CoreTone 1.1cz

Portable macro instrument software sampler for the *BupBoop Audio Suite*
Care and Feeding Instructions

Written and Designed by: Osman Celimli

◊Table of Contents

►What is CoreTone? 3
○Licensing 3
○Setting Up Your Development Environment 3
○Using CoreTone in Your Software 4
○WinTone Wrapper Library 5
○BupBoop Multiplatform Wrappers 6

►Audio Channel Structure 7
►Samples 8
►Macro Instruments and Sound Effects 9
○Sound Effects 9
○Instrument Packages 9
○Patch Scripts 10
►Music 11
○Music Tracks 11
○Music Scripts 11

►CoreTone Code Compiler (CTCC) 13
►Core-SASS Music-Oriented Language 15
○Constants 15
○Comments 15
○Timing 15
○Sample Packages 15
○Macro Instruments and Sound Effects 15
○Music Tracks 18
►PlayTone Auditioning Suite 23
○Getting Started 24
○Playing Music 25
○Playing Sound Effects 26

►What is CoreTone?

CoreTone is a portable wavetable synthesizer, music player, and sound effects driver; as well as the primary component of the *BupBoop Audio Suite*. It is based upon macro instruments, allowing for programmable envelopes instead of the traditional four-stage *ADSR* setup. Internally, the driver treats both instruments and sound effects as a single entity (referred to as *macros*), the only difference being that instruments are played with a frequency offset (i.e. their "note").

Other features include stereo panning, variable channel count, and priority-based sharing of resources by music and sound effects. CoreTone's rendering frequency and driver tick rate are both adjustable at compile time but default to 48 KHz and 240 Hz, respectively.

The primary target of *BupBoop* and its supporting libraries is 32-Bit Microsoft Windows with development in Visual C++ 6.0. However, the CoreTone synthesizer was written with the intention of portability among most popular operating systems and performant microcontrollers. CTCC, CoreTone's SASS compiler, is also platform agnostic. The only major assumption made for both components is a little-endian target and a full C-Standard library in the case of CTCC.

Windows-specific portions of *BupBoop* include WinTone and PlayTone. The former being a set of DirectSound wrappers and tools to simplify the use of CoreTone in Windows games, while the latter is a general-use auditioning program. Multiplatform software can be simplified through the use of the BupBoop Multiplatform Wrappers.

Please note that the authors are not responsible for any personal distress, harm, or destruction of property while using the *BupBoop Audio Suite*; however unlikely they may be.

○Licensing

All versions of BupBoop / CoreTone are available under the zlib license, whose terms and conditions are listed below in full and made available in all major component files:

```
(C) 2015 Osman Celimli

  This software is provided 'as-is', without any express or implied
  warranty.  In no event will the authors be held liable for any damages
  arising from the use of this software.

  Permission is granted to anyone to use this software for any purpose,
  including commercial applications, and to alter it and redistribute it
  freely, subject to the following restrictions:

  1. The origin of this software must not be misrepresented; you must not
     claim that you wrote the original software. If you use this software
     in a product, an acknowledgment in the product documentation would be
     appreciated but is not required.
  2. Altered source versions must be plainly marked as such, and must not be
     misrepresented as being the original software.
  3. This notice may not be removed or altered from any source distribution.
```

○Setting Up Your Development Environment

The contents of this folder may be extracted wherever you like. For convenience I recommend a folder close to root, such as `C:\BUPBOOP\` or `C:\WINDEV\BUPBOOP\`. After completing this step your development directory should have the following residents:

File / Folder	**Description**
.\CoreTone\	CoreTone Driver
.\WinTone\	Windows + DirectSound Wrappers
.\PlayTone\	PlayTone Auditioning Suite
.\CTCC\	CoreTone Code Compiler (Core-SASS)
.\SASS\	Example Core-SASS Files
.\types.h	
.\bupboop.h	
.\BupBoop.dsw	Visual C++ 6.0 Workspace
.\CoreTone Programmer's Manual.pdf	This Document

○Using CoreTone in Your Software

The most important aspect of a sound driver is *usually* generating sound. However, in order to get CoreTone up and running with your current project you'll need to observe the following:

- **Notable Definitions / Equates**
  - **`CORETONE_CHANNELS`**
    How many channels the CoreTone SoftSynth will have available for rendering music and sound effects. This defaults to **`16`**.
  - **`CORETONE_DEFAULT_VOLUME`**
    How loud music will be played when CoreTone is first initialized.
  - **`CORETONE_MUSIC_STACKDEPTH`**
  - **`CORETONE_PATCH_STACKDEPTH`**
    The music and macro script decoder stack length. This indicates how many **`CALL`**s and **`LOOP`**s you can use in your scripts. A value of **`4`** is recommended.
  - **`CORETONE_DISPATCH_DEPTH`**
  - **`CORETONE_REQUEST_DEPTH`**
    Specifies the depth of the dispatch and request queues for sound effects. Increasing these allows more sound effects to be queued up for playback or manipulation by the user in between driver ticks. A value of **`32`** is the default, but this can be lowered to **`8`** for most software.
  - **`CORETONE_RENDER_RATE`** and **`CORETONE_DECODE_RATE`**
    Samplerate of CoreTone's final rendered audio and the rate at which the driver itself performs update ticks. These default to 48 KHz and 240 Hz, respectively.
  - **`CORETONE_SAMPLES_MAXENTRIES`**
  - **`CORETONE_SAMPLES_MAXLENGTH`**
    The maximum number of samples which CoreTone can load from a Sample Package, and the largest size for each individual sample. These default to **`256`** and **`32768`**, respectively.

- **Render Buffer Format**
  CoreTone expects the target of its render buffer to be 16-Bit Signed Stereo PCM with a length of **`CORETONE_BUFFER_LEN`** mono samples or **`CORETONE_BUFFER_SAMPLES`** stereo samples.

- **Consumables / Sound Data**
  CoreTone accepts four types of consumables: Sample Packages, Sound Effects, Instrument Packages, and Music Tracks. At minimum, a Sample Package and Instrument Package are required at initialization time to setup the driver and begin rendering.

- **Initializing and Running**
  After configuring your consumables as described above, you can initialize the driver by calling **`ct_init(uint8_t *pSamplePak, uint8_t *pInstrPak)`** which is supplied with the base address of a Sample Package **`pSamplePak`** and Instrument Package **`pInstrPak`**. If any errors occur during setup, a nonzero value will be returned. Otherwise, CoreTone will then be ready for rendering and the driver update routine **`ct_update(int16_t *pBuffer)`** may be called at the specified driver tick rate, **`CORETONE_DECODE_RATE`**.

- **Commonly Used Subroutines**
  Once CoreTone has been initialized and is executing once every update tick, the following functions will be useful to the programmer:
  - **`void ct_pause(void)`**
  - **`void ct_resume(void)`**
    Pause and silence or unpause and resume all driver activity.

- **`void ct_stopAll(void)`**
  Cease playback of any currently decoding music, sound effects, or samples.
- **`void ct_setRenderCall(ct_renderCall_t pCall)`**
  Assign the function `pCall` as a post-render callback.

- **`void ct_playSFX(uint8_t *pSFX, int8_t cPriority, int8_t cVol_Left, int8_t cVol_Right)`**
  Request playback of the sound effect whose base address is `pSFX`, with priority `cPriority`, and panning `cVol_Left` and `cVol_Right`.
- **`void ct_stopSFX(int8_t cPriority)`**
  Cease playback any sound effect whose priority is `cPriority`.

