caused by the traversal of the sound through the ear canal.

The Convolvotron provides an average HRTF for use with the system. The basic two board set is capable of providing 4 anechoic sounds or one echoic sound. The echoic sound can be reflected off other object in the simulation which can be given acoustic properties of wood, metal, etc. Several board sets can be used in tandem to provide the number and type of sounds required.

2. Communicating with the Convolvotron

The Convolvotron runs on a PC by itself. It receives all data via serial ports. Functions have been made to communicate with the Convolvotron and other functions have been added that the Convolvotron itself does not readily support.

At the current time, the Convolvotron communicates at 9600 baud, no parity, 8 data bits and 1 stop bit. The serial cable connecting the PC and the SG carries information for both the Convolvotron and MIDI. (See Hardware Configuration Diagram in Section 8) The actual packets for the Convolvotron are delivered by a "pinging" method. There are two different types of packets.

2.1 Data Packets

Data packets are packets that hold information about the listener and the sound sources. Each time the host system is pinged, it follows this basic logic to minimize transmission time between the two systems:

1. Send the head position
2. For each sound source used in the simulation,

a. If it has not changed position, send a flag saying so and go on to the next source.
b. If the source is a uniform sound source, then only send the X, Y, and Z locations of the source.
c. Otherwise, send the current position and quaternion of the source.

2.2 Special Packets

A special packet is a direct command to the Convolvotron. This includes commands such as the initialization and closing of the Convolvotron. The host system must wait for the Convolvotron to send a "ping" if a special packet is to be called.

3. Programming the Convolvotron

Listed here are predefined functions to use the Convolvotron in the C language. They are currently only compiled for SGIs, however, they are scheduled to be ported over to the PC environment in the near future.

3.1 void Ctron_open(char *device, char nsources, char units)

This call is made to initialize the Convolvotron.

Device is the port to use. The device must be a /dev/ttyd device. The Convolvotron does not support flow control of any kind, therefore, do not use /dev/ttyf or /dev/ttym. (ie: use /dev/ttyd4)

Nsources is the maximum number of sources you will be using in your simulation. The lower this number, the less data that will need to be sent for each source. This will also lessen the number of formulas the Convolvotron will have to create, and hence, the faster the Convolvotron will be able to update within your simulation.

Units is defined as an enumeration in the include file, ctron.h. Units is the distance of the sources in relation to the world. The following units are defined: inches, feet, millimeters, centimeters, meters, and Lunits. An Lunit is the Convolvotron's propietary unit for measuring distance.

3.2 void Ctron_close(char stopsound)

This call is made to close cummunications with the Convolvotron.

Stopsound is a flag that can be set to tell the Convolvotron to quiet all sounds or to leave them running. If stopsound is false (0), the Convolvotron will leave the sounds active. If stopsound is true (1), the Convolvotron will shut them off before uninitialization.

3.3 void Ctron_amplify(char source, float dB)

This call is made to amplify a sound source.

Source is a number from 1 to 4 (the corresponding sound source) to modify.

DB is the amount of decibels to amplify the specified source. The maximum volume for any source is 120 dB.

3.4 void Ctron_model_default(void)

This call resets the Convolvotron's phase-shifting parameters to the default settings.

3.5 void Ctron_model_head(float aural_ocular, float inter_aural, float aural_crown)

This call sets the basic Convolvotron's phase-shifting parameters. It is unknown at this time what units the Convolvotron uses for these parameters, so it is advised not to change them.

3.6 int Ctron_loadHRTF(char *filename)

This call loads a HRTF from the Convolvotron's disk, and uploads it to the Convolvotron for processing. A value is returned from this function as follows:

0	Load proceeded OK
-1	The filename could not be located
-2	The filename specified is an invalid HRTF.

3.7 void Ctron_showmap(void)

This call is made to display a "map" of the location of the listener and each of the sound sources on the Convolvotron's screen.

3.8 void Ctron_locate_head(point6d pq)

This call is made to position the listener's head within the universe.

Pq is defined as a point6d. The include file, ctron.h, defines a point6d to be an array of seven floats. A point6d holds position and quaternion information. The first 3 floats hold the X, Y, and Z position values. The other 4 floats hold the X, Y, Z, and W quaternion values.

3.9 void Ctron_locate_source(char source, point6d pq)

This call is made to position a sound source within the universe.

Source is a number from 1 to 4 of the corresponding sound source to modify its location.

Pq is defined, again, as a point6d. See Ctron_locate_head for more information.

3.10 void Ctron_uniform(char source, point3d p)

This call is made to make a sound source uniform. A uniform sound is a sound that has position only, therefore, a sound cannot be "pointed" a particular direction as Ctron_locate_source does. Making a sound uniform cuts back on processing time for the Convolvotron. Always use uniform sounds where possible.

Source is a number from 1 to 4 of the corresponding sound source to modify its location.

P is defined as a point3d, holding only position information. The first 3 floats are X, Y, and Z values of the position.

3.11 void Ctron_update(void)

This call should be made each time through the simulation loop (ie: each frame). This call will check if the Convolvotron has "pinged" the host system. If it has not, it returns control to the host program. If the Convolvotron has "pinged" the host system, all updated information will be sent to the Convolvotron.

3.12 Point6ds

As stated before, a point6d is an array of 7 floats. Three functions are given for manipulation of point6ds.

3.12.1 point6d_init(point6d)

Point6d_init initializes a point6d. This function is not entirely necessary, as it sets position and quaternion X, Y, and Z values to 0.0, and then the quaternion W value to 1.0. Notice this does not take a pointer to a point6d. Point6d_init is #defined to set each value individually.

3.12.2 point6d_copy(point6d in, point6d out)

Point6d_copy copies one point6d to another. This function is #defined to simply set *out* equal to *in*.

3.12.3 int point6d_equal(point6d pq1, point6d pq2)

Point6d_equal returns true (1) if both point6ds are equal, false (0) if they are not.

4. Low-level Communications

The communications between the host system and the Convolvotron is entirely transparent. This section explains the actual data sent between the two systems.

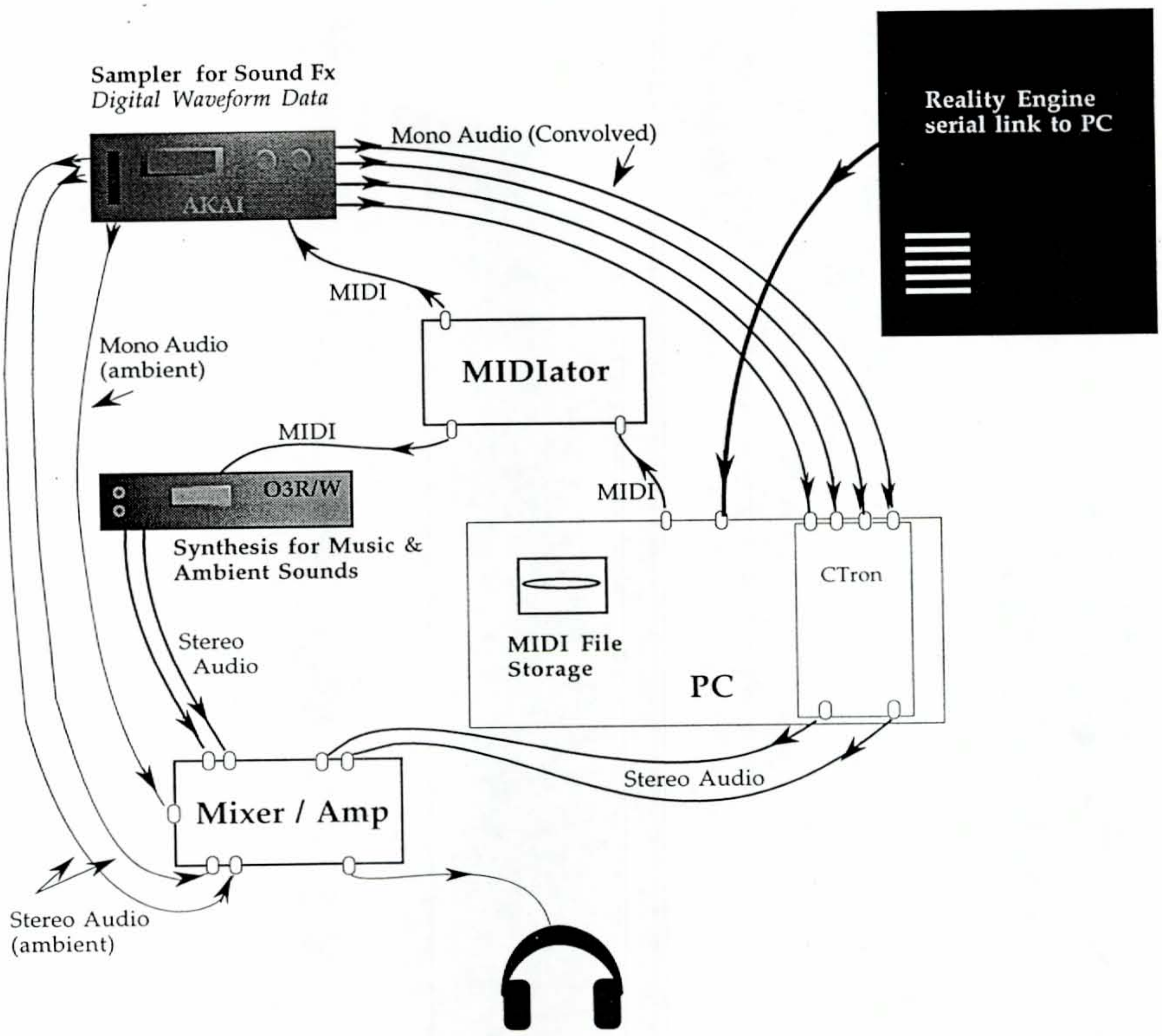
The execution cycle begins by the host system calling Ctron_open(). It will instruct you to bring up the remote PC program. The Convolvotron then sends its "ping" character, and awaits the next cycle from the host system, after it has initialized the Convolvotron. A single data byte is sent as the first character each time through the loop to determine if the the data waiting to be sent is MIDI data or Convolvotron data.

Each data position contains 28 bytes of data. The 28 bytes are formed from seven floats, each float being 4 bytes apiece. At each cycle, the head position is sent. The host system then begins a loop for each sound source that was passed along the Ctron_open() call. The first character for each sound source is a flag saying whether or not the source has changed position. If the source has changed position, it must then send a flag saying whether or not the sound source is uniform. If it is uniform, only the X, Y, and Z position values are sent (4 byte floats * 3 = 12 bytes) to cut back on transmission time. Otherwise, all 28 bytes are passed to the Convolvotron for processing. If, however, the source has not changed position, the host system must then iterate through the loop and send the information regarding the next sound source.

