```
/******************************************/
/*
  playsaw.cpp
  by Gary P. Scavone, 2006

  This program will output sawtooth waveforms
  of different frequencies on each channel.
*/
/******************************************/

#include "RtAudio.h"
#include <iostream>
#include <cstdlib>

/*
typedef char MY_TYPE;
#define FORMAT RTAUDIO_SINT8
#define SCALE  127.0
*/

typedef signed short MY_TYPE;
#define FORMAT RTAUDIO_SINT16
#define SCALE  32767.0

/*
typedef S24 MY_TYPE;
#define FORMAT RTAUDIO_SINT24
#define SCALE  8388607.0

typedef signed long MY_TYPE;
#define FORMAT RTAUDIO_SINT32
#define SCALE  2147483647.0

typedef float MY_TYPE;
#define FORMAT RTAUDIO_FLOAT32
#define SCALE  1.0

typedef double MY_TYPE;
#define FORMAT RTAUDIO_FLOAT64
#define SCALE  1.0
*/

// Platform-dependent sleep routines.
#if defined( WIN32 )
  #include <windows.h>
  #define SLEEP( milliseconds ) Sleep( (DWORD) milliseconds )
#else // Unix variants
  #include <unistd.h>
```

```
  #define SLEEP( milliseconds ) usleep( (unsigned long) (milliseconds * 1000.0) )
#endif

#define BASE_RATE 0.005
#define TIME   1.0

void usage( void ) {
  // Error function in case of incorrect command-line
  // argument specifications
  std::cout << "\nuseage: playsaw N fs <device> <channelOffset> <time>\n";
  std::cout << "    where N = number of channels,\n";
  std::cout << "    fs = the sample rate,\n";
  std::cout << "    device = optional device to use (default = 0),\n";
  std::cout << "    channelOffset = an optional channel offset on the device (default = 0),
\n";
  std::cout << "    and time = an optional time duration in seconds (default = no limit).
\n\n";
  exit( 0 );
}

// REPLACE ABOVE with
/*****************************************/
/*
  playsaw.cpp
  by Gary P. Scavone, 2006

  Modified for hw732_playsine.cpp
  John Ellinger CS312
*/
/*****************************************/

#include "RtAudio.h"
#include <iostream>
#include <cmath> // for M_PI

// EITHER/OR
typedef float MY_TYPE;
#define FORMAT RTAUDIO_FLOAT32
// typedef double MY_TYPE;
// #define FORMAT RTAUDIO_FLOAT64
// EITHER/OR END

const int FS = 44100;                    // CD sample rate
const MY_TYPE T = 1.0 / FS; // sample period
```

```
const MY_TYPE k2PI = M_PI * 2.0;
const MY_TYPE k2PIT = k2PI * T;

void errorCallback( RtAudioError::Type type, const std::string &errorText )
{
  // This example error handling function does exactly the same thing
  // as the embedded RtAudio::error() function.
  std::cout << "in errorCallback" << std::endl;
  if ( type == RtAudioError::WARNING )
    std::cerr << '\n' << errorText << "\n\n";
  else if ( type != RtAudioError::WARNING )
    throw( RtAudioError( errorText, type ) );
}

unsigned int channels;
// REPLACE ABOVE with
unsigned int channels = 1;

RtAudio::StreamOptions options;
unsigned int frameCounter = 0;
bool checkCount = false;
unsigned int nFrames = 0;
const unsigned int callbackReturnValue = 1;

//#define USE_INTERLEAVED
#if defined( USE_INTERLEAVED )

// Interleaved buffers
int saw( void *outputBuffer, void *inputBuffer, unsigned int nBufferFrames,
         double streamTime, RtAudioStreamStatus status, void *data )
{
  unsigned int i, j;
  extern unsigned int channels;
  MY_TYPE *buffer = (MY_TYPE *) outputBuffer;
  double *lastValues = (double *) data;

  if ( status )
    std::cout << "Stream underflow detected!" << std::endl;

  for ( i=0; i<nBufferFrames; i++ ) {
    for ( j=0; j<channels; j++ ) {
      *buffer++ = (MY_TYPE) (lastValues[j] * SCALE * 0.5);
      lastValues[j] += BASE_RATE * (j+1+(j*0.1));
      if ( lastValues[j] >= 1.0 ) lastValues[j] -= 2.0;
    }
  }

  frameCounter += nBufferFrames;
```

```
  if ( checkCount && ( frameCounter >= nFrames ) ) return callbackReturnValue;
  return 0;
}

#else // Use non-interleaved buffers

int saw( void *outputBuffer, void * /*inputBuffer*/, unsigned int nBufferFrames,
