|  |  |  |
| --- | --- | --- |
| Bochs Developers Guide | | |
| [Prev](http://docs.google.com/sb16-emulation-basics.html) | Chapter 2. About the code | [Next](http://docs.google.com/harddisk-redologs.html) |

2.9. The sound lowlevel interface

This file is intended for programmers who would like to port the sound output routines to their platform. It gives a short outline what services have to be provided.

You should also have a look at the exisiting files, *SOUNDLOW.CC*, *SOUNDMOD.CC* and e.g. *SOUNDLNX.CC* for Linux or *SOUNDWIN.CC* for Windows and their respective header files to get an idea about how these things really work.

2.9.1. Files

The main include file for a lowlevel sound driver is *iodev.h*. It has all definitions for the system-independent functions that a sound driver uses. The sound driver also needs to include *soundlow.h* for the definitions of the base classes *bx\_sound\_lowlevel\_c*, *bx\_soundlow\_waveout\_c*, *bx\_soundlow\_wavein\_c* and *bx\_soundlow\_midiout\_c*.

Additionally, every output driver will have an include file, which should be included on top of soundmod.cc to allow the emulator to use that driver. The code to initialize the object for the selected drivers can be found in that file, so a soundcard emulation does not need to include the specific driver headers.

To actually make the emulator use any specific driver as the default, *BX\_SOUND\_LOWLEVEL\_NAME* has to be set to the name of the respective driver.

Note that if your class contains any system-specific statements, include-files and so on, you should enclose both the include-file and the CC-file in an *#if defined* (OS-define) construct. Also don't forget to add your file to the list of lowlevel sound object files (*SOUNDLOW\_OBJS*) in the file *configure.in* and to regenerate the configure script,

2.9.2. Defines and strutures

#define BX\_SOUNDLOW\_WAVEPACKETSIZE 19200  
  
#define BX\_SOUNDLOW\_OK 0  
#define BX\_SOUNDLOW\_ERR 1  
  
typedef struct {  
 Bit16u samplerate;  
 Bit8u bits;  
 Bit8u channels;  
 Bit8u format;  
 Bit16u volume;  
} bx\_pcm\_param\_t;  
  
const bx\_pcm\_param\_t default\_pcm\_param = {44100, 16, 2, 1};

The maximum size of a wave data packet, the return values of the lowlevel functions, the structure for the PCM parameters and the default parameter set are also important for the sound driver development. They can be found in the main include file *soundlow.h*.

All lowlevel sound methods called from the device code have to return either *BX\_SOUNDLOW\_OK* if the function was successful, or *BX\_SOUNDLOW\_ERR* if not. If any of the initialization functions fail, the device emulation should disable the affected feature.

2.9.3. Classes

The following classes are involved with the sound lowlevel interface:

* *bx\_soundmod\_ctl\_c* is a pseudo device that is used to initialize the sound drivers depending on the configuration.
* *bx\_sound\_lowlevel\_c* is the base class of the lowlevel sound support. It has methods to return pointers to the objects for the available services *waveout*, *wavein* and *midiout*. The base class returns NULL for all services.
* *bx\_sound\_dummy\_c* is derived from *bx\_sound\_lowlevel\_c*. It returns vaild pointers for all services, but the output classes are only implemented as stubs and the *wavein* service returns silence. This "dummy" driver is used whenever a OS specific driver does not implement all services.
* *bx\_soundlow\_waveout\_c*, *bx\_soundlow\_wavein\_c* and *bx\_soundlow\_midiout\_c* are the base classes for the services provided by the Bochs lowlevel sound support. Some methods are stubs and used by the "dummy" sound driver, others are helper methods and used by the OS specific implementations derived from these base classes.
* *bx\_sound\_OS\_c* is derived from *bx\_sound\_lowlevel\_c*. It returns vaild pointers for all services it implements for the selected *OS* (operating system / library) or NULL for services it does not implement. In the second case the Bochs sound init code falls back to the "dummy" driver.