After all data has been sent to the Convolvotron, the host system must then send the letter 'E'. This signifies the end-of-packet byte, per se, for error correction. If this byte is not received, all data that has been received is discarded. The execution cycle continues on this manner until Ctron_close() is called.

5. Additional Notes on the Convolvotron: Echoic Sounds

The Convolvotron also supports another mode in which it can be setup to "sound texture" walls in a small area of 3D space. This mode allows the walls or other large objects in the room to have reflective properties with which the sounds interact. While this provides for more realistic sounds, only one audio source can be utilized in this "echoic" mode in the single Convolvotron configuration we have (though several Convolvotrons can be used in conjunction with one another to provide several echoic sounds if needed.) The libraries for this mode have yet to be implemented.



MIDIator and MIDI Libraries

6. MIDI Calls

MIDI is the basis for creating music or sound that can be processed through the Convolvotron. Since both the MIDI and the Convolvotron work together, the host system must wait for the Convolvotron to be ready before sending any sort of MIDI data.

6.1 void MIDI_PlayFile(char *fil)

This is perhaps the most powerful command offered in the MIDI library. This will load the file specified in fil, (on the SG side, not the PC) and play it automatically in the background. MIDI_PlayFile() currently only supports format 0 and format 1 MIDI files.

6.2 Basic MIDI Output calls

The following functions will describe how to construct music of your own without MIDI files. This may be useful when you wish only one or two notes to be played without the need of creating a MIDI file simply for a few notes, or perhaps playing digital samples with the AKAI digial sampler.

However, a basic understanding of MIDI should be learned before proceeding:

* MIDI works on 16 channels. Calling any of the following functions will require you to enter a channel you wish to modify/use. The AKAI works only on channel 16. The AKAI was set up this way so MIDI files containing background music can be played the same time as digital samples can.

* MIDI uses what is called a 'key scale'. The key value for a few functions are paired up with the keys on a musical keyboard. The value 60 for key stands for "Middle C". This value can range from 24 to 107. (Any value outside this range will be changed to the closest lower or higher value.)

* Velocity is the loudness at which the note should be heard. Medium volume is defined as 64. This value can range from 0 to 127.

6.2.1 int MIDI_NoteOff(char channel, char key)

This function turns a note off.

6.2.2 int MIDI_NoteOn(char channel, char key, char velocity)

This function turns a note on.

6.2.3 int MIDI_PolyPressure(char channel, char key, char pressure)

This function sets a previously turned on note to a new velocity as specified by pressure. Pressure can be higher or lower than the original velocity.

6.2.4 int MIDI_Control(char channel, char function, char value)

This function sets a specific control function to a specified value for only a single channel. Consult MIDI.H for more information on the functions offered.

6.2.5 int MIDI_ChannelMode(char channel, char function)

This function sends a specified function down the specified channel. Again, consult MIDI.H for more information.

6.2.6 int MIDI_Program(char channel, char program)

This function changes the program number on the specified channel. Programs are sometimes referenced as "instruments". Consult the MIDI device you are using for more information on what program you wish to use.

6.2.7 int MIDI_Aftertouch(char channel, char pressure)

This function performs the same function as MIDI_PolyPressure(), except it changes the pressure values for all notes that are on the specified channel.

6.2.8 int MIDI_PitchBend(char channel, char change)

This function alters the pitch of a channel by a specified number of semitones. Change can be either positive or negative.

7. Additional Notes on Development

When the Convolvotron project was first started, we used High C as our compiler for the PC. Serial routines from ARI's joystick source code were used to handle communications between the PC and the SG.

After a few weeks of development, memory problems began to arise in our code. Pointers would be dropped or would point to improper locations, and memory became corrupted which caused a system crash. Mike Goslin and I scanned the code for possible problems, but could not find anything wrong. We then resorted to a PC-Lint checker, which also could find nothing wrong with the code.

I decided to port the code over to Watcom C++, a fairly new compiler. When the conversion was complete, the program ran perfectly without any problems.

This has led me to believe there may be a bug in High C's memory handling, and it should be looked into. I personally will be using Watcom C++ from now on (as a preferred choice) for any projects I may be assigned in the future.

Hardware Configuration

8. Basic Setup

The Convolvotron is operated by commands from a serial port. The SG "talks" to the PC by sending the PC instructions of what to perform.

The cables from the Convolvotron inputs are hooked to the 4 outputs on the Akai digital sound generator. Output 5 from the Akai is used for monoural ambient sounds, and outputs 6 and 7 are used for stereo ambient sounds. The stereo output from the Convolvotron is then hooked up to a mixer. If the PC should receive MIDI data, it then sends the MIDI data out to the MIDIator MIDI box, which either branches off to the Akai or the Korg 03R/W synthesizer. The synthesizer's stereo outputs are then hooked to another input on the mixer. The mixer processes both signals then raises the volume through an amplifier. The output from the amplifier is then heard through a pair of head phones and/or a set of speakers.

References

CyberArts: Exploring Art and Technology,
Edited by Linda Jacobson
Miller Freeman, Inc, San Francisco CA
1992

Section 2 : Music and Sounds (pp 67 - 120)

-Good overview of the Convolvotron and other spatialized sound systems.

Sound and Hearing: A Conceptual Introduction,
R. Duncan Luce,
Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1993

-Source book on hearing and psychoacoustics

Appendix A: PC Code Client

MAIN.CPP (on PC side)

```
---------------------------------

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <dos.h>
#include <stdarg.h>
#include "data.hpp"
#include "atron.h"
#include "ctronlib.hpp"
#include "types.hpp"
#include "main.hpp"
#include "myportal.hpp"

Portal *MIDI, *Remote;
AudioData ad;
uchar *MIDIbuf, done;
int resync_attempts, cursor_x, cursor_y;
unsigned char *ScreenPtr;
unsigned char *Rows[25];

extern short int portbase;

Ctron_Main::Ctron_Main(long baud)
{
   int i;

   // Initialize accessing to the screen memory for text mode.
   for (i = 0; i < 25; i++)
      Rows[i] = (unsigned char *)((0xB800 << 4) + (160 * i));

   cursor_off();

   clear();

   drawline(5);
   drawline(21);
   drawline(23);

   gotoxy(0, 22);
    myprintf("Resync attempts: %-5u", resync_attempts);

   gotoxy(0, 6);
     puts("Beginning Convolvotron initialization...\n");
    // Initialize our MIDI buffer.
    MIDIbuf = new uchar[256];

    // Create our Data object.
    Data = new Ctron_Data;

    // Open our communication lines.
    MIDI->Summon(COM2, 3, 38400, NO_PARITY, 8, 1);
    MIDI->RaiseDTR();
    Remote->Summon(COM1, 4, baud, NO_PARITY, 8, 1);
    Remote->RaiseDTR();

    // Show our status.
    myprintf("\nCOM1 is open at %ld baud for Ctron calls.\n", baud);
    puts("COM2 is open at 38400 baud for MIDI calls.\n\n");

    done = FALSE;
    resync_attempts = 0;

    puts("Ready to receive information... Sending initial ping.\n\n");
}

Ctron_Main::~Ctron_Main(void)
{
    // Kill our MIDI buffer.
    delete []MIDIbuf;

    // Shut down our ports.
    MIDI->Banish();
    Remote->Banish();

    // Kill our object.
    delete Data;

    gotoxy(0, 24);

    cursor_on();
}

// Resync with the remote system if there was an error in the communications.
void Ctron_Main::resync(void)
{
    int i;
    char s[5];

    resync_attempts++;
    gotoxy(0, 22);
    myprintf("Resync attempts: %-5u", resync_attempts);
    gotoxy(56, 22);
    puts("\a[ Resync in progress ]\a");
    while (i = Remote->DataWaiting())
    {
        Remote->GetByte();
        delay(15);
    }
    gotoxy(56, 22);
    puts("                    ");
}

// Display current head & source positions and pass along to the Ctron.
void Ctron_Main::process(void)
{
    int i, t;
    float head[6], src[6];
    union REGS r;

    gotoxy(0, 0);
    puts("Headpos: ");
    for (i = 0; i < 6; i++)
    {
        head[i] = Data->headpos(i);
        myprintf("%5.3f ", head[i]);
    }
    puts("\n");
    ctronlocatehead(head, &ad);
    for (i = 0; i < 4; i++)
    {
        myprintf("Source%d: ", i + 1);
        for (t = 0; t < 6; t++)
        {
            src[t] = Data->sourcepos(i, t);
            myprintf("%5.3f ", src[t]);
        }
        puts("\n");
        if (Data->uniform(i))
            ctronuniform(i, src, &ad);
        else
            ctronlocatesource(i, src, &ad);
    }
    ctronupdate();
}

// Process a special packet if it's being received.
void Ctron_Main::special(void)
{
    char s[80];
    int i1, i2;
    float f1, f2, f3;
    switch (Data->packettype)
    {
      case INIT:                    // Ctron_open()
         Data->getINITpacket(&i1, &i2);
         Data->set_sources(i1);
         ctronopen(i1, i2, &ad);
        break;
      case CLOSE:                   // Ctron_close()
        Data->getCLOSEpacket(&i1);
         ctronclose(i1);
         done = 1;
        break;
      case AMPLIFY_SOURCE:          // Ctron_amplify()
         Data->getAMPLIFYpacket(&i1, &f1);
         ctronamplify(i1, f1);
        break;
      case MODEL_HEAD:              // Ctron_model_head()
        Data->getMODELpacket(&f1, &f2, &f3);
         ctronmodelhead(f1, f2, f3);
        break;
      case LOAD_HRTF:
         Data->getHRTFpacket(s);
         ctronloadHRTF(s);
        break;
      case SHOW_MAP:                // Ctron_showmap()
         ctronshowmap(&ad);
        break;
    }
}

// Send output to the MIDI device.
void Ctron_Main::MIDIout(int len)
{
    uchar *ptr = MIDIbuf;
    short int p = portbase;

    // This is a really ugly hack I had to make in order to get this to work.  Instead of
    // referencing a different serial object (like normal), I had to manually change the
    // output port and set it back.  I had to do this because C++ doesn't interface to
    // DOS interrupts.  This will be fixed in the future (hopefully...).
    portbase = COM2BASE;
    while (len--)
    {
       MIDI->PutByte(*ptr++);
      delay(3);
    }
    portbase = p;
}
```

```
// Sends a ping to the remote system, counts down for a timeout (ie: if the remote
// doesn't respond), and returns an appropriate error code.
int Ctron_Main::send_ping(void)
{
   int timeout, bad;

   bad = 0;

again:
   timeout = 30000;
   Remote->PutByte('P');
   while (!Remote->DataWaiting() && !kbhit())
   {
      if (timeout-- == 0)
      {
         clearline(24);
         gotoxy(0, 24);
         bad++;
         if (bad == 10)
         {
            puts("\a\a\aFatal error: Remote timeout, aborting...");
            return -1;
         }
         puts("\a** WARNING: Remote timed out after packet handshake **");
         goto again;
      }
   }
   return 0;
}

// Receives a packet.
int Ctron_Main::ReceivePacket(void)
{
   int res;

   res = Data->GetPacket();
   if (res == -1)
   {
      clearline(24);
      gotoxy(0, 24);
      puts("\aFailed to receive End-of-Packet byte!\a");
   }
   return res;
}

uchar Ctron_Main::packettype(void)
{
   return Data->packettype;
```

```
}

// This function positions the cursor to a specific location on the screen.
void Ctron_Main::gotoxy(int x, int y)
{
    union REGS r;

    r.h.ah = 2;
    r.h.bh = 0;
    r.h.dl = x;
    r.h.dh = y;
    int386(0x10, &r, &r);

    ScreenPtr = Rows[y] + (x << 1);
    cursor_x = x;
    cursor_y = y;
}

// This function eliminates an entire line on the screen.
void Ctron_Main::clearline(int a)
{
    int i = 80;
    unsigned char *temp;

    temp = Rows[a];
    while (i--)
    {
        *temp++ = ' ';
        *temp++ = 0x07;
    }
}

// This function draws a straight line on the screen.
void Ctron_Main::drawline(int a)
{
    int i = 80;
    unsigned char *temp;

    temp = Rows[a];
    while (i--)
    {
        *temp++ = '-';
        *temp++ = 0x07;
    }
}

// This function clears the screen.
void Ctron_Main::clear(void)
{
    int i;

    ScreenPtr = Rows[0];
    i = 2000;
    while (i--)
    {
        *ScreenPtr++ = ' ';
        *ScreenPtr++ = 0x07;
    }
    gotoxy(0, 0);
}

// This function writes a character to the screen, and beeps if a bell character is
// passed in.
void Ctron_Main::putch(uchar ch)
{
    switch (ch)
    {
        case 7:
            sound(720);
            delay(50);
            nosound();
            break;
        case 10:
        case 13:
            gotoxy(0, cursor_y + 1);
            break;
        default:
            *ScreenPtr++ = ch;
            *ScreenPtr++ = 0x07;
            break;
    }
}

// This function writes an entire string of text to the screen.
void Ctron_Main::puts(char *s)
{
    char *buf = s;

    while (*buf)
        putch(*buf++);
}

// This function handles printf()ing to the screen.
void Ctron_Main::myprintf(char *fmt, ...)
{
    va_list va;
    char buf[160];
    va_start(va, fmt);
    vsprintf(buf, fmt, va);
    va_end(va);

    puts(buf);
}

// This function turns the cursor on.
void Ctron_Main::cursor_on(void)
{
    union REGS r;

    r.h.ah = 1;
    r.h.ch = 0x06;
    r.h.cl = 0x07;
    int386(0x10, &r, &r);
}

// This function turns the cursor off.
void Ctron_Main::cursor_off(void)
{
    union REGS r;

    r.h.ah = 1;
    r.h.ch = 0x26;
    r.h.cl = 0x07;
    int386(0x10, &r, &r);
}

void main(void)
{
    int i, len;
    Ctron_Main *Main;
    uchar ch, *ptr;

    MIDI = new Portal;
    Remote = new Portal;

    // Create our Main Ctron class (at 9600 baud).
    Main = new Ctron_Main(9600);

    do
    {
        if (Main->send_ping() == -1)
        {
            done = 1;
            break;
        }
        if (kbhit())
        {
          if (getch() == 27)
            done = 1;
        }
        else
        {
          // See if this is a MIDI packet or a Ctron packet.
          ch = Remote->GetByte();

          switch (ch)
          {
            case 1:
              Main->gotoxy(0, 24);

              // Find out length of packet.
              len = Remote->GetByte();
              if (len != 0)
              {
                Main->clearline(24);
                Main->gotoxy(0, 24);
                Main->myprintf("Last transmitted MIDI packet:  Length %d, bytes left",
len);
                i = len;
                ptr = MIDIbuf;
                while (i--)
                {
                  Main->myprintf("(%d) ", i);
                  *ptr = Remote->GetByte();
                  *ptr++;
                }

                Main->MIDIout(len);
              }
              break;
            case 2:
              if (Main->ReceivePacket() == -1)   // Error in receive
                Main->resync();
              else
              {
                if (Main->packettype() == UPDATE)
                  Main->process();
                else
                  Main->special();
              }
              break;
            default:
              Main->resync();
              break;
          }
        }
    } while (!done);

    // End it all.
    delete Main;
    delete MIDI;
    delete Remote;
}
```