- **`void ct_playMusic(uint8_t *pMusic)`**
  Request playback of the music track base address is `pMusic`.
- **`void ct_stopMusic(void)`**
  Cease playback of any currently decoding music track.
- **`void ct_attenMusic(int8_t cVol)`**
  Request a change in the current music attenuation to the level supplied in `cVol`, this may range from 0 (max attenuation, silent) to 127 (no attenuation, loudest).
- **`int32_t ct_checkMusic(void)`**
  Check to see whether the music is playing (nonzero return) or not (zero), useful to determine if a single-shot track has finished or if a stop request has completed.
- **`int32_t ct_getMood(void)`**
  Get the mood flag of the currently playing music track. If no music is playing, the mood flag will be zero (neutral). This can be used to alter game object behavior based upon music cues.

- **Post-Render Callbacks**
  CoreTone allows for the assignment of a single post-render callback. This is a function of the type **`ct_renderCall_t`**, with the following format:

  `int32_t rendrCall(void *pBuffer, uint32_t uiFreq, uint32_t uiLen)`

  Once a post-render callback is assigned using **`ct_setRenderCall(pCall)`**, the function **pCall** will be executed at the end of each render frame and supplied with the raw stereo buffer CoreTone has rendered into, **`pBuffer`**. This may be manipulated by the post-render callback in order to perform additional mixing or post-processing. The samplerate and length of the buffer in stereo samples are provided in **`uiFreq`** and **`uiLen`**, respectively.
  The post-render callback will continue to be executed at the end of each render frame until it returns zero.

○WinTone Wrapper Library

Windows support for CoreTone is accomplished through the use of the wrapper library WinTone, which performs audio updates and output using DirectSound and MultiMedia Timers. All user-facing CoreTone functions (prefaced by **`ct_x`** and in **`coretone.h`**) have equivalent WinTone functions which are identical in name and argument, but prefaced by **`wt_x`**. These should be used instead of calls directly to CoreTone in Windows. For multiplatform software, please use the BupBoop Multiplatform Wrappers.

Some functions have slightly different arguments or offer features not natively available by CoreTone, these are listed below:

- **`HRESULT wt_init(HWND hParent, uint8_t *pSamplePak, uint8_t *pInstrPak)`**

Initialize both the CoreTone library and DirectSound hookups using the supplied parent window, sample package, and instrument package.
Will return a generic **`E_FAIL`** HRESULT if there was an issue initializing CoreTone or the specific Windows + DirectSound error if the problem originated from an interaction with either of those.

- **`void wt_close(void)`**
  Close down WinTone and all supporting components, should be called before exiting your Windows application.

- **`int32_t wt_logOutput(char *pszFile)`**
  Attempt to begin logging the softsynth output to the given file in the RIFF/WAVE format. Will return a nonzero value if the log could not start.
- **`void wt_closeLog(void)`**
  Stop the currently active output log, stamp the RIFF/WAVE header on the target file, and close it.

○BupBoop Multiplatform Wrappers

Multiplatform support for CoreTone and the automatic selection of a wrapper library for a given target is accomplished through **`bupboop.h`**. All audio-pertinent CoreTone functions (prefaced by **`ct_x`** and in **`coretone.h`**) have equivalent BupBoop functions which are identical in name and argument, but prefaced by **`bb_x`**. These should be used instead of calls directly to CoreTone or any platform-specific wrapper library such as WinTone in multiplatform game software.

All initialization, including library setup and teardown is left aside as a platform specific implementation due to the vast differences among the operating systems and embedded devices which may be used with BupBoop and CoreTone.

►Audio Channel Structure

All channels of CoreTone's software synthesizer are identical in features and based upon the playback of pre-recorded waveforms with variations in both frequency and amplitude. These are shared by any currently playing music and sound effects, making them a finite (and valuable) system resource.

For all intents and purposes each channel has the properties listed below:

- **Priority**
  The current priority of the instrument or sound effect playing in the channel. When sound effects and instruments compete for a given audio channel, those with higher priorities win out. A priority of zero indicates the channel is currently free (no one is using it).
- **Sample**
  The channel's currently assigned waveform.
- **Sample Mode**
  How the waveform specified above will be played. This can be either **`SINGLESHOT`** (play straight to the end, then stop) or **`LOOPING`** (repeat a selected region).
- **Phase**
  A 16.16-Bit precision phase accumulator indicating the current waveform position.
- **Phase Adjustment**
  A 16.16-Bit precision phase accumulation value indicating the rate at which the waveform will be traversed during rendering. The final phase adjustment value is actually the sum of several sub-frequencies:
  - **Base**
    16.16-Bit precision value representing the current base frequency of the channel. This is ignored (zeroed) when playing a sound effect, but is used for the note frequency in instrument decoding.
  - **Pitch and Pitch Adjustment**
    16.16-Bit precision value representing the current offset from the base frequency and how much to adjust this offset each driver tick. This is ignored (zeroed) when playing a sound effect, but is used for detuning and pitch bends in instruments.
  - **Offset and Offset Adjustment**
    16.16-Bit precision value representing the offset from the sum of the base frequency and pitch. This is the sole frequency used by sound effects, but is used by notes for variation from their base frequency.

    Noting the above, we can calculate the phase adjustment as follows:
    - **For Instruments**
      ```
      pitch += pitchAdjustment;
      offset += offsetAdjustment;
      phaseAdjustment = base + pitch + offset;
      ```
    - **For Sound Effects**
      ```
      offset += offsetAdjustment;
      phaseAdjustment = offset;
      ```

- **Volume and Volume Adjustment**
  An 8.8 precision signed value, of which the upper 8-Bits are used as the final channel volume during rendering. Each driver tick the volume adjustment value is added to the current volume value, allowing for simple amplitude modulation. A value of 0 indicates silence while 127 is maximum positive volume and -128 is maximum negative (results in the inversion of the waveform) volume.
- **Panning**
  Two 8-Bit values representing the panning of the channel in the left and right speakers. These are combined with the volume attribute above to calculate the final stereo amplitudes, and the same range restrictions apply.

►Samples

The pre-recorded waveforms used by CoreTone to generate its audio are stored in data structures called Sample Packages. These contain not only the 8-Bit Signed Monophonic samples which will be used by Instruments and Sound Effects, but information pertaining to the waveforms' frequency content. They are arranged in the following format:

```
Offset    Contents
0         'C'
1         'S'
2         'M'
3         'P'
4-7       Number of Samples
8+        Directory Entries (16-Bytes Each):
 \
  0-3           Data Start Offset
  4-7           Data Length
  8-11          Sample Frequency (Sf)
  12-15         Content Frequency (Bf)

...       Sample Data
```

Once a sample package has been verified by CoreTone, its base address, directory start address, and sample count will be cached for later use. However, CoreTone will also be generating a 32.32 table to speed up calculations later on. This is the *frequency ratio table* and to fully understand its purpose we must first describe *phase increment calculations*.

Calculating playback phase increment (Pi) of a particular sample based upon a desired frequency requires the knowledge of four components: Rendering Frequency (Rf), Playback Frequency (Pf), Sample Frequency (Sf), and the Sampled Frequency (Bf).

The naive formula is: Pi = (Sf / Rf) * (Pf / Bf), but the two divides aren't desirable for performance reasons. Noting that Sf, Rf, and Bf are fixed once the sample package is loaded, we can reformat the equation:

Pi = Pf * Fr, where Fr = (Sf / (Rf * Bf)) and will be precalculated and stored in 32.32 precision which should give tolerable accuracy when multiplied by a 16.16 frequency to yield the final 16.16 phase increment.

The size of the frequency ratio table is set by **`CORETONE_SAMPLES_MAXENTRIES`**, which defaults to **`256`**. Lowering this value will reduce CoreTone's memory requirements, but will also reduce the maximum number of samples which may be used by the software synthesizer.