2.9.4. The base class ***bx\_sound\_lowlevel\_c***

class bx\_sound\_lowlevel\_c : public logfunctions {  
public:  
 bx\_sound\_lowlevel\_c();  
 virtual ~bx\_sound\_lowlevel\_c();  
  
 virtual bx\_soundlow\_waveout\_c\* get\_waveout() {return NULL;}  
 virtual bx\_soundlow\_wavein\_c\* get\_wavein() {return NULL;}  
 virtual bx\_soundlow\_midiout\_c\* get\_midiout() {return NULL;}  
  
protected:  
 bx\_soundlow\_waveout\_c \*waveout;  
 bx\_soundlow\_wavein\_c \*wavein;  
 bx\_soundlow\_midiout\_c \*midiout;  
};

The base class for sound lowlevel support is derived from the *logfunctions* class to make the Bochs logging capabilities available in the sound driver code. The constructor of this base class only initializes all pointers to NULL and the destructor deletes the objects if necessary.

2.9.5. The ***waveout*** base class ***bx\_soundlow\_waveout\_c***

class bx\_soundlow\_waveout\_c : public logfunctions {  
public:  
 bx\_soundlow\_waveout\_c();  
 virtual ~bx\_soundlow\_waveout\_c();  
  
 virtual int openwaveoutput(const char \*wavedev);  
 virtual int set\_pcm\_params(bx\_pcm\_param\_t \*param);  
 virtual int sendwavepacket(int length, Bit8u data[], bx\_pcm\_param\_t \*src\_param);  
 virtual int get\_packetsize();  
 virtual int output(int length, Bit8u data[]);  
 virtual int closewaveoutput();  
  
 virtual int register\_wave\_callback(void \*, get\_wave\_cb\_t wd\_cb);  
 virtual void unregister\_wave\_callback(int callback\_id);  
  
 virtual bx\_bool mixer\_common(Bit8u \*buffer, int len);  
protected:  
 void convert\_pcm\_data(Bit8u \*src, int srcsize, Bit8u \*dst, int dstsize, bx\_pcm\_param\_t \*param);  
 void start\_mixer\_thread(void);  
  
 bx\_pcm\_param\_t emu\_pcm\_param, real\_pcm\_param;  
 int cvt\_mult;  
  
 int cb\_count;  
 struct {  
 void \*device;  
 get\_wave\_cb\_t cb;  
 } get\_wave[BX\_MAX\_WAVE\_CALLBACKS];  
 int pcm\_callback\_id;  
};

The base class for wave output support is also derived from the *logfunctions* class. In addition to wave output methods used from sound devices, it contains everything required for the mixer thread feature (register PCM sources, convert data formats, start mixer).

The constructor should *not* allocate the output devices. This should be done in *openwaveoutput()*.

This table shows the waveout class methods, where are they called from and if a platform / library specific implementation is required.

**Table 2-4. Waveout methods**

|  |  |  |
| --- | --- | --- |
| Method | Called from | Platform code |
| *openwaveoutput()* | Sound init code | Required |
| *set\_pcm\_params()* | *openwaveoutput()* and *sendwavepacket()* | Required |
| *sendwavepacket()* | Sound device emulation | Optional |
| *get\_packetsize()* | Mixer thread | Optional |
| *output()* | Mixer thread | Required |
| *closewaveoutput()* | Sound device emulation | Optional |
| *register\_wave\_callback()* | *openwaveoutput()* and sound device emulation | Optional |
| *unregister\_wave\_callback()* | class destructor and sound device emulation | Optional |
| *mixer\_common()* | Mixer thread | Optional |
| *convert\_pcm\_data()* | Internal | No |
| *start\_mixer\_thread()* | Internal | No |

2.9.5.1. int openwaveoutput(const char \*wavedev)

*openwaveoutput()* is called when the sound output subsystem initializes. It should do the following:

* Set up the default PCM parameters for output.
* Open the given device, and prepare it for wave output.
* Register the callback function for the PCM buffer queue (*sendwavepacket()* adds the output to the queue and the mixer thread gets it from there).
* Start the mixer thread, unless the sound library has it's own one (e.g. SDL).

*openwaveoutput()* will only be called once, whereas *set\_pcm\_params()* is called whenever the PCM samplerate has been changed.

The parameters are the following:

* *wavedev* is the wave output device selected by the user. It is strictly system-dependent. Some sound libraries currently ignore this value and use the default one instead. The value is that of the *waveout=device* configuration parameter of the *sound* bochsrc option.

Note that only one wave output device will be used at any one time. *wavedev* may not have the same value throughout one session, but it will be closed before it is changed.