MAIN.HPP (on PC side)

```
#ifndef MAIN_HPP
#define MAIN_HPP

class Ctron_Main
{
private:
    Ctron_Data *Data;

public:
    Ctron_Main(long baud, int sources);
    ~Ctron_Main(void);

    void resync(void);
    void process(void);
    void special(void);
    void MIDIout(int len);
    int  send_ping(void);

    int  ReceivePacket(void);
    uchar packettype(void);

    void gotoxy(int x, int y);
    void clearline(int a);
    void drawline(int a);

    void clear(void);
    void putch(uchar ch);
    void puts(char *s);
    void myprintf(char *fmt, ...);

    void cursor_on(void);
    void cursor_off(void);
};

#endif
```

DATA.CPP (on PC side)

```
--------------------------------

#include <stdio.h>
#include <math.h>
#include <mem.h>
#include <conio.h>
#include "data.hpp"

Ctron_Data::Ctron_Data()
{
   int i, j;

   // Initialize our data structures.
   numsrc = 0;

   for (j = 0; j < 6; j++)
   {
      recv.head[j] = 0.0;
      ctron.head[j] = 0.0;
   }
   recv.head[6] = 1.0;

   for (i = 0; i < 4; i++)
   {
      for (j = 0; j < 6; j++)
      {
         recv.src[i].loc[j] = 0.0;
         ctron.src[i].loc[j] = 0.0;
      }
      recv.src[i].loc[6] = 1.0;
      recv.src[i].uniform = FALSE;
      recv.src[i].changed = FALSE;
   }
}

Ctron_Data::~Ctron_Data()
{
   // Do absolutely nothing.
}

// This function translates a World ToolKit WTpq into the Convolvotron's internal
// coordinate system.
void Ctron_Data::to_ctron(point6d from, point6d *to)
{
   int i, QX, QY, QZ, QW;
   point6d temp;

// CX = WZ
// CY = -WX
// CZ = -WY

   temp[0] = from[2];
   temp[1] = -from[0];
   temp[2] = -from[1];

// 3 = QX, 4 = QY, 5 = QZ, 6 = QW

   QX = 5;  // Quat X = WTK Quat Z
   QY = 3;  // Quat Y = (-) WTK Quat X
   QZ = 4;  // Quat Z = (-) WTK Quat Y

   from[QY] = -from[QY];  // Take care of negatives here
   from[QZ] = -from[QZ];

   QW = 6;  // QW stays the same

   temp[3] = asin(2 * ((from[QY] * from[QZ]) - (from[QX] * from[QW])));
   temp[4] = atan((-2 * ((from[QY] * from[QW]) + (from[QX] * from[QZ]))) /
           ((from[QZ] * from[QX]) + (from[QY] * from[QY]) -
            (from[QZ] * from[QZ]) - (from[QW] * from[QW])));
   temp[5] = atan((-2 * ((from[QX] * from[QY]) + (from[QZ] * from[QW]))) /
           ((from[QX] * from[QX]) - (from[QY] * from[QY]) +
            (from[QZ] * from[QZ]) - (from[QW] * from[QW])));

   memcpy(to, &temp, sizeof(point6d));
}

// This function returns a head position.
float Ctron_Data::headpos(int num)
{
   return ctron.head[num];
}

// This function returns a source's position.
float Ctron_Data::sourcepos(uchar source, int num)
{
   return ctron.src[source].loc[num];
}

// This function sets the number of sources used in the simulation.
void Ctron_Data::set_sources(uchar num)
{
   numsrc = num;
}

// This function returns the number of sources used.
uchar Ctron_Data::get_sources(void)
```

```
{
    return numsrc;
}

// This function receives an initialization packet.
void Ctron_Data::getINITpacket(int *srcs, int *units)
{
   int i;

   i = Remote->GetByte(); *srcs = i;
   i = Remote->GetByte(); *units = i;
}

// This function receives a close packet.
void Ctron_Data::getCLOSEpacket(int *killsnd)
{
   int i;

   i = Remote->GetByte(); *killsnd = i;
}

// This function receives an amplification packet.
void Ctron_Data::getAMPLIFYpacket(int *src, float *dB)
{
   char s[5];
   int i;

   i = Remote->GetByte(); *src = i;
   Remote->GetStr(s, sizeof(float));
   str2float(s, dB);
}

// This function receives a model head packet.
void Ctron_Data::getMODELpacket(float *aural_ocular, float *inter_aural, float
*aural_crown)
{
   char s[5];

   Remote->GetStr(s, sizeof(float)); str2float(s, aural_ocular);
   Remote->GetStr(s, sizeof(float)); str2float(s, inter_aural);
   Remote->GetStr(s, sizeof(float)); str2float(s, aural_crown);
}

// This function receives a HRTF packet.
void Ctron_Data::getHRTFpacket(char *fname)
{
   int t;
   char *buf = fname;
```

```
    t = Remote->GetByte();
    while (t--)
        *buf++ = Remote->GetByte();
}

// This function converts all received WTpqs into Ctron coordinates.
void Ctron_Data::convert(void)
{
    int i;

    to_ctron(recv.head, &ctron.head);
    for (i = 0; i < 4; i++)
        to_ctron(recv.src[i].loc, &ctron.src[i].loc);
}

// This function receives an entire packet.
int Ctron_Data::GetPacket(void)
{
    char s[10], ch, i, y, j;

    packettype = Remote->GetByte();
    if (packettype == UPDATE)
    {
        for (i = 0; i < 7; i++)
        {
            Remote->GetStr(s, 4);
            str2float(s, &recv.head[i]);
        }
        for (i = 0; i < numsrc; i++)
        {
            recv.src[i].changed = Remote->GetByte();
            if (recv.src[i].changed)
            {
                recv.src[i].uniform = Remote->GetByte();
                if (recv.src[i].uniform)
                    j = 3;          // If uniform, read 3 floats: X, Y, Z
                else
                    j = 7;          // else read all 7.
                for (y = 0; y < j; y++)
                {
                    Remote->GetStr(s, 4);
                    str2float(s, &recv.src[i].loc[y]);
                }
            }
        }
        if (Remote->GetByte() != 'E') return -1;
        convert();
    }
    return 0;
```

```
}

// This function returns the uniformity of a source.  (TRUE or FALSE)
uchar Ctron_Data::uniform(char source)
{
    return recv.src[source].uniform;
}

// This function converts a 4-byte string into a float.
void Ctron_Data::str2float(char *s, float *f)
{
   memcpy(f, s, sizeof(float));
}
```

