

---

## Oto (v3)



A low-level library to play sound.

- Oto (v3)
  - Platforms
  - Prerequisite
    - \* macOS
    - \* iOS
    - \* Linux
    - \* FreeBSD, OpenBSD
  - Usage
    - \* Playing sounds from memory
    - \* Playing sounds by file streaming
    - \* Advanced usage
  - Crosscompiling

### Platforms

- Windows (no Cgo required!)
- macOS (no Cgo required!)
- Linux
- FreeBSD
- OpenBSD
- Android
- iOS
- WebAssembly
- Nintendo Switch
- Xbox

### Prerequisite

On some platforms you will need a C/C++ compiler in your path that Go can use.

- iOS: On newer macOS versions type `clang` on your terminal and a dialog with installation instructions will appear if you don't have it

- 
- If you get an error with clang use xcode instead `xcode-select --install`
  - Linux and other Unix systems: Should be installed by default, but if not try GCC or Clang

## macOS

Oto requires `AudioToolbox.framework`, but this is automatically linked.

## iOS

Oto requires these frameworks:

- `AVFoundation.framework`
- `AudioToolbox.framework`

Add them to “Linked Frameworks and Libraries” on your Xcode project.

## Linux

ALSA is required. On Ubuntu or Debian, run this command:

```
1 apt install libasound2-dev
```

On RedHat-based linux distributions, run:

```
1 dnf install alsa-lib-devel
```

In most cases this command must be run by root user or through `sudo` command.

## FreeBSD, OpenBSD

BSD systems are not tested well. If ALSA works, Oto should work.

## Usage

The two main components of Oto are a `Context` and `Players`. The context handles interactions with the OS and audio drivers, and as such there can only be **one** context in your program.

From a context you can create any number of different players, where each player is given an `io.Reader` that it reads bytes representing sounds from and plays.

Note that a single `io.Reader` must **not** be used by multiple players.

---

## Playing sounds from memory

The following is an example of loading and playing an MP3 file:

```
1 package main
2
3 import (
4     "time"
5     "os"
6
7     "github.com/ebitengine/oto/v3"
8     "github.com/hajimehoshi/go-mp3"
9 )
10
11 func main() {
12     // Read the mp3 file into memory
13     fileBytes, err := os.ReadFile("./my-file.mp3")
14     if err != nil {
15         panic("reading my-file.mp3 failed: " + err.Error())
16     }
17
18     // Convert the pure bytes into a reader object that can be used
19     // with the mp3 decoder
20     fileBytesReader := bytes.NewReader(fileBytes)
21
22     // Decode file
23     decodedMp3, err := mp3.NewDecoder(fileBytesReader)
24     if err != nil {
25         panic("mp3.NewDecoder failed: " + err.Error())
26     }
27
28     // Prepare an Oto context (this will use your default audio device)
29     // that will
30     // play all our sounds. Its configuration can't be changed later.
31     op := &oto.NewContextOptions{}
32
33     // Usually 44100 or 48000. Other values might cause distortions in
34     // Oto
35     op.SampleRate = 44100
36
37     // Number of channels (aka locations) to play sounds from. Either 1
38     // or 2.
39     // 1 is mono sound, and 2 is stereo (most speakers are stereo).
40     op.ChannelCount = 2
41
42     // Format of the source. go-mp3's format is signed 16bit integers.
43     op.Format = oto.FormatSignedInt16LE
44
45     // Remember that you should **not** create more than one context
46     otoCtx, readyChan, err := oto.NewContext(op)
```

---

```
44     if err != nil {
45         panic("oto.NewContext failed: " + err.Error())
46     }
47     // It might take a bit for the hardware audio devices to be ready,
48     // so we wait on the channel.
49     <-readyChan
50     // Create a new 'player' that will handle our sound. Paused by
51     // default.
52     player := otoCtx.NewPlayer(decodedMp3)
53     // Play starts playing the sound and returns without waiting for it
54     // (Play() is async).
55     player.Play()
56     // We can wait for the sound to finish playing using something like
57     // this
58     for player.IsPlaying() {
59         time.Sleep(time.Millisecond)
60     }
61     // Now that the sound finished playing, we can restart from the
62     // beginning (or go to any location in the sound) using seek
63     // newPos, err := player.(io.Seeker).Seek(0, io.SeekStart)
64     // if err != nil{
65     //     panic("player.Seek failed: " + err.Error())
66     // }
67     // println("Player is now at position:", newPos)
68     // player.Play()
69     // If you don't want the player/sound anymore simply close
70     err = player.Close()
71     if err != nil {
72         panic("player.Close failed: " + err.Error())
73     }
74 }
```

## Playing sounds by file streaming

The above example loads the entire file into memory and then plays it. This is great for smaller files but might be an issue if you are playing a long song since it would take too much memory and too long to load.

In such cases you might want to stream the file. Luckily this is very simple, just use `os.Open`:

```
1 package main
2
3 import (
4     "bytes"
```

---

```

5     "os"
6     "time"
7
8     "github.com/hajimehoshi/go-mp3"
9     "github.com/hajimehoshi/oto/v3"
10 )
11
12 func main() {
13     // Open the file for reading. Do NOT close before you finish
    playing!
14     file, err := os.Open("./my-file.mp3")
15     if err != nil {
16         panic("opening my-file.mp3 failed: " + err.Error())
17     }
18
19     // Decode file. This process is done as the file plays so it won't
20     // load the whole thing into memory.
21     decodedMp3, err := mp3.NewDecoder(file)
22     if err != nil {
23         panic("mp3.NewDecoder failed: " + err.Error())
24     }
25
26     // Rest is the same...
27
28     // Close file only after you finish playing
29     file.Close()
30 }

```

The only thing to note about streaming is that the *file* object must be kept alive, otherwise you might just play static.

To keep it alive not only must you be careful about when you close it, but you might need to keep a reference to the original file object alive (by for example keeping it in a struct).

## Advanced usage

Players have their own internal audio data buffer, so while for example 200 bytes have been read from the `io.Reader` that doesn't mean they were all played from the audio device.

Data is moved from `io.Reader`→internal buffer→audio device, and when the internal buffer moves data to the audio device is not guaranteed, so there might be a small delay. The amount of data in the buffer can be retrieved using `Player.UnplayedBufferSize()`.

The size of the underlying buffer of a player can also be set by type-asserting the player object:

```
1 myPlayer.(oto.BufferSizeSetter).SetBufferSize(newBufferSize)
```

This works because players implement a `Player` interface and a `BufferSizeSetter` interface.

---

## Crosscompiling

Crosscompiling to macOS or Windows is as easy as setting `G00S=darwin` or `G00S=windows`, respectively.

To crosscompile for other platforms, make sure the libraries for the target architecture are installed, and set `CGO_ENABLED=1` as Go disables Cgo on crosscompiles by default.