         double /*streamTime*/, RtAudioStreamStatus status, void *data )
{
  unsigned int i, j;
  extern unsigned int channels;
  MY_TYPE *buffer = (MY_TYPE *) outputBuffer;
  double *lastValues = (double *) data;

  if ( status )
    std::cout << "Stream underflow detected!" << std::endl;

  double increment;
  for ( j=0; j<channels; j++ ) {
    increment = BASE_RATE * (j+1+(j*0.1));
    for ( i=0; i<nBufferFrames; i++ ) {
      *buffer++ = (MY_TYPE) (lastValues[j] * SCALE * 0.5);
      lastValues[j] += increment;
      if ( lastValues[j] >= 1.0 ) lastValues[j] -= 2.0;
    }
  }

  frameCounter += nBufferFrames;
  if ( checkCount && ( frameCounter >= nFrames ) ) return callbackReturnValue;
  return 0;
}
#endif

// REPLACE ABOVE with
// One-channel sine wave generator replaces saw callback function
int sine(void *outputBuffer, void *inputBuffer, unsigned int nBufferFrames,
         double streamTime, RtAudioStreamStatus status, void *userData)
{
  MY_TYPE *buffer = (MY_TYPE *)outputBuffer;
  if (status)
    std::cout << "Stream underflow detected!" << std::endl;
  static MY_TYPE phz = 0;
  MY_TYPE freq = 440.0;
  MY_TYPE amp = 1.0;
  // //phase increment formula
  const MY_TYPE phzinc = k2PIT * freq;
  for (uint32_t i = 0; i < nBufferFrames; i++)
  {
```

```
    *buffer++ = amp * sin(phz);
    phz += phzinc;
    if (phz >= k2PI)
      phz -= k2PI;
  }
  frameCounter += nBufferFrames;
  if (checkCount && (frameCounter >= nFrames))
    return callbackReturnValue;
  return 0;
}

int main( int argc, char *argv[] )
{
  unsigned int bufferFrames, fs, device = 0, offset = 0;
  // REPLACE ABOVE with
  unsigned int bufferFrames;
  unsigned int  fs = FS;
  unsigned int   device = 0;
  unsigned int    offset = 0;

  // minimal command-line checking
  if (argc < 3 || argc > 6 ) usage();

  RtAudio dac;
  if ( dac.getDeviceCount() < 1 ) {
    std::cout << "\nNo audio devices found!\n";
    exit( 1 );
  }

  channels = (unsigned int) atoi( argv[1] );
  fs = (unsigned int) atoi( argv[2] );
  if ( argc > 3 )
    device = (unsigned int) atoi( argv[3] );
  if ( argc > 4 )
    offset = (unsigned int) atoi( argv[4] );
  if ( argc > 5 )
    nFrames = (unsigned int) (fs * atof( argv[5] ));
  if ( nFrames > 0 ) checkCount = true;

  double *data = (double *) calloc( channels, sizeof( double ) );
  // REPLACE ABOVE with
  MY_TYPE *data = (MY_TYPE *)calloc(channels, sizeof(MY_TYPE));

  // Let RtAudio print messages to stderr.
  dac.showWarnings( true );

  // Set our stream parameters for output only.
  bufferFrames = 512;
```

```
  RtAudio::StreamParameters oParams;
  oParams.deviceId = device;
  oParams.nChannels = channels;
  oParams.firstChannel = offset;

  if ( device == 0 )
    oParams.deviceId = dac.getDefaultOutputDevice();

  options.flags = RTAUDIO_HOG_DEVICE;
  options.flags |= RTAUDIO_SCHEDULE_REALTIME;
#if !defined( USE_INTERLEAVED )
  options.flags |= RTAUDIO_NONINTERLEAVED;
#endif
  try {
    dac.openStream( &oParams, NULL, FORMAT, fs, &bufferFrames, &saw, (void *)data,
&options, &errorCallback );
  // REPLACE ABOVE with
  dac.openStream(&oParams, nullptr, FORMAT, fs, &bufferFrames, &sine, (void *)data,
&options, &errorCallback);
    dac.startStream();
  }
  catch ( RtAudioError& e ) {
    e.printMessage();
    goto cleanup;
  }

  if ( checkCount ) {
    while ( dac.isStreamRunning() == true ) SLEEP( 100 );
  }
  else {
    char input;
    //std::cout << "Stream latency = " << dac.getStreamLatency() << "\n" << std::endl;
    std::cout << "\nPlaying ... press <enter> to quit (buffer size = " << bufferFrames <<
").\n";
    std::cin.get( input );

    try {
      // Stop the stream
      dac.stopStream();
    }
    catch ( RtAudioError& e ) {
      e.printMessage();
    }
  } // delete this brace

 cleanup:
  if ( dac.isStreamOpen() ) dac.closeStream();
  free( data );
```

```
  return 0;
}
```