DATA.HPP (on PC side)

```
#ifndef DATA_HPP
#define DATA_HPP

#include "myportal.hpp"
#include "types.hpp"

extern Portal *MIDI, *Remote;

typedef struct
{
   point6d loc;
   uchar   uniform;
   uchar   changed;
} srcinfo;

typedef struct
{
   point6d head;
   srcinfo src[4];
} pos_info;

class Ctron_Data
{
private:
   pos_info recv, ctron;
   uchar numsrc;

   // Converts WTK to Ctron coordinates.
   void  to_ctron(point6d from, point6d *to);
   void  convert(void);
public:
  uchar packettype;

  Ctron_Data();
  ~Ctron_Data();

   // Returns converted Ctron coordinates.
   float headpos(int num);
   float sourcepos(uchar source, int num);

   void  set_sources(uchar num);
  uchar get_sources(void);

   void  point6d_init(point6d *p);
   void  getINITpacket(int *srcs, int *units);
   void  getCLOSEpacket(int *killsnd);
   void  getAMPLIFYpacket(int *src, float *dB);
   void  getMODELpacket(float *aural_ocular, float *inter_aural, float *aural_crown);
   void  getHRTFpacket(char *fname);
  int   GetPacket(void);
   uchar uniform(char source);
   void  str2float(char *s, float *f);
};

#endif
```

MYPORTAL.CPP (on PC side)

```
#include <stdio.h>
#include <conio.h>
#include <mem.h>
#include <dos.h>
#include "myportal.hpp"

unsigned short int Qrear;
unsigned short int Qfront;
short int portbase;
char *combuf;
char inited = 0;

static void interrupt commhandler();

//   The 8250 UART has 10 registers accessible through 7 port addresses.
//   Here are their addresses relative to COM1BASE and COM2BASE. Note
//   that the baud rate registers, (DLL) and (DLH) are active only when
//   the Divisor-Latch Access-Bit (DLAB) is on. The (DLAB) is bit 7 of
//   the (LCR).
```

```
//
//   o TXR Output data to the serial port.
//   o RXR Input data from the serial port.
//   o LCR Initialize the serial port.
//   o IER Controls interrupt generation.
//   o IIR Identifies interrupts.
//   o MCR Send control signals to the modem.
//   o LSR Monitor the status of the serial port.
//   o MSR Receive status of the modem.
//   o DLL Low byte of baud rate divisor.
//   o DHH High byte of baud rate divisor.
#define TXR        0    /* Transmit register (WRITE) */
#define RXR        0    /* Receive register  (READ)  */
#define IER        1    /* Interrupt Enable          */
#define IIR        2    /* Interrupt ID              */
#define LCR        3    /* Line control              */
#define MCR        4    /* Modem control             */
#define LSR        5    /* Line Status               */
#define MSR        6    /* Modem Status              */
#define DLL        0    /* Divisor Latch Low         */
#define DLH        1    /* Divisor latch High        */

//-------------------------------------------------------------
// Bit values held in the Line Control Register (LCR).
//  bit   meaning
//  ---   -------
//  0-1   00=5 bits, 01=6 bits, 10=7 bits, 11=8 bits.
//  2     Stop bits.
//  3     0=parity off, 1=parity on.
//  4     0=parity odd, 1=parity even.
//  5     Sticky parity.
//  6     Set break.
//  7     Toggle port addresses.
//-------------------------------------------------------------

//-------------------------------------------------------------
// Bit values held in the Line Status Register (LSR).
//  bit   meaning
//  ---   -------
//  0     Data ready.
//  1     Overrun error - Data register overwritten.
//  2     Parity error - bad transmission.
//  3     Framing error - No stop bit was found.
//  4     Break detect - End to transmission requested.
//  5     Transmitter holding register is empty.
//  6     Transmitter shift register is empty.
//  7        Time out - off line.
//-------------------------------------------------------------
#define RCVRDY      0x01
```

```
#define OVRERR      0x02
#define PRTYERR     0x04
#define FRMERR      0x08
#define BRKERR      0x10
#define XMTRDY      0x20
#define XMTRSR      0x40
#define TIMEOUT     0x80

//------------------------------------------------------------
//  Bit values held in the Modem Output Control Register (MCR).
//   bit    meaning
//   ---    -------
//   0      Data Terminal Ready. Computer ready to go.
//   1      Request To Send. Computer wants to send data.
//   2      auxillary output #1.
//   3      auxillary output #2.(Note: This bit must be
//          set to allow the communications card to send
//          interrupts to the system)
//   4      UART ouput looped back as input.
//   5-7    not used.
//------------------------------------------------------------
#define DTR         0x01
#define RTS         0x02
#define MC_INT      0x08

//------------------------------------------------------------
//  Bit values held in the Modem Input Status Register (MSR).
//   bit    meaning
//   ---    -------
//   0      delta Clear To Send.
//   1      delta Data Set Ready.
//   2      delta Ring Indicator.
//   3      delta Data Carrier Detect.
//   4      Clear To Send.
//   5      Data Set Ready.
//   6      Ring Indicator.
//   7      Data Carrier Detect.
//------------------------------------------------------------
#define CTS         0x10
#define DSR         0x20
#define RI          0x40
#define DCD         0x80

//------------------------------------------------------------
//  Bit values held in the Interrupt Enable Register (IER).
//   bit    meaning
//   ---    -------
//   0      Interrupt when data received.
//   1      Interrupt when transmitter holding reg. empty.
//  2     Interrupt when data reception error.
//  3     Interrupt when change in modem status register.
//  4-7   Not used.
//----------------------------------------------------------------
#define RX_INT       0x01

//----------------------------------------------------------------
// Bit values held in the Interrupt Identification Register (IIR).
//  bit   meaning
//  ---   -------
//  0     Interrupt pending
//  1-2   Interrupt ID code
//        00=Change in modem status register,
//        01=Transmitter holding register empty,
//        10=Data received,
//        11=reception error, or break encountered.
//  3-7   Not used.
//----------------------------------------------------------------
#define RX_ID        0x04
#define RX_MASK      0x07

// These are the port addresses of the 8259 Programmable Interrupt
// Controller (PIC).
#define IMR          0x21  /* Interrupt Mask Register port */
#define ICR          0x20  /* Interrupt Control Port      */

// An end of interrupt needs to be sent to the Control Port of
// the 8259 when a hardware interrupt ends.
#define EOI          0x20  /* End Of Interrupt */

// The (IMR) tells the (PIC) to service an interrupt only if it
// is not masked (FALSE).
#define IRQ0         0xFE
#define IRQ1         0xFD
#define IRQ2         0xFB
#define IRQ3         0xF7  /* COM2 or COM4 (standard) */
#define IRQ4         0xEF  /* COM1 or COM3 (standard) */
#define IRQ5         0xDF
#define IRQ6         0xBF
#define IRQ7         0x7F

// The actual interrupt vector our function needs to be installed on is the
// IRQ + 8h.
#define VECTOR0      0x08
#define VECTOR1      0x09
#define VECTOR2      0x0A
#define VECTOR3      0x0B
#define VECTOR4      0x0C
#define VECTOR5      0x0D
```

```
#define VECTOR6       0x0E
#define VECTOR7       0x0F


// Again, this class was incredibly hacked in order to make it work properly.  Until I
// can figure out an interface for this class to work "object-orientedly", the current
// setup will have to do.

int oldIMR;

Portal::Portal(void)
{
   if (!inited)
   {
      combuf = new char[65535];
      inited = 1;
   }
   Qfront = 0;
   Qrear  = 0;
   wait_for_CTS = 0;

   _dos_setvect(0x0B, commhandler);
   _dos_setvect(0x0C, commhandler);

   Portal_IRQ = IRQ4;

   oldIMR = inp(IMR);
}

Portal::~Portal(void)
{
   if (inited)
   {
      delete []combuf;
      inited = 0;
   }

   outp(IMR, oldIMR);
}

void Portal::set_port(int s_port)
{
   switch (s_port)
   {
      case COM1: portbase = COM1BASE; break;
      case COM2: portbase = COM2BASE; break;
      case COM3: portbase = COM3BASE; break;
      case COM4: portbase = COM4BASE; break;
   }
```

```
    Portal_Port = s_port;
}

void Portal::set_baud(long s_baud)
{
    char c;
    int divisor;

    if (s_baud == 0)
        return;     /* Avoid divide by zero */

    divisor = (int) (115200L / s_baud);

    _disable();
    c = inp(portbase + LCR);
    outp(portbase + LCR, (c | 0x80));       /* Set DLAB */
    outp(portbase + DLL, (divisor & 0x00FF));
    outp(portbase + DLH, ((divisor >> 8) & 0x00FF));
    outp(portbase + LCR, c);                /* Reset DLAB */
    _enable();

    Portal_Baud = s_baud;
}

void Portal::set_bits(int s_parity, int s_databits, int s_stopbits)
{
    int  setting;

    setting  = s_databits - 5;
    setting  |= ((s_stopbits == 1) ? 0x00 : 0x04);
    setting  |= s_parity;

    _disable();
    outp(portbase + LCR, setting);
    _enable();

    Portal_Parity   = s_parity;
    Portal_DataBits = s_databits;
    Portal_StopBits = s_stopbits;
}

long Portal::get_baud(void)
{
    return Portal_Baud;
}

int Portal::get_parity(void)
{
    return Portal_Parity;
```

```
}

int Portal::get_databits(void)
{
   return Portal_DataBits;
}

int Portal::get_stopbits(void)
{
   return Portal_StopBits;
}

void Portal::Summon(int s_port, int s_IRQ, long s_baud, int s_parity, int s_databits,
int s_stopbits)
{
   int c;

   set_port(s_port);
   set_baud(s_baud);
   set_bits(s_parity, s_databits, s_stopbits);

   _disable();
   c = inp(portbase + MCR) | MC_INT;
   outp(portbase + MCR, c);
   outp(portbase + IER, RX_INT);
   c = inp(IMR) & Portal_IRQ;
   outp(IMR, c);
   _enable();

   c = inp(portbase + MCR) | RTS | DTR;
   outp(portbase + MCR, c);
}

void Portal::UseCTS(char ch)
{
   if (ch)
      wait_for_CTS = 1;
   else
      wait_for_CTS = 0;
}

void Portal::SetRTS(char ch)
{
   unsigned int p = portbase + MCR;

   if (ch)
      outp(p, inp(p) | RTS);
   else
      outp(p, inp(p) & ~RTS);
```

```
}

void Portal::Banish(void)
{
   // Turn off COMM interrupts
   _disable();
   outp(portbase + IER, 0);
   outp(portbase + MCR, 0);
   _enable();
}

int Portal::DataWaiting(void)
{
//    return (!(Qfront == Qrear));

// Changed to show how many characters are waiting in the buffer, not just a TRUE
// or FALSE flag.
   return (Qfront - Qrear);
}

unsigned char Portal::GetByte(void)
{
   while (!DataWaiting());
   return combuf[Qrear++];
}

void Portal::PutByte(unsigned char byte)
{
   outp(portbase + MCR, MC_INT | RTS | DTR);

   if (wait_for_CTS)
   {
      /* Wait for Clear To Send from modem */
      while ((inp(portbase + MSR) & CTS) == 0);
   }

   /* Wait for transmitter to clear */
   while ((inp(portbase + LSR) & XMTRDY) == 0);

   _disable();
   outp(portbase + TXR, byte);
   _enable();
}

void Portal::StuffByte(unsigned char byte)
{
   combuf[Qfront] = byte;
   Qfront++;
}
```

```
int Portal::CarrierDetected(void)
{
   if (inp(portbase + MSR) & DCD)
      return 1;
   return 0;
}

void Portal::RaiseDTR(void)
{
   outp(portbase + MCR, 0x03);
}

void Portal::DropDTR(void)
{
   outp(portbase + MCR, 0x00);
}

void Portal::FlushBuffer(void)
{
   Qfront = 0;
   Qrear = 0;
   memset(combuf, 0, 65535);
}

void Portal::GetStr(char *s, int len)
{
   char *buf = s;

   while (len--)
      *buf++ = GetByte();
}

static void interrupt commhandler()
{
   _disable();

   if ((inp(portbase + IIR) & RX_MASK) == RX_ID)
   {
      combuf[Qfront] = inp(portbase + RXR);
      Qfront++;
   }

   outp(0x20, 0x20);

   _enable();
}
```