►Macro Instruments and Sound Effects

In CoreTone, instruments and sound effects have been merged into a single entity called *macros* which describe their sound through a *patch script*. This allows the audio programmer to not only create elaborate instruments with all the complex envelopes of sound effects, but also reduces driver size and complexity.

While the instruments and sound effect patch scripts are decoded identically, they are not stored in the same data structures. Sound effects live inside of individual files, while instruments are stored in a single Instrument Package. The headers prefacing individual Sound Effects and Instrument Packages differ as well. A description of this formatting and encoding follows.

○Sound Effects

A single sound effect may have one or more associated patch scripts, allowing multiple channels of sound to be dispatched and controlled simultaneously. Well behaved sound effects will be formatted as shown below:

```
Offset    Contents
0         'C'
1         'S'
2         'F'
3         'X'
4-7       Number of Channels
8+        Channel Directory
 \
  0-3           Sample Number
  4-7           Patch Script Offset

...       Patch Scripts
```

○Instrument Packages

Instruments may only be dispatched by **`NOTE ON`** commands in a music track, and therefore have slightly different encoding than sound effects. In particular, all instruments are stored together in one binary.

One patch script is associated with each instrument, allowing control over a single channel while the instrument is active. However, two offsets within this script are used by the driver: the start and the note off offset. The latter of these is a user-specified point within the patch script which the macro decoder will *jump* to when a note off command is encountered in the music script. This allows customized behavior for the release phase of each instrument.

Unlike sound effects, the frequency an instrument is played at will be offset not only by the effects in its patch script, but the currently playing note in a music script and any pitch bends applied to it.

```
Offset    Contents
0         'C'
1         'I'
2         'N'
3         'S'
4-7       Number of Instruments
8+        Instrument Directory
 \
  0-3           Sample Number
  4-7           Patch Script Offset
  8-11          Note Off Offset

...       Patch Scripts
```

○Patch Scripts

Following the instrument or sound effect's header are the scripts for each of its channels. This script is a simple bytecode with commands for adjusting the properties of the audio channel and controlling the execution path of the sound script itself. The commands, their format, and operations are listed below:

- **`0: CORETONE_PATCH_END()`**
  Close the patch script decoding process and silence the channel.
- **`1: CORETONE_PATCH_MODE_SINGLESHOT()`**
  Place the channel's assigned sample in **`SINGLESHOT`** mode, playback will continue from its current position until the sample's end. This is useful for percussion samples or "baked-in" releases.
- **`2: CORETONE_PATCH_MODE_LOOP(usLoopStart, usLoopEnd)`**
  Place the channel's assigned sample in **`LOOPING`** mode, playback will advance from its current position until reaching the supplied loop region (**`usLoopStart`** and **`usLoopEnd`**). It will then wrap within that region until the mode is changed or the patch script ends.
- **`3: CORETONE_PATCH_VOLUME(cVol, cAdj_Lo, cAdj_Hi)`**
  Set the volume and volume adjustment for this channel.
- **`4: CORETONE_PATCH_FREQUENCY(sOffset_Lo, sOffset_Hi, sAdj_Lo, sAdj_Hi)`**
  Set the frequency offset and frequency offset adjustment for this channel. The decimal of the frequency offset is cleared.
- **`5: CORETONE_PATCH_LOOP_START(cCount)`**
  Set the start of a looping region and define how many counts the loop will last. Negative loop counts are considered infinite. The maximum number of loops within loops is set at compile time by **`CORETONE_PATCH_STACKDEPTH`**, which has a default of **`4`**.
- **`6: CORETONE_PATCH_LOOP_END()`**
  Decrement the current loop count, branching back to the loop start if the loop count is greater than zero or negative (infinite).
- **`7: CORETONE_PATCH_NOP()`**
  Does absolutely nothing.

- Wait commands are a special case which are somewhat borrowed from MIDI, the delay (in ticks) is variable length up to four bytes long. The MSB of each byte indicates whether or not to extend the delay count and the seven remaining bits are placed into bits **`7-0`**, **`14-8`**, **`21-15`**, or **`28-22`** of said accumulated count.
  All other patch commands are below **`0x7F`** and have their MSB clear, so there shouldn't be any concern of crossover.

►Music

Music, like sound effects and instruments, uses a simple scripting language. Macro Instruments are played for various durations with adjustable frequency offsets, pitch slides, and panning. Looping is also supported, and as with macros, the maximum loop depth is configurable by adjusting **CORETONE_MUSIC_STACKDEPTH**, which has a default of **4**. Infinite loops as applicable in macros are identical in music scripts. Script patterns/subroutines may also be called and returned from, with the same depth limit as loops.

All voices of a music script have an adjustable priority. This allows tracks of music to become more or less important than the sound effects competing with them for available CoreTone channels. By specifying a priority of zero, a music script is stopped. It should also be noted that music scripts dispatch from the first channel upward, while sound effects play from the last channel downward in order to avoid contention.

○Music Tracks

Well behaved music tracks begin with a header indicating their channel requirements as formatted below:

```
Offset    Contents
0         'C'
1         'M'
2         'U'
3         'S'
4-7       Number of Channels
8+        Channel Directory:
 \
  0             Initial Channel Priority
  1-4           Music Script Offset

 ...      Music Scripts
```

○Music Scripts

Following a music track's header are its music scripts, one for each primary channel script and additional subscripts for **CALL**s. The scripting commands are made up of one byte blocks and are a merger of macro scripts and General MIDI. The commands, their format, and descriptions of their operations are listed below:

- **0: CORETONE_MUSIC_SET_PRIORITY(cPriority)**
  Set the music track's current playback priority. Setting the priority to zero will terminate the decode process.
- **1: CORETONE_MUSIC_SET_PANNING(cPanLeft, cPanRight)**
  Set the music track's current stereo panning using the two signed bytes following the command. As with all other signed amplitudes in CoreTone, negative panning values will cause waveform inversion.
- **2: CORETONE_MUSIC_SET_INSTRUMENT(ucInstrument)**
  Select an instrument from the currently loaded instrument package which will be used to play all notes on the script's assigned channel. Instruments may be reselected at any time during a scripts life.
- **3: CORETONE_MUSIC_NOTE_ON(ucNote)**
  Start playing the supplied note on currently selected instrument. The note values immediately following the command byte are equivalent to those used in General MIDI.
- **4: CORETONE_MUSIC_NOTE_OFF()**
  Have the currently playing instrument (if any) move to the note off portion of its script.
- **5: CORETONE_MUSIC_PITCH(sPitch_Lo, sPitch_Hi, sAdj_Lo, sAdj_Hi)**
  Set the current pitch offset and adjustment rate.

- **6: CORETONE_MUSIC_LOOP_START(cCount)**
  Set the start of a looping region and define how many counts the loop will last. Negative loop counts are considered infinite.
- **7: CORETONE_MUSIC_LOOP_END()**
  Decrement the current loop count, branching back to the loop start if the loop count is greater than zero or negative (infinite).
- **8: CORETONE_MUSIC_CALL(iOffset)**
  Call ("`jsr`") the music script whose base address is calculated by the sum of the current decode address (immediately after the **CALL** command) + **iOffset**.
- **9: CORETONE_MUSIC_RETURN()**
  Return ("`rts`") from the current music script position to the command after the last **CALL** performed.
- **10: CORETONE_MUSIC_BREAK()**
  Return ("`rts`") all channels (not just the current one) to the lowest return address found in their pattern **CALL** stack, effectively bringing all channels back to their "main" script.
- **11: CORETONE_MUSIC_NOP()**
  Strives to do absolutely nothing aside from consume one byte of script space.
- **12: CORETONE_MUSIC_SET_MOOD(iMood)**
  Sets the current mood flag to the supplied value, this may be read by the game software using `ct_getMood()`, and used to adjust game behavior based upon music activity. The mood flag is defaulted to zero (neutral) when a music track is started or stopped.