2.9.5.2. int set\_pcm\_params(bx\_pcm\_param\_t \*param)

This function should called from *openwaveoutput()* to initialize the output device with the default parameters and from *sendwavepacket()* whenever the samplerate has been changed in the emulated sound device. It should do the following:

* Open the wave output device, unless *openwaveoutput()* did that already.
* Prepare the device for data and set the device parameters to those given in the function call.

The parameters are the following:

* *param* is a pointer to a structure containing the set of parameters required to set up a sound device for PCM output.

The members of the structure *bx\_pcm\_param\_t* are these:

* *samplerate* is the desired frequency of the output. Because of the capabities of the soundcards, it can have any value between 5000 and 48,000.
* *bits* is either 8 or 16, denoting the resolution of one sample.
* *channels* is the number of channels (2 for stereo output, or 1 for mono output.
* *format* is a bit-coded value (see below).
* *volume* is the output volume to be used by the mixer code. The 16 bit value consists of two 8 bit values for each channel.

**Table 2-5. format bits**

|  |  |
| --- | --- |
| Bit number | Meaning |
| 0 (LSB) | 0: unsigned data  1: signed data |
| 1..6 | Type of codec (see below) |
| 7 | 0: no reference byte  1: with reference byte |
| 8..x | reserved (0) |

**Table 2-6. codecs**

|  |  |
| --- | --- |
| Value | Meaning |
| 0 | PCM (raw data) |
| 1 | reserved |
| 2 | 2-bit ADPCM (Creative Labs format) |
| 3 | 2.4-bit (3-bit) ADPCM (Creative Labs format) |
| 4 | 4-bit ADPCM (Creative Labs format) |

Other codecs are not supported by the SB hardware. In fact, most applications will translate their data into raw data, so that in most cases the codec will be zero.

The number of bytes per sample can be calculated from this as (bits / 8) \* channels.

2.9.5.3. int sendwavepacket(int length, Bit8u data[], bx\_pcm\_param\_t \*src\_param)

This function is called whenever a data packet of at most *BX\_SOUNDLOW\_WAVEPACKETSIZE* is ready at the soundcard emulation. It should then do the following:

* Add this wave packet to the waveout buffer chain after converting to 16 bit signed little endian. If the samplerate has been changed *set\_pcm\_params()* should be called to update the sound hardware settings.

Parameters:

* *length* is the number of data bytes in the data stream. It will never be larger than *BX\_SOUNDLOW\_WAVEPACKETSIZE*.
* *data* is the array of data bytes.
* *src\_param* is a pointer to a structure containing the PCM parameters (see above).

The order of bytes in the data stream is the same as that in the Wave file format:

**Table 2-7. wave output types**

|  |  |
| --- | --- |
| Output type | Sequence of data bytes |
| 8 bit mono | Sample 1; Sample 2; Sample 3; etc. |
| 8 bit stereo | Sample 1, Channel 0; Sample 1, Channel 1; Sample 2, Channel 0; Sample 2, Channel 1; etc. |
| 16 bit mono | Sample 1, LSB; Sample 1, MSB; Sample 2, LSB; Sample 2, MSB; etc. |
| 16 bit stereo | Sample 1, LSB, Channel 0; Sample 1, MSB, Channel 0; Sample 1, LSB, Channel 1; Sample 1, MSB, Channel 1; etc. |

Typically 8 bit data will be unsigned with values from 0 to 255, and 16 bit data will be signed with values from -32768 to 32767, although the soundcard emulations are not limited to this. site.

2.9.5.4. int get\_packetsize()

This function is called from the mixer thread to retrieve the size of a wave data packet based on the current samplerate. By default the packet size is big enough to send output for 0.1 seconds. If the host sound driver / library uses a different value, this value should be returned with this method.

2.9.5.5. int output(int length, Bit8u data[])

This function is called from the mixer thread to send the mixed PCM output to the host sound hardware.

Parameters:

* *length* is the number of data bytes in the data stream. It will never be larger than the value returned from *get\_packetsize*.
* *data* is the array of data bytes.

2.9.5.6. int closewaveoutput()

This function is currently only called from the soundcard emulation if the "file" driver is used. This makes the runtime change of the output file possible. By default this method does nothing and the wave output device is closed in the destructor of the specific class.