MYPORTAL.HPP (on PC side)

```
----------------------------------------

#ifndef MYPORTAL_HPP
#define MYPORTAL_HPP

#define COM1          1
#define COM2          2
#define COM3          3
#define COM4          4

// COMPORT addresses
#define COM1BASE      0x3F8  /* Base port address for COM1 */
#define COM2BASE      0x2F8  /* Base port address for COM2 */
#define COM3BASE      0x3E8  /* Base port address for COM3 */
#define COM4BASE      0x2E8  /* Base port address for COM4 */

#define NO_PARITY     0x00
#define EVEN_PARITY    0x18
#define ODD_PARITY    0x08

class Portal
{
private:
   int  Portal_Port;
   int  Portal_IRQ;
   long Portal_Baud;
   int  Portal_Parity;
   int  Portal_DataBits;
   int  Portal_StopBits;

   int  vector;
   int  wait_for_CTS;

   void set_port(int  s_port);
   void set_baud(long s_baud);
   void set_bits(int  s_parity, int s_databits, int s_stopbits);

public:
   // Constructor and Destructor
   Portal(void);
   ~Portal(void);

   long get_baud(void);
   int  get_parity(void);
   int  get_databits(void);
   int  get_stopbits(void);

   void add_to_buf(char ch);
```

```
    short int get_portbase(void);

    void Summon(int s_port, int s_IRQ, long s_baud, int s_parity, int s_databits, int
s_stopbits);
    void Banish(void);

    int  DataWaiting(void);
    unsigned char GetByte(void);
    void PutByte(unsigned char byte);

    void StuffByte(unsigned char byte);

    void UseCTS(char ch);
    void SetRTS(char ch);
    void RaiseDTR(void);
    void DropDTR(void);
    int  CarrierDetected(void);
    void FlushBuffer(void);

    void GetStr(char *s, int len);
};

#endif


TYPES.HPP (on PC side)
---------------------------------

#ifndef TYPES_HPP
#define TYPES_HPP

#define FALSE   0
#define TRUE    1

typedef unsigned char   uchar;
typedef float           point6d[7];

enum  CtronCommands
{
   INIT, UPDATE, CLOSE, MODEL_HEAD, LOCATE_HEAD,
    LOCATE_SOURCE, AMPLIFY_SOURCE, SHOW_MAP, LOAD_HRTF
};

#endif
```