- Wait commands are a special case which is somewhat borrowed from MIDI, the delay (in ticks) is variable length up to four bytes long. The MSB of each byte indicates whether or not to extend the delay count and the seven remaining bits are placed into bits **7-0**, **14-8**, **21-15**, or **28-22** of said accumulated count.
  All other music commands are below **0x7F** and have their MSB clear, so there shouldn't be any concern of crossover.

►CoreTone Code Compiler (CTCC)

The CoreTone Code Compiler (CTCC) generates Sound Effects, Instrument Packages, and Music Tracks from textual source files written in a music-oriented programming language called Sound ASSembler (SASS). Two syntaxes are used, one for macro instrument and sound effect definitions, the other for music tracks. CTCC can also generate Sample Packages from special descriptor files and 8-Bit Signed Monophonic Samples.

CTCC is a commandline application and can operate in four modes: Sample Package, Sound Effects, Instrument Pack, and Music. Running CTCC with no arguments yields its usage:

```
C:\WinDev\BupBoop>CTCC.exe
Usage :
  ctcc (-smp/-sfx/-ins/-mus) [outfile] (source data)
 Where the specific syntax depends upon the operating mode, all of which are
 listed below...

::Sample Pack Mode (-smp),
  ctcc -smp output.smp samples.txt
 All samples in the sample descriptor file must be stored in 8-Bit Signed PCM
 with a frequency between 1.0Hz and 65535.99Hz

::Sound Effect Mode (-sfx),
  ctcc -sfx samples.txt sfx_scripts.txt
 Generated sound effect files (*.sfx) will be stored in the same directory as
 sfx_scripts.txt, with one sound effect file per SASS sound effect definition.

::Instrument Pack Mode (-ins),
  ctcc -ins output.ins samples.txt instruments.txt
```

In Sample Pack mode CTCC requires two arguments, the first of which indicates where the generated Sample Package will be stored and the second specifying the location of a sample descriptor file containing information about all samples which will be included in the package.

Building the included example sample package can be accomplished as follows:

```
C:\WinDev\BupBoop\SASS\Samples>..\..\ctcc -smp ..\CoreSamples.smp ..\CoreSamples.txt
SMP Mode...
Parsing sample descriptor file : ..\CoreSamples.txt
Saving sample package to : ..\CoreSamples.smp

Success! 2294 Bytes written to : ..\CoreSamples.smp
Total Samples : 10, Included Files : 9
```

In Sound Effects mode CTCC requires two filenames: the first of which indicates the Sample Package to use and the second of which is the SASS Sound Effects Script file. Individual sound effect binaries will be generated in the same directory as the Sound Effect Script file, one for each sound effect. A `*.h` file containing all sound effect priorities will also be generated.

Building the included example sound effects file can be accomplished as follows:

```
C:\WinDev\BupBoop\SASS\DemoSFX>..\..\ctcc -sfx ..\CoreSamples.txt DemoSFX.txt
SFX Mode...
Parsing sample descriptor file : ..\CoreSamples.txt
Parsing macro script file : DemoSFX.txt
Saving sound effect binary to : Get.sfx
Saving sound effect binary to : Yoomp.sfx
Saving sound effect binary to : Bonk.sfx
Saving sound effect binary to : Slip.sfx
Saving sound effect binary to : Lift_Start.sfx
Saving sound effect binary to : Lift_Stop.sfx

Priority definitions saved to : DemoSFX.h
Total Sound Effects : 6
```

In Instrument Pack mode CTCC requires three arguments, the first of which is the destination file where the Instrument Package will be stored, the second indicates the Sample Package to use, and the third is the SASS Instrument Script File.

Building the included example macro instruments can be accomplished as follows:

```
C:\WinDev\BupBoop\SASS>..\ctcc -ins CoreMacros.ins CoreSamples.txt CoreMacros.txt
INS Mode...
Parsing sample descriptor file : CoreSamples.txt
Parsing macro script file : CoreMacros.txt
Saving instrument package to : CoreMacros.ins

Success! 648 Bytes written to : CoreMacros.ins
Total Macro Instruments : 10
```

In Music mode, CTCC requires three or more arguments, the first of which indicates where the generated Music Track will be stored, the second of which is the SASS Instrument Scripts used in the piece, and the third and beyond contain the SASS Music Scripts for each CoreTone channel. Any channel other than the first is optional, and if a piece of music requires more than the total number of available channels on a particular CoreTone target, they will be dropped at playback time.

Building the included example song can be accomplished as follows:

```
C:\WinDev\BupBoop\SASS\DemoMusic>..\..\ctcc -mus DemoMusic.mus ..\CoreMacros.txt
 Lead_A.txt Lead_B.txt Bass.txt Perc.txt
MUS Mode...
Parsing macro script file : ..\CoreMacros.txt
Parsing music script file : Lead_A.txt
Parsing music script file : Lead_B.txt
Parsing music script file : Bass.txt
Parsing music script file : Perc.txt
Linking and saving music track to : DemoMusic.mus

Success! 203 Bytes written to : DemoMusic.mus
Total Tracks 4, Total Blocks : 7
```

►Core-SASS Music-Oriented Language

The CoreTone Code Compiler (CTCC) uses an extension of the SASS language called Core-SASS, which caters to the unique features of the CoreTone SoftSynth. A reference to the general SASS language and its Core-SASS extensions is provided below:

○Constants

Values may be entered in decimal, hexadecimal, and binary using the following syntax:

```
32              ; Decimal (No prefix)
-32             ; Negative Decimal (Leading "-")
$20             ; Hexadecimal (Leading "$")
%100000         ; Binary (Leading "%")
```

As CoreTone uses fractional-integer values (i.e. 16.16 or 8.8 precision math), a decimal may be assigned to each value as follows:

```
32.16           ; Decimal
-32.16          ; Negative Decimal
$20.10          ; Hexadecimal
%100000.10000   ; Binary
```

The format of the decimal matches that of its leading integer. Values written without a decimal are assumed to have one with value zero.

○Comments

Follow a semicolon ";" and span a single line:

```
; This is a comment, it will span the whole line
as4 240 ; Comments may also follow commands and values
```

○Timing

All values quantifying time (note on, rests, waits, etc.) are given in CoreTone update ticks, which are 240Hz by default. Using this, we can write the note sequence below:

```
f.3 240         ; Play an F in the third octave for one second
rest 80         ; Note off and rest for 1/3 second
as5 160         ; Play an A sharp in the fifth octave for 2/3 second
```

○Sample Packages

Waveforms available for use by instruments and sound effects are stored in Sample Packages which are created using a descriptor file in the format below:

```
;name          -       file            Sf      Bf              [start] [end]
Pulse_06       :       Pulse_06.raw    32000   2000
...
```

Each entry has a **name**, which is its identity for use by instruments and sound effects when selecting a sample. The **source file** is where the sample's waveform data will be pulled from when generating a sample packages. **Sf** is the *sampling* frequency, i.e. the samplerate of the input file. **Bf** is the *sampled* frequency, or the frequency of *what* was recorded in the input file. Optional **start** (inclusive) and **end** (exclusive) offsets allow selection of sample ranges within a file. This can be used to define multiple samples from different regions of a single file.