2.9.5.7. int register\_wave\_callback(void \*arg, get\_wave\_cb\_t wd\_cb)

This function is called from *openwaveoutput()* to register the function to retrieve data from the PCM output buffer chain. Other sound emulation devices (e.g. OPL3, PC speaker) can register a function to poll the data from the device emulation. The return value is the ID of the registered function and it is usually used to unregister the source.

Parameters:

* *arg* is the pointer to the device emulation object.
* *wd\_cb* is the pointer to a static function that returns wave data from the device emulation. This function is usually called from the *mixer\_common()* method.

2.9.5.8. void unregister\_wave\_callback(int callback\_id)

This function is usually called from the destructor of the sound emulation device to unregister it's registered function to poll PCM data. If the driver / library doesn't use the default mixer thread, a specific implementation of this method my be required.

Parameter:

* *callback\_id* is the ID of the function to unregister.

2.9.5.9. bx\_bool mixer\_common(Bit8u \*buffer, int len)

This is the main wave output mixing function. It is called from the mixer thread, it polls the wave data from all registered sources and it mixes the data using a simple algorithm (addition and clipping). The return value indicates whether or not wave data is available for output.

Parameters:

* *buffer* is the output buffer for the wave data.
* *len* is the maximum length of the output buffer.

2.9.5.10. void convert\_pcm\_data(Bit8u \*src, int srcsize, Bit8u \*dst, int dstsize, bx\_pcm\_param\_t \*param)

This function converts the PCM data sent from the sound device emulation to the 16 bit stereo signed little endian format. It should be called in *sendwavepacket()* after allocating the output buffer in the buffer queue. Future versions might also perform resampling here.

Parameters:

* *src* is the buffer containing data sent from the sound emulation.
* *srcsize* is the amount of wave data to be converted.
* *dst* is the buffer for the converted wave data.
* *dstsize* is the size of the destination buffer.
* *param* is a pointer to the struture containing the format parameters of the source data.

2.9.5.11. void start\_mixer\_thread()

This function starts the mixer thread and it should be called in *openwaveoutput()* unless the sound driver / library has it's own way to do this (e.g. SDL). This function also initializes the mutex required for locking the mixer thread when adding data to the buffer chain or unregistering a source.

2.9.6. The ***wavein*** base class ***bx\_soundlow\_wavein\_c***

class bx\_soundlow\_wavein\_c : public logfunctions {  
public:  
 bx\_soundlow\_wavein\_c();  
 virtual ~bx\_soundlow\_wavein\_c();  
  
 virtual int openwaveinput(const char \*wavedev, sound\_record\_handler\_t rh);  
 virtual int startwaverecord(bx\_pcm\_param\_t \*param);  
 virtual int getwavepacket(int length, Bit8u data[]);  
 virtual int stopwaverecord();  
  
 static void record\_timer\_handler(void \*);  
 void record\_timer(void);  
protected:  
 int record\_timer\_index;  
 int record\_packet\_size;  
 sound\_record\_handler\_t record\_handler;  
};

The base class for wave input support is also derived from the *logfunctions* class. It contains the framework for wave input (recording) support. The base class is used by the "dummy" sound driver and returns silence to let the input mechanism of the soundcard emulation work. The soundcard emulator object needs to implement a callback function to notifies the emulation about available data. This function usually calls the driver method to get the wave data packet. The driver objects has a periodic timer with an interval of 0.1 emulated seconds that is active during recording. The timer handler processes the wave data recorded with platform or library specific function and finally notifies the emulator.

The constructor of the base class only initializes the timer ID. OS specific implementations should initialize other required members here.

The destructor of the base class only calls *stopwaverecord()*. OS specific implementations should close the input device here if necessary.

2.9.6.1. int openwaveinput(char \*device, sound\_record\_handler\_t rh)

*openwaveinput()* is called when the sound emulation first receives a sound recording command. It should do the following:

* Open the given device, and prepare it for wave input

*or*

* Store the device name so that the device can be opened in *startwaverecord()*.

In addition to this the record handler value should be stored and the record timer should be registered. This is the definition of record handler callback function:

typedef Bit32u (\*sound\_record\_handler\_t)(void \*arg, Bit32u len);

*openwaveinput()* will only be called once, whereas *startwaverecord()* is called for every new wave input command to the soundcard emulation. If feasible, it could be useful to open and/or lock the input device in *startwaverecord()* as opposed to *openwaveinput()* to ensure that it can be used by other applications while Bochs doesn't need it.