Appendix B: SGI Code

CTRON.C (on remote side)

```
------------------------------------

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <malloc.h>
#include "ctron.h"
#include "../include/serial.h"

cinformation cinfo;

char MIDIdata[10];
int  MIDI_nomerge = 0;
long  MIDI_currtime = 0L;
int  MIDI_division=0;
long  MIDI_tempo=500000;
float MIDI_lasttime = 0.0;
long  MIDI_toberead = 0L;
FILE *MIDI_file;
char *MIDI_msgbuff = NULL;
int  MIDI_msgsize = 0;
int  MIDI_msgindex = 0;
char  MIDI_playing = 0;

/* This function converts a float into a 4-byte string.  Note that the Kubota does not
byte-swap like an SG does. */
void float2str(float f, char *s)
{
#ifndef KUBOTA
        /* float = four bytes, do byte-swapping for the pc */
   void *temp = &f;
   char *pc = temp;

  s[0] = pc[3];
  s[1] = pc[2];
  s[2] = pc[1];
  s[3] = pc[0];
#else
   memcpy(s, &f, sizeof(float));
#endif
}

void Ctron_open(char *device, char nsources, char units)
{
    printf("Opening Convolvotron...\n");
    openSerialPort(device, CONV);
    printf("Bring up Remote PC now.\n");
    while (!Serial_datawaiting(CONV));
    printf("Flushing buffer...\n");
    Serial_flush(CONV);
    Serial_putchar( 2, CONV);
    Serial_putchar( INIT, CONV );          /* Special Packet: Ctron_open() */
    Serial_putchar( nsources, CONV );
    Serial_putchar( units, CONV );
    Cinfo_init( nsources );
}

void Ctron_close(char stopsound)
{
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 2, CONV);
    Serial_putchar( CLOSE, CONV );        /* Special Packet: Ctron_close() */
    Serial_putchar( stopsound, CONV );
#ifndef KUBOTA
   sginap(20);
#else
   sleep(1);
#endif
    closeSerialPort(CONV);
}

void Ctron_amplify(char source, float dB)
{
   char s[5];

   source--;
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 2, CONV);
    Serial_putchar( AMPLIFY_SOURCE, CONV); /* Special Packet: Ctron_amplify() */
    Serial_putchar(source, CONV);
    float2str(dB, s);
    writeSerialPort(s, 4, CONV);
}

void Ctron_model_default(void)
{
    Ctron_model_head(0.0, 7.0, -7.5);
}

void Ctron_model_head(float aural_ocular, float inter_aural, float aural_crown)
{
   char s[5];
```

```
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 2, CONV);
    Serial_putchar(MODEL_HEAD, CONV);      /* Special Packet: Ctron_model_head()
*/
    float2str(aural_ocular, s);
    writeSerialPort(s, 4, CONV);
    float2str(inter_aural, s);
    writeSerialPort(s, 4, CONV);
    float2str(aural_crown, s);
    writeSerialPort(s, 4, CONV);
}

int Ctron_loadHRTF(char *filename)
{
    int t;

    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 2, CONV);
    Serial_putchar(LOAD_HRTF, CONV);       /* Special Packet: Ctron_loadHRTF()
*/
    t=strlen(filename);
    Serial_putchar(t, CONV);
    while (t--)
        Serial_putchar(*filename++, CONV);
    while (!Serial_datawaiting(CONV));
    return Serial_getchar(CONV);
}

void Ctron_showmap(void)
{
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 2, CONV);
    Serial_putchar( SHOW_MAP, CONV);       /* Special Packet: Ctron_showmap() */
}

void Ctron_locate_head( point6d pq )
{
    point6d_copy( pq, cinfo.head );
}

void Ctron_locate_source(char source, point6d pq)
{
    source--;
    cinfo.src[source].uniform = FALSE;
    if ( !point6d_equal( cinfo.src[source].location, pq ))
    {
       point6d_copy( pq, cinfo.src[source].location );
       cinfo.src[source].changed = TRUE;
    }
}

void Ctron_uniform(char source, point3d p)
{
    point6d pq;

    source--;
    point6d_init( pq );
    pq[0] = p[0];
    pq[1] = p[1];
    pq[2] = p[2];
    if ( !point6d_equal( cinfo.src[source].location, pq ))
    {
       point6d_copy( pq, cinfo.src[source].location );
       cinfo.src[source].changed = TRUE;
    }
    cinfo.src[source].uniform = TRUE;
}

void Ctron_output( point6d pq )
{
    char s[5];
    int i;

    for (i = 0; i < 7; i++ )
    {
         float2str( pq[i], s );
         writeSerialPort( s, 4, CONV );
    }
}

void Ctron_uni_output( point6d pq )
{
    char s[5];
    int i;

    for (i = 0; i < 3; i++)
    {
       float2str(pq[i], s);
       writeSerialPort(s, 4, CONV);
    }
}

void Ctron_update(void)
{
   int i;

   /* Check if remote PC is ready to receive information */
   if (!Serial_datawaiting(CONV))
        return;

   Serial_flush( CONV );
   Serial_putchar( 2, CONV);
        /* signal ctron that update packet is coming */
   Serial_putchar( UPDATE, CONV );
        /* send head point6d info */
   Ctron_output( cinfo.head );

      /* up to four sound sources specified in init function */
   for (i = 0; i < 4; i++)
   {
        Serial_putchar( cinfo.src[i].changed, CONV );
        /* don't send data if position is unchanged */
        if ( cinfo.src[i].changed )
        {
           cinfo.src[i].changed = FALSE;
           Serial_putchar( cinfo.src[i].uniform, CONV );
           if ( !cinfo.src[i].uniform )
                Ctron_output(cinfo.src[i].location);
        else
            Ctron_uni_output(cinfo.src[i].location);
        }
   }
   Serial_putchar('E', CONV);             /* Let the remote know we're done */
}

void Cinfo_init( char sources )
{
   int i;

   point6d_init( cinfo.head );
   cinfo.numsrc = sources;
   for (i = 0; i < 4; i++)
   {
      point6d_init( cinfo.src[i].location );
        cinfo.src[i].uniform = FALSE;
        cinfo.src[i].changed = TRUE;
   }
}

int point6d_equal(point6d a, point6d b)
{
   int i;
```

```
    for (i=0; i<7; i++)
        if (a[i] != b[i])
            return (0);
    return (1);
}

void MIDI_NoteOff(char channel, char key)
{
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 1, CONV);       /* MIDI packet */
    Serial_putchar( 3, CONV);       /* Packet length of 3 bytes */
    Serial_putchar(0x80+(channel-1), CONV);
    Serial_putchar(key, CONV);
    Serial_putchar(64, CONV);
}

void MIDI_NoteOn(char channel, char key, char velocity)
{
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 1, CONV);
    Serial_putchar( 3, CONV);
    Serial_putchar(0x90+(channel-1), CONV);
    Serial_putchar(key, CONV);
    Serial_putchar(velocity, CONV);
}

void MIDI_PolyPressure(char channel, char key, char pressure)
{
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 1, CONV);
    Serial_putchar( 3, CONV);
    Serial_putchar(0xA0+(channel-1), CONV);
    Serial_putchar(key, CONV);
    Serial_putchar(pressure, CONV);
}

void MIDI_Control(char channel, char function, char value)
{
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 1, CONV);
    Serial_putchar( 3, CONV);
    Serial_putchar(0xB0+(channel-1), CONV);
    Serial_putchar(function, CONV);
    Serial_putchar(value, CONV);
}
```

```
void MIDI_ChannelMode(char channel, char function)
{
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 1, CONV);
    Serial_putchar( 3, CONV);
    Serial_putchar(0xB0+(channel-1), CONV);
    switch (function)
    {
        case RESET_ALL_CONTROLS:
            Serial_putchar(0x79, CONV);
            Serial_putchar(0x00, CONV);
            break;
        case LOCAL_CONTROL_ON:
            Serial_putchar(0x7A, CONV);
            Serial_putchar(0x7F, CONV);
            break;
        case LOCAL_CONTROL_OFF:
            Serial_putchar(0x7A, CONV);
            Serial_putchar(0x00, CONV);
            break;
        case ALL_NOTES_OFF:
            Serial_putchar(0x7B, CONV);
            Serial_putchar(0x00, CONV);
            break;
        case OMNI_MODE_OFF:
            Serial_putchar(0x7C, CONV);
            Serial_putchar(0x00, CONV);
            break;
        case OMNI_MODE_ON:
            Serial_putchar(0x7D, CONV);
            Serial_putchar(0x00, CONV);
            break;
        case MONO_MODE_ON:
            Serial_putchar(0x7E, CONV);
            Serial_putchar(0x00, CONV);
            break;
        case MONO_MODE_OFF:
            Serial_putchar(0x7F, CONV);
            Serial_putchar(0x00, CONV);
            break;
    }
}

void MIDI_Program(char channel, char program)
{
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 1, CONV);
    Serial_putchar( 2, CONV);
    Serial_putchar(0xC0+(channel-1), CONV);
    Serial_putchar(program, CONV);
}

void MIDI_Aftertouch(char channel, char pressure)
{
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 1, CONV);
    Serial_putchar( 2, CONV);
    Serial_putchar(0xD0+(channel-1), CONV);
    Serial_putchar(pressure, CONV);
}

void MIDI_PitchBend(char channel, char change)
{
    while (!Serial_datawaiting(CONV));
    Serial_flush(CONV);
    Serial_putchar( 1, CONV);
    Serial_putchar( 2, CONV);
    Serial_putchar(0xE0+(channel-1), CONV);
    Serial_putchar(change, CONV);
}

float MIDI_ticks2sec(long ticks, int division, int tempo)
{
    return ((float) (((float)(ticks) * (float)(tempo)) / ((float)(division) * 1000000.0)));
}

void MIDI_dotime(void)
{
   float a,g;

    a=MIDI_ticks2sec(MIDI_currtime, MIDI_division, MIDI_tempo);
#ifdef DEBUG_MIDI
    printf("Time=%f   ",a);
#endif
    g=a-MIDI_lasttime;
    MIDI_lasttime=a;
   if (g != 0.0)
   {
        g*=100;
#ifdef DEBUG_MIDI
        printf("(resting %3.2f seconds)   ",g/100);
#endif
#ifndef KUBOTA
        sginap((long)g);
#else
        sleep((long)g);
#endif
   }
}

void MIDI_PlayFile(char *fil)
{
   MIDI_playing=1;
#ifndef KUBOTA
    sproc((void (*)(void *))MIDI_BeginPlay, 0, fil);
#endif
}

void MIDI_BeginPlay(char *fil)
{
    MIDI_file=fopen(fil, "r");
   MIDI_readheader();
   while (MIDI_readtrack());
   fclose(MIDI_file);
   MIDI_playing=0;
}

int MIDI_readmt(char *s)
{
   char buff[32];
   int n = 0;
   char *p = s;
   int c;

   while ( n++<4 && (c=fgetc(MIDI_file)) != EOF )
   {
        if ( c != *p++ )
        {
           strcpy(buff,"expecting ");
           strcat(buff,s);
           MIDI_error(buff);
        }
   }
   return(c);
}

int MIDI_getc(void)
{
   int c;

   c=fgetc(MIDI_file);
   if (c == EOF)
        MIDI_error("premature EOF");
   MIDI_toberead--;
   return(c);
}


void MIDI_readheader(void)
{
   int format, ntrks;

   if (MIDI_readmt("MThd") == EOF)
        return;

   MIDI_toberead = MIDI_read32bit();
   format = MIDI_read16bit();
   ntrks = MIDI_read16bit();
   MIDI_division = MIDI_read16bit();
#ifdef DEBUG_MIDI
    printf("Header format=%d ntrks=%d MIDI_division=%d\n", format, ntrks,
MIDI_division);
#endif
   while (MIDI_toberead > 0)
        MIDI_getc();
}

int MIDI_readtrack(void)
{
   static int chantype[] =
   {
        0, 0, 0, 0, 0, 0, 0, 0,          /* 0x00 through 0x70 */
        2, 2, 2, 2, 1, 1, 2, 0           /* 0x80 through 0xf0 */
   };
   long lookfor;
   int c, c1, type;
   int MIDI_sysexcontinue=0;
   int running=0;
   int status=0;
   int needed;

   if (MIDI_readmt("MTrk") == EOF)
        return(0);

   MIDI_toberead = MIDI_read32bit();
   MIDI_currtime = 0;
#ifdef DEBUG_MIDI
    printf("Track start\n");
#endif
   MIDI_lasttime=0;

   while (MIDI_toberead > 0)
```

```
{
    MIDI_currtime += MIDI_readvarinum();
    c = MIDI_getc();
    if (MIDI_sysexcontinue && c != 0xf7)
       MIDI_error("didn't find expected continuation of a sysex");
    if ((c & 0x80) == 0)
    {
       if (status == 0)
            MIDI_error("unexpected running status");
       running = 1;
    }
    else
    {
       status = c;
       running = 0;
    }
    needed = chantype[(status>>4) & 0xf];
    if (needed)
    {
       if (running)
            c1 = c;
       else
            c1 = MIDI_getc();
       MIDI_channelmessage( status, c1, (needed>1) ? MIDI_getc() : 0 );
       continue;;
    }
    switch (c)
    {
       case 0xff:
            type = MIDI_getc();
            lookfor = MIDI_toberead - MIDI_readvarinum();
            MIDI_msginit();
            while (MIDI_toberead > lookfor)
              MIDI_msgadd(MIDI_getc());
            MIDI_metaevent(type);
            break;
       case 0xf0:
            lookfor = MIDI_toberead - MIDI_readvarinum();
            MIDI_msginit();
            MIDI_msgadd(0xf0);
            while (MIDI_toberead > lookfor)
              MIDI_msgadd(c=MIDI_getc());
            if (c==0xf7 || MIDI_nomerge==0)
              MIDI_sysex();
            else
              MIDI_sysexcontinue = 1;
            break;
       case 0xf7:
            lookfor = MIDI_toberead - MIDI_readvarinum();
```

```
                if (!MIDI_sysexcontinue)
                   MIDI_msginit();
                while (MIDI_toberead > lookfor)
                   MIDI_msgadd(c=MIDI_getc());
                if (!MIDI_sysexcontinue)
                {
                   MIDI_dotime();
#ifdef DEBUG_MIDI
                    printf("Arbitrary bytes, leng=%d\n",MIDI_msgleng());
#endif
                }
                else
                   if (c == 0xf7)
                   {
                        MIDI_sysex();
                        MIDI_sysexcontinue = 0;
                   }
                break;
           default:
                MIDI_badbyte(c);
                break;
        }
   }
#ifdef DEBUG_MIDI
    printf("Track end\n");
#endif
   return(1);
}

void MIDI_badbyte(int c)
{
   char buff[32];

   sprintf(buff,"unexpected byte: 0x%02x",c);
   MIDI_error(buff);
}

void MIDI_textevent(int type, int leng, char *mess)
{
   static char *ttype[] =
   {
        NULL,
        "Text Event",              /* type=0x01 */
        "Copyright Notice",        /* type=0x02 */
        "Sequence/Track Name",
        "Instrument Name",         /* ...     */
        "Lyric",
        "Marker",
        "Cue Point",               /* type=0x07 */
        "Unrecognized"
   };
   int unrecognized = (sizeof(ttype)/sizeof(char *)) - 1;
   register int n, c;
   register char *p = mess;

   if (type < 1 || type > unrecognized)
        type = unrecognized;
   MIDI_dotime();
#ifdef DEBUG_MIDI
   printf("Meta Text, type=0x%02x (%s)  leng=%d\n",type,ttype[type],leng);
   printf("     Text = <");
   for ( n=0; n<leng; n++ )
   {
        c = *p++;
        printf( (isprint(c)||isspace(c)) ? "%c" : "\\0x%02x" , c);
   }
    printf(">\n");
#endif
}

void MIDI_metaevent(int type)
{
   int leng, denom;
   char *m = MIDI_msg();

   leng=MIDI_msgleng();
   switch (type)
   {
        case 0x00:
           MIDI_dotime();
#ifdef DEBUG_MIDI
           printf("Meta event, sequence number = %d\n",MIDI_to16bit(m[0],m[1]));
#endif
           break;
        case 0x01:     /* Text event */
        case 0x02:     /* Copyright notice */
        case 0x03:     /* Sequence/Track name */
        case 0x04:     /* Instrument name */
        case 0x05:     /* Lyric */
        case 0x06:     /* Marker */
        case 0x07:     /* Cue point */
        case 0x08:
        case 0x09:
        case 0x0a:
        case 0x0b:
        case 0x0c:
        case 0x0d:
        case 0x0e:
```

```
        case 0x0f:      /* These are all text events */
           MIDI_dotime();
           MIDI_textevent(type, leng, m);
          break;
        case 0x2f:      /* End of Track */
           MIDI_dotime();
#ifdef DEBUG_MIDI
           printf("Meta event, end of track\n");
#endif
          break;
        case 0x51:      /* Set MIDI_tempo */
           MIDI_dotime();
           MIDI_tempo=MIDI_to32bit(0,m[0],m[1],m[2]);
#ifdef DEBUG_MIDI
            printf("Tempo, microseconds-per-MIDI-quarter-note=%d\n",
MIDI_tempo);
#endif
          break;
        case 0x54:
           MIDI_dotime();
#ifdef DEBUG_MIDI
           printf("SMPTE, hour=%d minute=%d second=%d frame=%d fract-
frame=%d\n", m[0],m[1],m[2],m[3],m[4]);
#endif
          break;
        case 0x58:
           denom=1;
           while (m[1]-->0)
                denom*=2;
           MIDI_dotime();
#ifdef DEBUG_MIDI
           printf("Time signature=%d/%d  MIDI-clocks/click=%d  32nd-notes/24-
MIDI-clocks=%d\n",m[0],denom,m[2],m[3]);
#endif
          break;
        case 0x59:
           MIDI_dotime();
#ifdef DEBUG_MIDI
           printf("Key signature, sharp/flats=%d  minor=%d\n",m[0],m[1]);
#endif
          break;
        case 0x7f:
           MIDI_dotime();
#ifdef DEBUG_MIDI
           printf("Meta event, sequencer-specific, type=0x%02x eng=%d\n",type,leng);
#endif
          break;
        default:
           MIDI_dotime();
```

```
#ifdef DEBUG_MIDI
            printf("Meta event, unrecognized, type=0x%02x leng=%d\n",type,leng);
#endif
           break;
   }
}

void MIDI_sysex(void)
{
   MIDI_dotime();
#ifdef DEBUG_MIDI
    printf("Sysex, leng=%d\n",MIDI_msgleng());
#endif
}

void MIDI_channelmessage(int status, int c1, int c2)
{
   int chan;

   chan=status & 0xf;

   switch (status & 0xf0)
   {
        case 0x80:
           MIDI_dotime();
#ifdef DEBUG_MIDI
            printf("Note off, chan=%d pitch=%d vel=%d\n",chan+1,c1,c2);
#endif
           MIDI_NoteOff(chan+1, c1);
           break;
        case 0x90:
           MIDI_dotime();
#ifdef DEBUG_MIDI
            printf("Note on, chan=%d pitch=%d vel=%d\n",chan+1,c1,c2);
#endif
           MIDI_NoteOn(chan+1, c1, c2);
           break;
        case 0xa0:
           MIDI_dotime();
#ifdef DEBUG_MIDI
            printf("Pressure, chan=%d pitch=%d press=%d\n",chan+1,c1,c2);
#endif
           MIDI_PolyPressure(chan+1, c1, c2);
           break;
        case 0xb0:
           MIDI_dotime();
#ifdef DEBUG_MIDI
            printf("Parameter, chan=%d control=%d value=%d\n",chan+1,c1,c2);
#endif
```

```
            MIDI_Control(chan+1, c1, c2);
            break;
         case 0xe0:
            MIDI_dotime();
#ifdef DEBUG_MIDI
            printf("Pitchbend, chan=%d msb=%d lsb=%d\n",chan+1,c1,c2);
#endif
            MIDI_PitchBend(chan+1, c1);
            break;
         case 0xc0:
            MIDI_dotime();
#ifdef DEBUG_MIDI
            printf("Program, chan=%d program=%d\n",chan+1,c1);
#endif
            MIDI_Program(chan+1, c1);
            break;
         case 0xd0:
            MIDI_dotime();
#ifdef DEBUG_MIDI
            printf("Channel pressure, chan=%d pressure=%d\n",chan+1,c1);
#endif
            MIDI_Aftertouch(chan+1, c1);
            break;
   }
}

long MIDI_readvarinum(void)
{
   long value;
   int c;

   c = MIDI_getc();
   value = c;
   if ( c & 0x80 )
   {
         value &= 0x7f;
         do
         {
            c = MIDI_getc();
            value = (value << 7) + (c & 0x7f);
         } while (c & 0x80);
   }
   return (value);
}

long MIDI_to32bit(int c1, int c2, int c3, int c4)
{
   long value = 0L;
```

```
    value = (c1 & 0xff);
    value = (value<<8) + (c2 & 0xff);
    value = (value<<8) + (c3 & 0xff);
    value = (value<<8) + (c4 & 0xff);
    return (value);
}

int MIDI_to16bit(int c1, int c2)
{
    return ((c1 & 0xff ) << 8) + (c2 & 0xff);
}

long MIDI_read32bit(void)
{
    int c1, c2, c3, c4;

    c1 = MIDI_getc();
    c2 = MIDI_getc();
    c3 = MIDI_getc();
    c4 = MIDI_getc();
    return MIDI_to32bit(c1,c2,c3,c4);
}

int MIDI_read16bit(void)
{
    int c1, c2;

    c1 = MIDI_getc();
    c2 = MIDI_getc();
    return MIDI_to16bit(c1,c2);
}

void MIDI_error(char *s)
{
     printf("Error: %s\n",s);
    exit(1);
}

void  MIDI_msginit(void)
{
    MIDI_msgindex = 0;
}

char *MIDI_msg(void)
{
    return(MIDI_msgbuff);
}

int  MIDI_msgleng(void)
```

```
{
   return(MIDI_msgindex);
}

void MIDI_msgadd(int c)
{
   if (MIDI_msgindex >= MIDI_msgsize)
        MIDI_biggermsg();
   MIDI_msgbuff[MIDI_msgindex++] = c;
}

void MIDI_biggermsg(void)
{
   char *newmess;
   char *oldmess = MIDI_msgbuff;
   int oldleng = MIDI_msgsize;

   MIDI_msgsize += MSGINCREMENT;
   newmess = (char *) malloc((unsigned)(sizeof(char)*MIDI_msgsize));

   if(newmess == NULL)
        MIDI_error("malloc error!");

   if ( oldmess != NULL )
   {
      strcpy(newmess, oldmess);
        free(oldmess);
   }
   MIDI_msgbuff = newmess;
}
```