All source data for sample packages must be stored as 8-Bit signed monophonic PCM.

○Macro Instruments and Sound Effects

As the decoding of instruments and sound effects are identical in CoreTone, their definitions are also similar and both are classified as macros. Instruments are expected to be used in music tracks under the control of the music playback routine. Sound effects are available for playback at any time by the user or game routine. Therefore **priorities** are exclusive to sound effects, while **note offs** and are only available to instruments.

Unlike many other SASS targets, CoreTone has no **tuning** parameter available to instruments. They are instead autotuned when creating the sample package based upon the value of **Bf** / **Sf**.

```
name priority                                   ; <- Label
        volume value                            ; <- Header / Init
        frequency value
        sample name
        mode singleshot/looping [start] [end]
        {
                rest ticks                      ; <- Body

                volume value
                frequency value
                loop count
                        ; (Loop Body)
                endloop

                noteoff
                end
        }
```

All definitions begin with a **label**, which contains a macro's **name** and **priority** (if a sound effect). **Names** are composed of one or more ASCII characters (such as "square," "explosion," or "powpow"), are used to identify instruments within music scripts, and generate equates for sound effects. **Priorities** are used only for sound effects and represent the importance of the particular sound effect relative to both the music tracks and other sound effects. For instrument definitions, the priority should be excluded.

Following the label is the definition's **header data**. This specifies the **initial state** of the instrument or sound effect through various **parameter adjustment** commands. Some common parameter adjustment commands are **sample**, **volume**, and **frequency**.

The remaining portion of the definition is the **body**, which specifies how the sound will change over time and is composed of one or more **note**, **flow control**, and **parameter adjustment** commands within **{** Curly Braces **}**. Most of these are available for both instruments and sound effects, but **noteoff** only applies to instruments and will have no effect on sound effects.

As single-channel sound effects can quickly become a limiting factor in game audio fidelity, CoreTone supports the definition of multi-channel sound effects. These resemble a normal sound effect definition, except with a **{** Curly Brace **}** region immediately after the label, containing one or more channel headers and bodies:

```
name priority                                   ; <- Label
{
        volume value [adjustment]               ; <- 1st Channel Header / Init
        frequency value  [adjustment]
        sample name
        mode looping 0 64
        {
                rest ticks                      ; <- 1st Channel Body
                loop count
                        ; (Loop Body)
                endloop
                end
        }

        volume value [adjustment]               ; <- 2nd Channel Header / Init
        frequency value [adjustment]
        sample name
        mode singleshot
        {
                rest ticks                      ; <- 2nd Channel Body
                end
        }
```

```
}
```

Commands may be divided into three categories: **note**, **parameter adjustment**, and **flow control**. **Note** commands control delays and note off actions. **Parameter adjustments** control how the instrument or sound effect will actually create its sound, these are also the only commands which may be used in the channel header area. **Flow control** commands include loops and end markers. A basic command listing follows:

- **noteoff,n** : note off
  Applicable only to instruments. Flags the portion instrument body following it as the note off (or "release") part of the script. If an instrument is playing and the music script encounters a rest command, this area of the instrument body will begin decoding on the next update cycle.

```
        ...             ; When a rest command is encountered in
        noteoff         ; the music script while an instrument
        rest 32         ; is playing, the instrument will start
        end             ; executing the portion of its script
        ...             ; directly following the noteoff command.
```

- **rest,r ticks** : rest
- **wait,w ticks** : wait
  Pause for the specified number of ticks before advancing to the next command. The instrument or sound effect will continue to play during this time.

```
        rest 14         ; Wait 14 ticks
        end             ; Stop
```

- **volume,v value [adjustment]**: set the current volume
  Set the channel volume to a value of -128 to 127, where larger absolute values are louder and zero is silence. The adjustment field is an optional 8.8 precision value which will be added to the current volume every driver tick. Negative values from -1 to -128 will invert the waveform.

```
        volume 26 -2    ; Set amplitude to 26, -2 each tick
        rest 12         ; Wait 12 ticks, amplitude will be near zero
        end             ; Stop
```

- **frequency,f value [adjustment]**: set the current frequency offset
  Set the channel frequency offset, where larger values yield a larger phase adjustment (higher frequency). The adjustment field is an optional 16.16 precision value which is added to the current frequency offset every update tick.

```
        frequency 0 1   ; Set frequency offset to 0, +1 each tick
        rest 16         ; Wait 16 ticks
        end             ; Stop
```

- **sample,s name**: select sample
  Select the sample from the supplied Sample Package which will be played over the current channel. Only one sample may be selected per channel and will be locked for the channel's lifetime.

```
        sample sine     ; Use the sine sample for this macro
        rest 5          ; Wait a bit for it to finish
        end             ; Stop
```

- **mode,m singleshot/looping [start] [end]**: specify sample mode
  Choose how the sample assigned to the channel will be played. Single shot mode will play a sample until its end point, upon which the channel will be silenced. Looping mode will repeatedly play a region of a waveform specified by the start (inclusive) and end (exclusive) values, in samples.

Sample modes may be changed throughout the course of a channel's life, and a waveform can be "walked" by steadily advancing loop points. Additionally, single shot mode will always begin from the currently playing offset within a sample. So switching from looped mode to single shot can be convenient for note off routines in samples which have a "baked in" decay.

```
sample square           ; Use the square sample...
mode looping 0 16       ; ...and loop from 0(i) - 16(e)
rest 5                  ; Wait a bit for it to finish
end                     ; Stop
```

```
sample snare            ; Use the snare drum...
mode singleshot         ; ...and play it straight through
rest 120                ; Wait a bit for it to finish
end                     ; Stop
```

- **loop,l count** : start of loop
  Specifies the start of a loop, and the number of times it will repeat. If negative values are used, the loop will continue indefinitely.

```
...
loop -1                 ; Repeat outer loop forever
        loop 10         ; Inner loop 10 times
                rest 6
        endloop
endloop
end                     ; Stop
```

- **endloop,el** : loop end
  Specifies the end of a given loop.

```
...
loop -1                 ; Repeat forever
        rest 18
endloop                 ; Marks end of above loop
end                     ; Stop
```

- **end,e** : sound end
  Stops decoding of the instrument or sound effect when reached, and frees the given channel.

```
...
end                     ; Stop
```

○Music Tracks

Music tracks are composed of one or more script blocks which contain commands representing how and when to play instruments. An example listing is shown below:

```
name priority                                   ; <- Main Block
{
        using instrument
        priority value
        pan left right
        mood value

        call name
        loop count
                ; (Loop Body)
        endloop

        rest ticks
        as3 ticks
        wait ticks
        end
}

name                                            ; <- Sub Block
{
```

```
        ...
        return  / break
}
```

All script blocks start with a **label**, which at minimum contain the block's **name**. Each music track must contain a single **main block** which is the first to be played and also contains a starting **priority** in its **label**. **Names** are composed of one or more ASCII characters and **priorities** are an integer value, with higher values indicating a more important music track. Using a priority of zero will cause the music track to never decode.

Following the **label**, and enclosed between two **{** Curly Braces **}** are one or more commands composing the **block body**. The basic command set may be divided into three categories: **notes**, **parameter adjustment**, and **flow control**. **Note** commands specify when to turn on and off a particular note (ala key on and key off). **Parameter adjustment** commands allow control over which macro instrument will be used and how it will be played. **Flow control** commands change how the SASS script will decode, allowing for loops, calls to different music blocks, and termination of playback.