The parameters are the following:

* *device* is the wave device selected by the user. It is strictly system-dependent. The value is that of the *wavein=device* configuration parameter of the *sound* bochsrc option.
* *rh* is a pointer to the record handler method of the sound emulation. When sound recording is active, this handler is called periodicly to notify the sound emulation about newly available data.

Note that only one wave input device will be used at any one time. *device* may not have the same value throughout one session, but it will be closed before it is changed.

2.9.6.2. int startwaverecord(bx\_pcm\_param\_t \*param)

This method receives a pointer to the required PCM parameters (samplerate, data format) as the argument and it should set up the input device for recording, calculate the size of the recording packet for 0.1 second and start the record timer.

2.9.6.3. int getwavepacket(int length, Bit8u data[])

This method is called from the record handler method of the sound emulation device to retrieve the recorded wave data packet.

2.9.6.4. int stopwaverecord()

This method is called to stop the wave recording. It deactivates the timer that calls the method to perform the recording.

2.9.7. The ***midiout*** base class ***bx\_soundlow\_midiout\_c***

class bx\_soundlow\_midiout\_c : public logfunctions {  
public:  
 bx\_soundlow\_midiout\_c();  
 virtual ~bx\_soundlow\_midiout\_c();  
  
 virtual int openmidioutput(const char \*mididev);  
 virtual int midiready();  
 virtual int sendmidicommand(int delta, int command, int length, Bit8u data[]);  
 virtual int closemidioutput();  
};

The base class for MIDI output support is also derived from the *logfunctions* class.

OS specific implementations should initialize required members in the constructor.

The destructor of the base class only calls *closemidioutput()*. OS specific implementations should close the input device here if necessary.

2.9.7.1. int openmidioutput(char \*device)

* *openmidioutput()* is called when the first midi output starts. It is only called if the midi output to the driver is active (midimode 1). It should prepare the given MIDI hardware for receiving midi commands.

Description of the parameters:

* *mididev* is a system-dependent variable. The value is that of the *midiout=device* configuration parameter of the *sound* bochsrc option.
* Note that only one midi output device will be used at any one time. *device* may not have the same value throughout one session, but it will be closed before it is changed.

2.9.7.2. int midiready()

*midiready()* is called whenever the applications asks if the midi queue can accept more data.

Return values:

* *BX\_SOUNDLOW\_OK* if the midi output device is ready.
* *BX\_SOUNDLOW\_ERR* if it isn't ready.

*Note: midiready()* will be called a few times *before* the device is opened. If this is the case, it should always report that it is ready, otherwise the application (not Bochs) will hang.

2.9.7.3. int sendmidicommand(int delta, int command, int length, Bit8u data[])

*sendmidicommand()*is called whenever a complete midi command has been written to the emulator. It should then send the given midi command to the midi hardware. It will only be called after the midi output has been opened. Note that if at all possible it should not wait for the completion of the command and instead indicate that the device is not ready during the execution of the command. This is to avoid delays in the program while it is generating midi output.

Description of the parameters:

* *delta* is the number of delta ticks that have passed since the last command has been issued. It is always zero for the first command. There are 24 delta ticks per quarter, and 120 quarters per minute, thus 48 delta ticks per second.
* *command* is the midi command byte (sometimes called status byte), in the usual range of 0x80..0xff. For more information please see the midi standard specification.
* *length* is the number of data bytes that are contained in the data structure. This does *not* include the status byte which is not replicated in the data array. It can only be greater than 3 for SysEx messages (commands *0xF0* and *0xF7*)
* *data[]* is the array of these data bytes, in the order they have in the standard MIDI specification. Note, it might be *NULL* if length==0.

2.9.7.4. int closemidioutput()

*closemidioutput()* is called before shutting down Bochs or when the emulator gets the *stop\_output* command through the emulator port. After this, no more output will be necessary until *openmidioutput()* is called again, but *midiready()* might still be called. It should do the following:

* Wait for all remaining messages to be completed
* Reset and close the midi output device

|  |  |  |
| --- | --- | --- |
| [Prev](http://docs.google.com/sb16-emulation-basics.html) | [Home](http://docs.google.com/index.html) | [Next](http://docs.google.com/harddisk-redologs.html) |
| Sound Blaster 16 Emulation | [Up](http://docs.google.com/about-the-code.html) | Harddisk Images based on redologs |