CTRON.H (on remote side)

```
#ifndef SGI_SOUND
#define SGI_SOUND

#define MSGINCREMENT 128

#define TRUE    1
#define FALSE       0

#define NUM_SOURCES 4

typedef float point3d[3];
typedef float point6d[7];

#define point6d_init(p) { p[0] = p[1] = p[2] = p[3] = p[4] = p[5] = 0.0; \
   p[6] = 1.0; }
#define point6d_copy(in, out) { out[0] = in[0]; out[1] = in[1]; \
   out[2] = in[2]; out[3] = in[3]; out[4] = in[4]; out[5] = in[5]; \
   out[6] = in[6]; }

enum CtronCommands { INIT, UPDATE, CLOSE, MODEL_HEAD, LOCATE_HEAD,
                  LOCATE_SOURCE, AMPLIFY_SOURCE, SHOW_MAP,
LOAD_HRTF
};

enum CtronUnits {
   INCHES, FEET,
    MMETER, CMETER, METER,
    LUNITS
};

extern char MIDI_playing;

typedef struct
{
    point6d location;
    char    uniform;
   char    changed;
} srcinfo;

typedef struct
{
   point6d head;
    char numsrc;
   srcinfo src[NUM_SOURCES];
} cinformation;

void float2str(float f, char *s);
void Ctron_open(char *device, char nsources, char units);
void Ctron_close(char stopsound);
void Ctron_amplify(char source, float dB);
void Ctron_locate_head(point6d pq);
void Ctron_locate_source(char source, point6d pq);
void Ctron_uniform(char source, point3d p);
void Ctron_model_default(void);
void Ctron_model_head(float aural_ocular, float inter_aural, float aural_crown);
int  Ctron_loadHRTF(char *filename);
void Ctron_update(void);
void Ctron_showmap(void);
void Cinfo_init( char sources );
void Ctron_output( point6d pq );
void Ctron_uni_output(point6d pq);

/* Main MIDI functions */
```

```
void  MIDI_NoteOff(char channel, char key);
void  MIDI_NoteOn(char channel, char key, char velocity);
void  MIDI_PolyPressure(char channel, char key, char pressure);
void  MIDI_Control(char channel, char function, char value);
void  MIDI_ChannelMode(char channel, char function);
void  MIDI_Program(char channel, char program);
void  MIDI_Aftertouch(char channel, char pressure);
void  MIDI_PitchBend(char channel, char change);
void  MIDI_PlayFile(char *fil);
void  MIDI_BeginPlay(char *fil);

/* MIDI file functions */

int   MIDI_readmt(char *s);
int   MIDI_getc(void);
long  MIDI_readvarinum(void);
long  MIDI_read32bit(void);
long  MIDI_to32bit(int c1, int c2, int c3, int c4);
int   MIDI_read16bit(void);
int   MIDI_to16bit(int c1, int c2);
char *MIDI_msg(void);
void  MIDI_textevent(int type, int leng, char *mess);
float MIDI_ticks2sec(long ticks, int division, int tempo);
void  MIDI_readheader(void);
int   MIDI_readtrack(void);
void  MIDI_badbyte(int c);
void  MIDI_metaevent(int type);
void  MIDI_sysex(void);
void  MIDI_channelmessage(int status, int c1, int c2);
void  MIDI_msginit(void);
int   MIDI_msgleng(void);
void  MIDI_msgadd(int c);
void  MIDI_biggermsg (void);
void  MIDI_error(char *s);
void  MIDI_dotime(void);

/* Control Defines */

#define MODULATION       0x01
#define BREATHCONTROL    0x02
#define FOOTCONTROL      0x04
#define PORTAMENTOTIME   0x05
#define DATAENTRY_MSB    0x06
#define MAINVOLUME       0x07
#define BALANCE          0x08
#define PAN              0x0A
#define EXPRESSION       0x0B
#define GPC1             0x10  /* General Purpose Controller 1 */
```

```
#define GPC2         0x11  /*  "    "    "    2 */
#define GPC3         0x12  /*  "    "    "    3 */
#define GPC4         0x13  /*  "    "    "    4 */
#define SUSTAIN        0x40
#define PORTAMENTO       0x41
#define SOSTENUTO       0x42
#define SOFT_PEDAL      0x43
#define HOLD_2        0x45
#define GPC5         0x50  /*  "    "    "    5 */
#define GPC6         0x51  /*  "    "    "    6 */
#define GPC7         0x52  /*  "    "    "    7 */
#define GPC8         0x53  /*  "    "    "    8 */
#define EXTEFFECT_DEPTH 0x5B  /* External Effects Depth */
#define TREMELO_DEPTH   0x5C
#define CHORUS_DEPTH    0x5D
#define CELESTE_DEPTH   0x5E  /* Also known as Detune Depth */
#define PHASER_DEPTH    0x5F
#define DATA_INCREMENT   0x60
#define DATA_DECREMENT   0x61
#define NRPN_LSB       0x62  /* Non-Registered Parameter Num */
#define NRPN_MSB       0x63  /* "    "    "    " */
#define RPN_LSB       0x64  /* Registered Parameter Number */
#define RPN_MSB       0x65  /*    "    "    " */

/* Channel Mode messages - use MIDI_ChannelMode() for these */

#define RESET_ALL_CONTROLS 0x01
#define LOCAL_CONTROL_ON   0x02
#define LOCAL_CONTROL_OFF  0x03
#define ALL_NOTES_OFF     0x04
#define OMNI_MODE_OFF     0x05
#define OMNI_MODE_ON      0x06
#define MONO_MODE_ON      0x07
#define POLY_MODE_OFF     0x07
#define MONO_MODE_OFF     0x08
#define POLY_MODE_ON      0x08

#endif
```