A basic command listing follows:

- **Note Commands**
  A key on event at a given note may be specified with a three letter note command, followed by a duration in driver ticks. The format is as follows:
  (Note Letter)(Natural, Sharp, or Flat)(Octave Number)
  Note letters may be **c**, **b**, **d**, **e**, **f**, **g**, and **a**. Natural, Sharp, and Flat for the given note may be specified using "**.**" (period), "**s**", and "**b**" respectively. The octave number may range from **0-9**.

```
        ...
        c.4 60          ; C-Natural 4th octave for 60 ticks
        rest 10
        bs4 20          ; B-Sharp 4th octave for 20 ticks
        rest 10
        bb2 80          ; B-Flat 2nd octave for 80 ticks
        rest 10
        ...
```

- **pitch**,**pt** **offset [adjustment]** : set pitch offset and adjustment
  Set the current 16.16 pitch offset and its optional adjustment each driver tick.

```
        ...
        as3 5           ; Start playing a note, wait five ticks
        pitch 0 0.16    ; INCREASE the frequency 0.16 counts / tick
        wait 20         ; After 20 ticks, it will be offset by
        pitch 0.240 0   ; 0.240 counts, hold it there...
        wait 5
        ...
```

- **rest**,**r** **ticks** : rest
  Acts as a key off event if following a note command, or a general delay if used on its own.

```
        ...
        as4 20
        rest 30         ; Key off, wait for 30 ticks
        ...
        rest 60         ; Wait for 60 ticks
        ...
```

- **wait,w ticks** : wait
  Pauses decoding for the specified number of ticks. Useful for delays where note off behavior is not desired.

```
...
wait 60         ; Wait for 60 ticks
...
```

- **using,u instrument** : set current instrument
  Select the instrument used for playback in the music track.

```
...
using piano     ; Using instrument "piano"
g.3 20          ; The following notes will be played with
rest 10         ; "piano," with each note request
b.3 20          ; specifying a key on, and each rest
rest 10         ; indicating a key off.
a.3 20
rest 10
...
```

- **pan,p left right** : set current panning
  Two 8-Bit values representing the panning of the channel in the left and right speakers. These range from -128 to 127, where larger absolute values are louder and zero is silence. Negative values from -1 to -128 will invert the waveform.

```
...
using piano     ; Using instrument "piano"
pan $7f $00     ; Set panning to full volume on the LEFT
as3 12          ; Play the note in the LEFT speaker
rest 24
pan $00 $7f     ; Set panning to full volume on the RIGHT
as3 12          ; Play the note in the RIGHT speaker
rest 24
...
```

- **priority,pr value** : set track priority
  Adjust the current playback priority of the music track. This may be useful to give certain parts of a piece more or less importance relative to sound effects.

```
...
priority 120    ; Set priority to 120
...
priority 240    ; Set priority to 240 (higher)
...
```

- **mood,m value** : set mood flag
  Sets the current mood flag to the supplied value, this may be read by the game software using `ct_getMood()`, and used to adjust game behavior based upon music activity. The mood flag is defaulted to zero (neutral) when a music track is started or stopped.

```
...
mood 13         ; Set mood flag to 13 (user-defined)
...
mood 0          ; Set mood flag to 0 (neutral)
...
```

- **loop,l count** : start of loop
  Specifies the start of a loop, and the number of times it will repeat. If negative values are used, the loop will continue indefinitely.

```
        loop -1                 ; Repeat outer loop forever
                loop 10         ; Repeat inner loop 10 times
                        as4 20
                        rest 20
                endloop
                gs3 30
                rest 20
        endloop
        end                     ; Stop
```

- **endloop,el** : loop end
  Specifies the end of a given loop.

```
        loop -1                 ; Repeat forever
                fs4 20
                rest 30
                call drumSolo
        endloop                 ; Marks end of above loop
        end                     ; Stop
```

- **call,c blockLabel** : call script block
  Begin decoding a given script block with the name specified in **blockLabel**.

```
        ...
        call pianoSolo          ; Call the script block below
        ...
}

pianoSolo
{
        ...
        return
}
```

- **return,rt** : return from called script block
  Resume decoding from where a given script block was called.

```
        ...
        call pianoSolo
        ...
}

pianoSolo
{
        ...
        return          ; Resume decoding after "call pianoSolo"
}
```

- **break,b** : pattern break
  Returns all tracks (not just the one encountering the command) to the lowest entry in their CALL stack. Effectively, all tracks will be returned to the "main" script if they are not there already.

```
mainTrack 120
{
        ...
        call pianoSolo
        ...
}

pianoSolo
{
        ...
        call drumminThang
}

drumminThang
{
        ...
        break           ; Resumes decoding after "call pianoSolo"
}
```

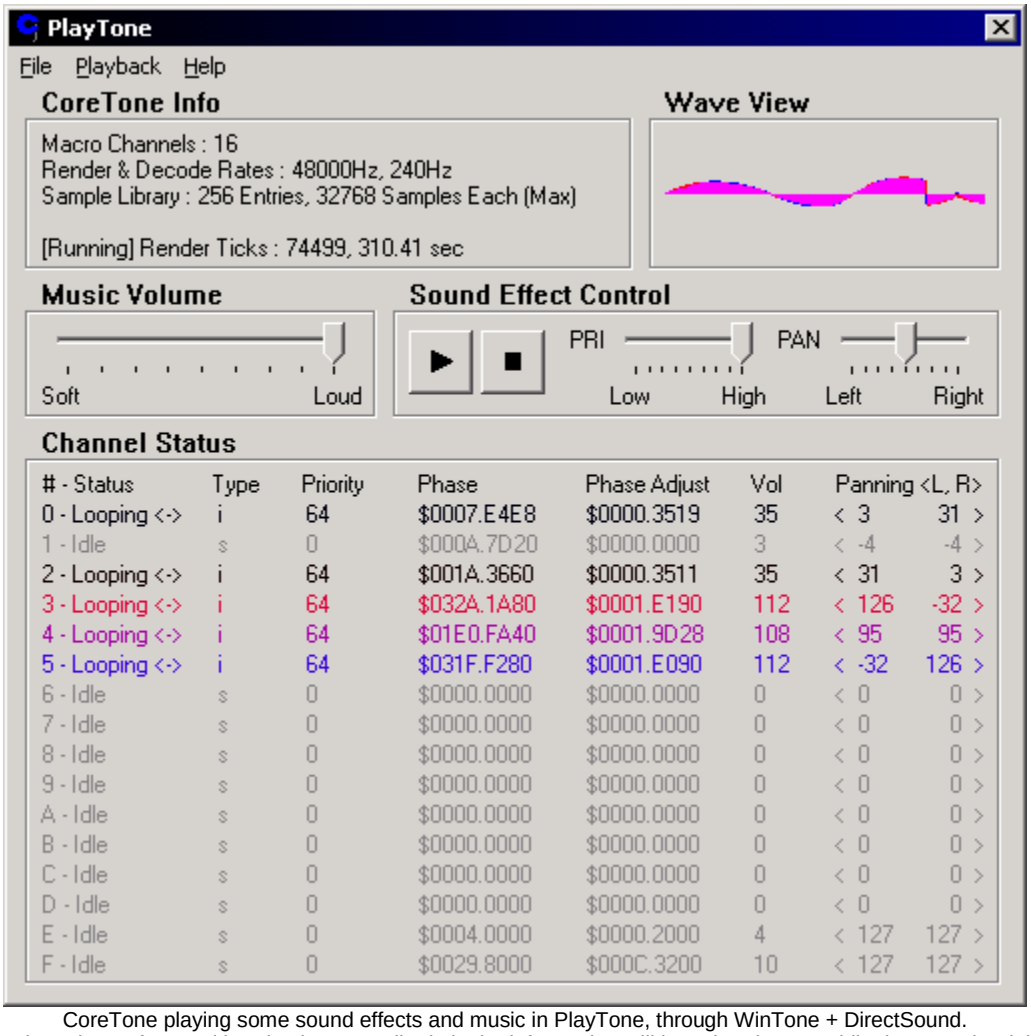
- **end,e**
  Stops decoding of the music track and frees the channel

```
...
end             ; Stop
```

►PlayTone Auditioning Suite

PlayTone is a Windows application for auditioning sample packages, macro instruments, music tracks, and sound effects. The state of CoreTone's macro channels are displayed in order to assist with SASS Script debugging.

Output logging to a `*.WAV` file is supported and will capture all music and sound effects played by the user. The SASS examples included with the BupBoop / CoreTone suite (located in the `./SASS` directory) may be loaded and listened to in PlayTone out of the box.



CoreTone playing some sound effects and music in PlayTone, through WinTone + DirectSound.
Channels and waveforms with a dominant amplitude in the left speaker will be colored **RED**, while those on the right will be colored **BLUE**. If a channel is played evenly on both speakers it will appear **PURPLE**.

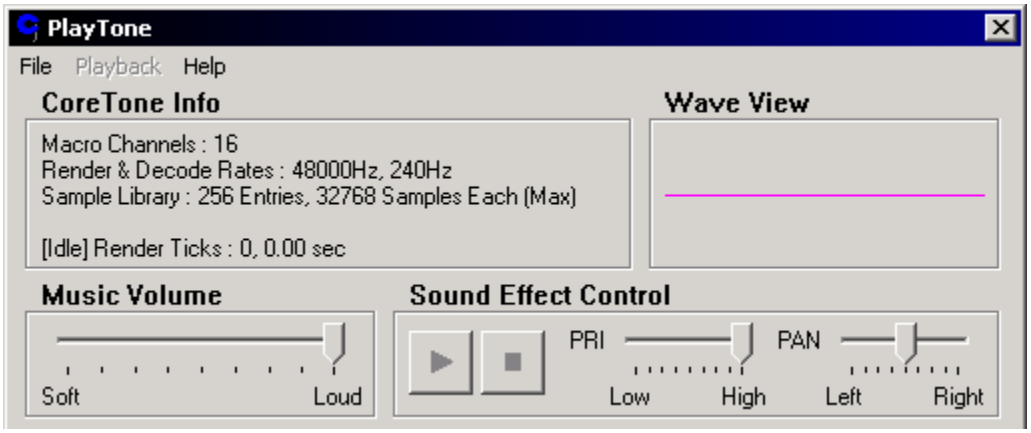
Helpful Shortcuts :

- **`CTRL+P`** *Open Sample Package (`*.smp`)*
- **`CTRL+I`** *Open Macro Instrument Package (`*.ins`)*
- **`CTRL+O`** *Open Music Track (`*.mus`)*
- **`CTRL+E`** *Open Sound Effect (`*.sfx`)*

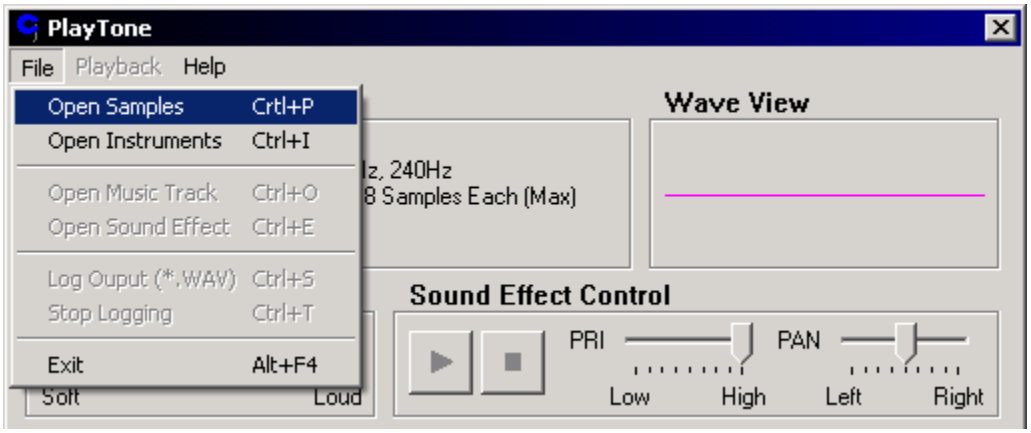
- **`Space`** *Play Music Track*
- **`CTRL+Space`** *Stop Music Track*
- **`X`** *Play Sound Effect*
- **`CTRL+X`** *Stop All Sound Effects*

○Getting Started

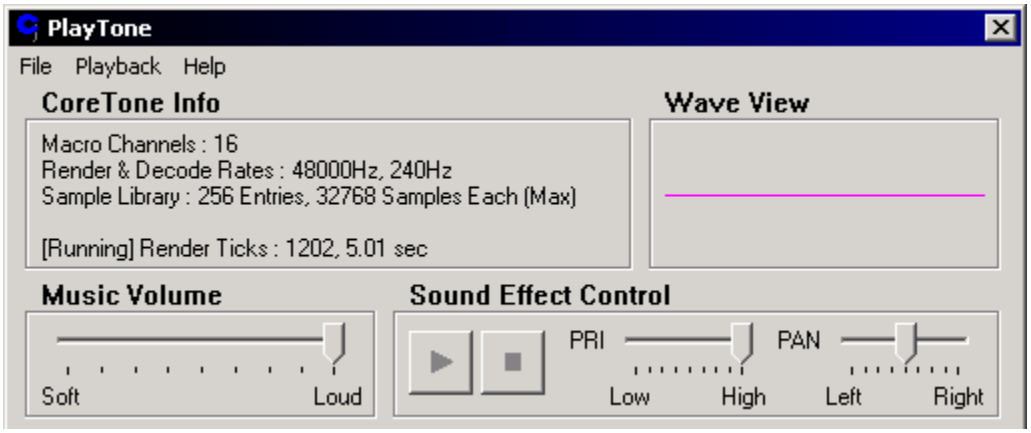
Upon running PlayTone, the user will be greeted with a report of the current CoreTone + WinTone SoftSynth capabilities and will await further input.



Before playing any music or sound effects, a Sample Package (`*.smp`) and Macro Instrument Package (`*.ins`) must be loaded. If you have not built your own, two are included in the `./SASS` directory (`./SASS/CoreSamples.smp` and `./SASS/CoreMacros.ins`).

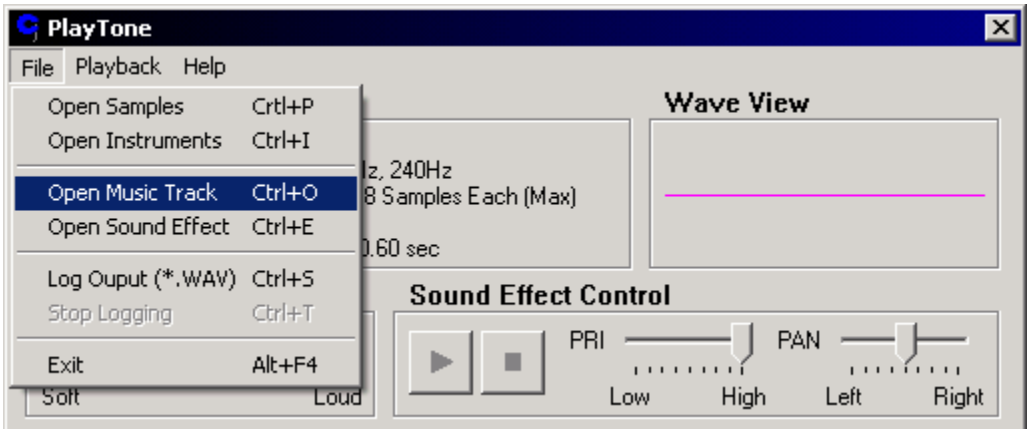


Once the Sample and Macro Instrument Packages have been loaded, PlayTone will try to initialize the SoftSynth. If successful, its state will change from **[Idle]** to **[Running]**.