SERIAL.C (on remote side)

```
/***************************************************************************
        bb_serial.c  -  bare bones routines to control the
                        serial port of a unix workstation


        Trent Tuggle - Oct 14, 1993
        Added other routines necessary for controlling MIDI device by Mike
        Tedder on Dec 15, 1993.
**************************************************************************/

#include        "../include/serial.h"

int             device[5], result, packet_size;
struct termio           serialPortControl, oldPortControl;
int             holding, holdchar;


/*---------------------------------------------------------------------
        open the serial_port for operation on the port specified by
        device_name (eg. "/dev/ttyd2")
-----------------------------------------------------------------------*/
int openSerialPort( char *deviceName, int stype )
{
   int i;

   holding=0; holdchar=0;
   if ( stype == TRACKING )
   {
      if ((i = open(deviceName,O_RDWR)) < 0) {
         printf("ERROR in openSerialPort:  open failed on %s\n",deviceName);
        return(0);
       }
       device[1] = i;
   }
   if ( stype == SOUND )
   {
       if ((i = open(deviceName, O_RDWR | O_NDELAY)) < 0) {
          printf("ERROR in openSerialPort: open failed on %s\n", deviceName);
         return(0);
       }
       device[3] = i;
   }
   if ( stype == CONV )
   {
       if ((i = open(deviceName, O_RDWR | O_NDELAY)) < 0) {
          printf("ERROR in openSerialPort: open failed on %s\n", deviceName);
         return(0);
       }
       device[4] = i;
   }
   if (stype != TRACKING && stype != SOUND && stype != CONV)
   {
       printf("Error: invalid stype specified for openSerialPort\n");
     return(0);
   }
```

```
    if (ioctl(i,TCGETA,&oldPortControl) < 0) {
        printf("ERROR in openSerialPort:  ");
        printf("getting old port control information failed\n");
       return(0);
   }
    copyTermIO(&oldPortControl,&serialPortControl);

    serialPortControl.c_iflag = IGNBRK;
   serialPortControl.c_oflag = 0;
   serialPortControl.c_cflag = B9600 | CS8 | CREAD | CLOCAL | CSTOPB;
   serialPortControl.c_lflag = 0;
   serialPortControl.c_line  = 0;  /* line discipline 1 */
   serialPortControl.c_cc[VMIN]  = 0;    /* min packet size */
   serialPortControl.c_cc[VTIME] = 0;

/* initial port setup so that we can talk */
    if (ioctl(i,TCSETA,&serialPortControl) < 0) {
        printf("ERROR in openSerialPort:  ");
        printf("setting serialPort control information failed\n");
       return(0);
   }

   return(1);
}



/*-------------------------------------------------------------------------
        open the serial_port for operation on the port specified by
        device_name (eg. "/dev/ttyd2")
-------------------------------------------------------------------------*/
int raiseBaudRate192( char *deviceName, int stype)
{
   int i;

   if (stype == TRACKING) i=device[1];
   if (stype == SOUND) i=device[3];
   if (stype == CONV) i=device[4];

    if (ioctl(i,TCGETA,&oldPortControl) < 0) {
        printf("ERROR in raiseBaudRate:  ");
        printf("getting old port control information failed\n");
       return(0);
   }
    copyTermIO(&oldPortControl,&serialPortControl);

   serialPortControl.c_cflag = B19200 | CS8 | CREAD | CLOCAL | CSTOPB;
```

```
/* write out new termio structure */
    if (ioctl(i,TCSETA,&serialPortControl) < 0) {
        printf("ERROR in raiseBaudRate:  ");
         printf("setting new termio control information failed\n");
       return(0);
    }

    return(1);
}

/*-------------------------------------------------------------------
        open the serial_port for operation on the port specified by
        device_name (eg. "/dev/ttyd2")
-----------------------------------------------------------------------*/
int raiseBaudRate384( char *deviceName, int stype)
{
    int i;

    if (stype == TRACKING) i=device[1];
    if (stype == SOUND) i=device[3];
    if (stype == CONV) i=device[4];

    if (ioctl(i,TCGETA,&oldPortControl) < 0) {
        printf("ERROR in raiseBaudRate:  ");
         printf("getting old port control information failed\n");
       return(0);
    }
     copyTermIO(&oldPortControl,&serialPortControl);

   serialPortControl.c_cflag = B38400 | CS8 | CREAD | CLOCAL | CSTOPB;

/* write out new termio structure */
    if (ioctl(i,TCSETA,&serialPortControl) < 0) {
        printf("ERROR in raiseBaudRate:  ");
         printf("setting new termio control information failed\n");
       return(0);
    }

    return(1);
}


/*----------------------------------------------------------------------
        read data from the serialPort

********  NOTE:    not needed now, implement later...    ********

------------------------------------------------------------------------*/
int Serial_datawaiting(int stype)
```

```
{
    if (holding) return 1;
    holdchar=Serial_getchar(stype);
    if (holdchar != -1) holding=1;
    return holding;
}

int Serial_getchar(int stype)
{
   char s[2];
   int result,i;

   if (holding)
   {
        holding=0;
        return holdchar;
   }

   if (stype == TRACKING) i=device[1];
   if (stype == SOUND) i=device[3];
   if (stype == CONV) i=device[4];

   result = read(i, s, 1);
   if (result != 1) return -1;
   return s[0];
}

void Serial_flush(int stype)
{
   while (Serial_datawaiting(stype))
        Serial_getchar(stype);
}

int readSerialPort( char *buffer, int length, int stype )
{
   int  result, i;

   if (stype == TRACKING) i=device[1];
   if (stype == SOUND) i=device[3];
   if (stype == CONV) i=device[4];

   result = read(i, buffer, length);
   if (result != length) {
        printf("read from serial port failed\n");
        return(0);
   }
   return(result);
}
```

```
int readCompletePacket( char *buffer, int length, int stype )
{
    int offset=0, count=0, i;

    if (stype == TRACKING) i=device[1];
    if (stype == SOUND) i=device[3];
    if (stype == CONV) i=device[4];

    while ( offset < length )
    {
        offset += read( i, buffer+offset, length );
        count++;
        if (count > 5000)
        {
            printf("Waited too long for packet ...\n");
            return 0;
        }
    }
    return 1;
}

/*---------------------------------------------------------------------
        write a buffer to the serialPort
---------------------------------------------------------------------*/
int Serial_putchar(char ch, int stype)
{
    char s[2];
    int result, i;

    if (stype == TRACKING) i=device[1];
    if (stype == SOUND) i=device[3];
    if (stype == CONV) i=device[4];

    s[0]=ch; s[1]='\0';
    result = write(i, s, 1);
    if (result != 1) return -1;
    return 0;
}

int writeSerialPort( char *buffer, int length, int stype)
{
    int  result, i;

    if (stype == TRACKING) i=device[1];
    if (stype == SOUND) i=device[3];
    if (stype == CONV) i=device[4];

    result = write(i, buffer, length);
    if (result != length) {
      printf("    write of <%s> failed, result = %d\n", (char *)buffer, result);
      perror("[writeSerialPort]");
        return(0);
   }
   return(result);
}



/*---------------------------------------------------------------------------
      close the serialPort (restore the original port control information)
---------------------------------------------------------------------------*/
int closeSerialPort(int stype)
{
   int i;

   if (stype == TRACKING) i=device[1];
   if (stype == SOUND) i=device[3];
   if (stype == CONV) i=device[4];

   if (ioctl(i,TCSETA,&oldPortControl) < 0) {
      printf("ERROR in closeSerialPort:  ");
      printf("resetting old port control information failed\n");
     return(0);
   }
   return(1);
}



/*------------*/
void copyTermIO( struct termio *source, struct termio *dest ) {
      dest->c_iflag = source->c_iflag;
      dest->c_oflag = source->c_oflag;
      dest->c_cflag = source->c_cflag;
      dest->c_lflag = source->c_lflag;
      dest->c_line  = source->c_line;
/*    strncpy(dest->c_cc,source->c_cc,NCC+NCC_PAD+NCC_EXT);*/
      bcopy(source->c_cc, dest->c_cc, 8);
}
```

SERIAL.H (on remote side)

```
/***************************************************************************
      bb_serial.h  -  bare bones routines to control the
                      serial port of a unix workstation
```

```
        Trent Tuggle - Oct 14, 1993

        Added other routines necessary for controlling MIDI device by Mike
        Tedder on Dec 15, 1993.
*****************************************************************************

#include    <stdio.h>
#include    <string.h>

#ifndef KUBOTA
#include    <bstring.h>
#endif

#include    <sys/ioctl.h>
#include    <sys/types.h>
#include    <sys/file.h>
#include    <termio.h>
#include    <unistd.h>
#include    <sys/fcntl.h>
#include    <fcntl.h>
#include    <sys/stat.h>


int openSerialPort(char *, int);
int raiseBaudRate192(char *, int);
int raiseBaudRate384(char *, int);
int Serial_datawaiting(int);
int Serial_getchar(int);
int readSerialPort(char *, int, int);
int Serial_putchar(char, int);
int writeSerialPort(char *, int, int);
int closeSerialPort(int);
int readCompletePacket(char *, int, int);
void copyTermIO(struct termio *, struct termio *);
void  Serial_flush(int);

/* Types for openSerialPort */

#define TRACKING 0   /* To open the Fastrak  */
#define SOUND    1   /* To open the MIDIator */
#define CONV     2   /* To open the Convolvotron */
```