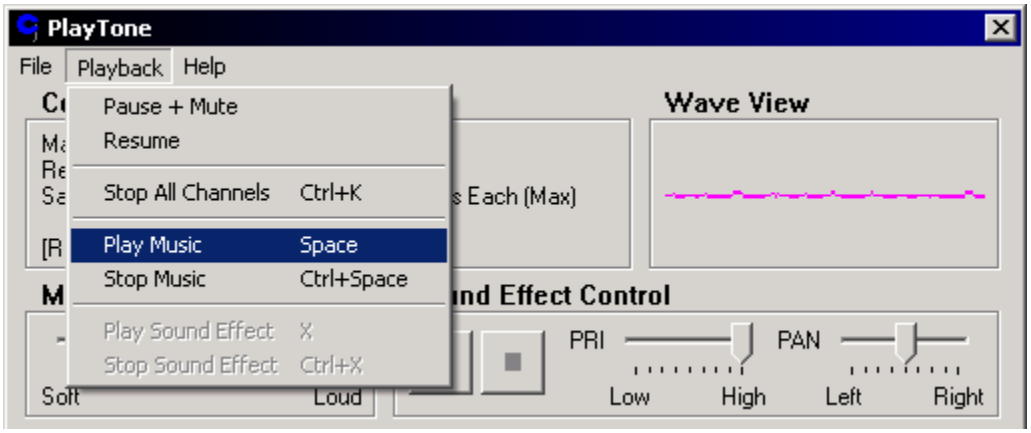


○Playing Music

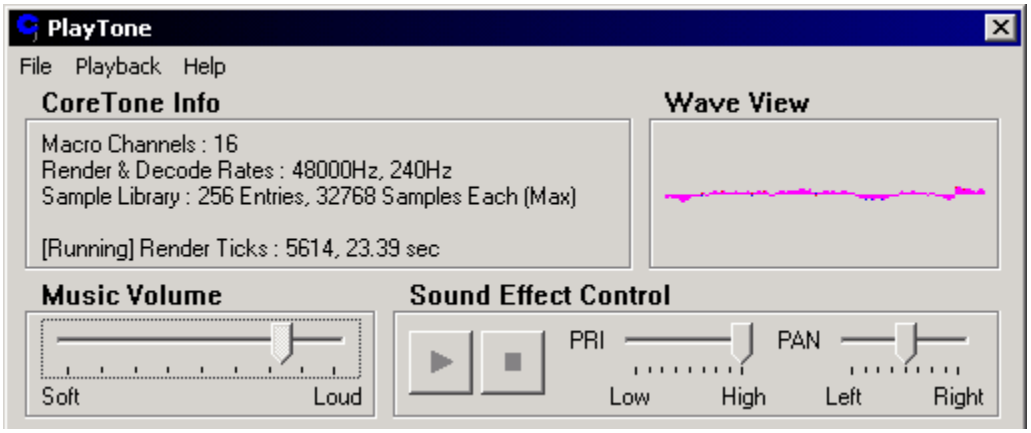
Once both a Sample Package (`*.smp`) and Macro Instrument Package (`*.ins`) have been loaded and the SoftSynth is running, PlayTone can load Music Tracks (`*.mus`). A demo music track is included in `./SASS/DemoMusic/DemoMusic.mus`.



After loading a music track it may be played and stopped using the controls under the **`Playback`** menu or pressing **`Space`** to play and **`CTRL+Space`** to stop.

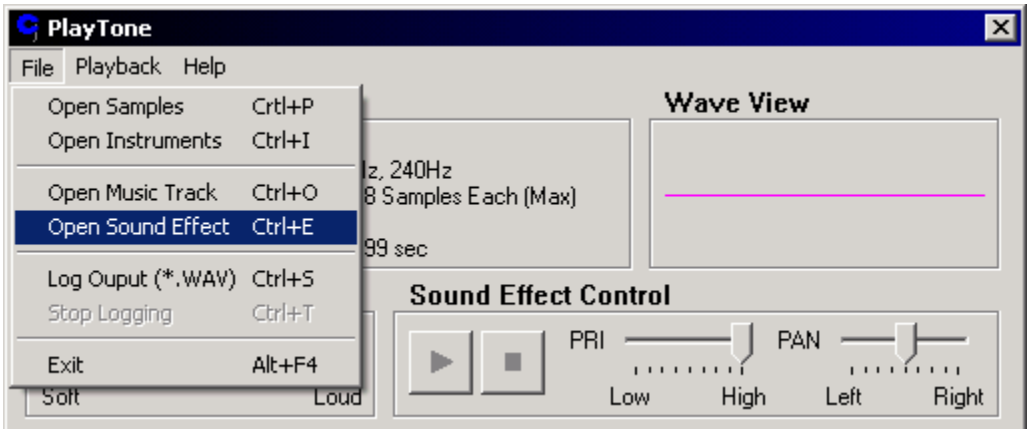


The music volume may be adjusted by using the trackbar on the left, useful for testing fade-out times or balancing music amplitude with sound effects.

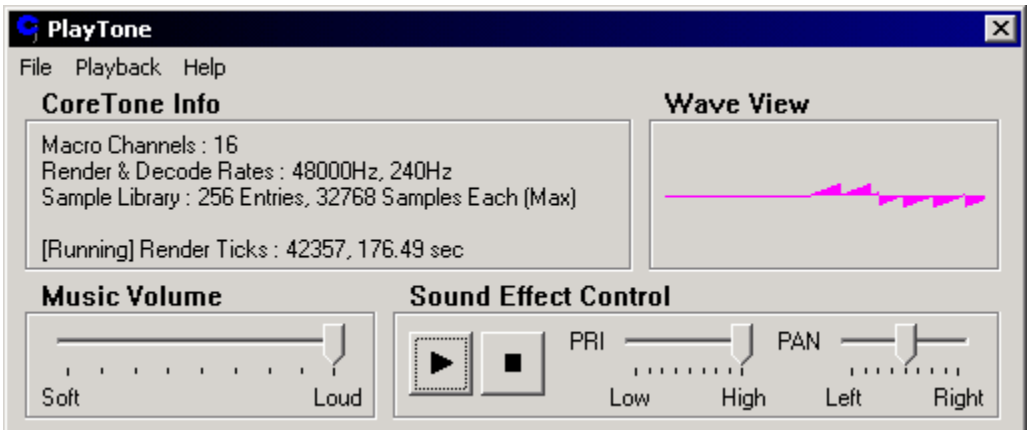


○Playing Sound Effects

As with Music Tracks, once both a Sample and Instrument Package have been loaded, PlayTone will allow Sound Effects (`*.sfx`) to be loaded and played.



Once loaded, a sound effect's playback may be manipulated using the Sound Effect Control box. The **`PLAY`**► and **`STOP`**■ buttons will dispatch and stop instances of the sound effect, respectively. The **`PRI`** and **`PAN`** trackbars control the importance of the sound effect and its stereo panning.



It should be noted that panning a sound effect left or right will only change the dominant playback speaker, sound effects will always be auditioned at full volume. As a precaution, adjusting a sound effect's priority will stop any of its previously dispatched instances.