# Building an application with PocketSphinx



> ***Caution!***
> *This tutorial describes pocketsphinx 5 pre-alpha release.*
> *It is not going to work for older versions.*

## Installation

PocketSphinx is a library that depends on another library called SphinxBase which provides common functionality across all CMUSphinx projects. To install Pocketsphinx, you need to install both Pocketsphinx and Sphinxbase. You can use Pocketsphinx with Linux, Windows, on MacOS, iPhone and Android.

First of all, download the released packages for pocketsphinx and sphinxbase from the projects download page or check them out from Subversion or Github.

For more details see the download page.

Unpack them into the same directory. On Windows, you will need to rename 'sphinxbase-X.Y' (where X.Y is the SphinxBase version number) to simply 'sphinxbase' to satisfy the project configuration of pocketsphinx.

### Installation on Unix system

To build pocketsphinx in a Unix-like environment (such as Linux, Solaris, FreeBSD etc.) you need to make sure you have the following dependencies installed: gcc, automake, autoconf, libtool, bison, swig (at least version 2.0), the Python development package and the pulseaudio development package.

If you want to build it without some dependencies you can use the respective configuration options like `--without-swig-python`. However, for beginners it's recommended to install all dependencies.

You need to download both the sphinxbase and pocketsphinx packages and unpack them. Please note that you cannot use sphinxbase and pocketsphinx of different version. So, please make sure that their versions are in sync. After unpacking, you should see the following two main folders:

```
sphinxbase-X.X
pocketsphinx-X.x
```

First, build and install SphinxBase. Change the current directory to the `sphinxbase` folder. If you downloaded it directly from the repository, you need to run the following command at least once to generate the `configure` file:

```
./autogen.sh
```

If you downloaded the release version, or ran `autogen.sh` at least once, then compile and install it with:

```
./configure
make
make install
```

The last step might require root permissions, so you might need to run `sudo make install`. If you want to use fixed-point arithmetic, you must configure SphinxBase with the `--enable-fixed` option. You can also set an installation prefix with `--prefix` or configure to use or not to use SWIG Python support.

Sphinxbase will be installed in the `/usr/local/` directory by default. Not all systems load libraries from this folder automatically. In order to load them you need to configure the path to look for

shared libaries. This can be done either in the `/etc/ld.so.conf` file or by exporting the environment variables:

```
export LD_LIBRARY_PATH=/usr/local/lib
export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
```

For more details on the linker configuration see the Shared Libraries HOWTO.

Then change to the pocketsphinx folder and perform the same steps:

```
./configure
make
make install
```

To test the installation, run `pocketsphinx_continuous -inmic yes` and check that it recognizes words you speak into your microphone.

If you get an error such as: `error while loading shared libraries: libpocketsphinx.so.3`, you may want to check your linker configuration of the `LD_LIBRARY_PATH` environment variable described above.

**Windows**

In MS Windows, under MS Visual Studio 2012 (or newer – we test with Visual C++ 2012 Express):

- load `sphinxbase.sln` in the sphinxbase directory
- compile all the projects in SphinxBase (from `sphinxbase.sln`)
- load `pocketsphinx.sln` in the pocketsphinx directory
- compile all the projects in PocketSphinx

MS Visual Studio will build the executables and libraries under `.\bin\Release` or `.\bin\Debug` (depending on the target you choose on MS Visual Studio). In order to run `pocketsphinx_continuous.exe`, don't forget to copy the sphinxbase.dll file to the bin folder. Otherwise the executable will fail to find this library. Unlike on Linux, the path to the model is not preconfigured in Windows, so you have to specify for pocketsphinx_continuous where to find the model with the `-hmm`, `-lm` and `-dict` options.

To recognize speech from your microphone, change to the pocketsphinx folder and run:

```
bin\Release\Win32\pocketsphinx_continuous.exe -inmic yes
    -hmm model\en-us\en-us -lm model\en-us\en-us.lm.bin
    -dict model\en-us\cmudict-en-us.dict
```

To recognize speech from a file run:

```
bin\Release\Win32\pocketsphinx_continuous.exe -infile test\data\goforward.raw
    -hmm model\en-us\en-us -lm model\en-us\en-us.lm.bin
    -dict model\en-us\cmudict-en-us.dict
```

## Pocketsphinx API core ideas

The Pocketsphinx API is designed to ease the use of speech recognizer functionality in your applications:

1. It is very likely to remain stable both in terms of source and binary compatibility, due to the use of abstract types.
2. It is fully re-entrant, so there is no problem having multiple decoders in the same process.
3. It allows a drastic reduction in code footprint and a modest but significant reduction in memory consumption.

The reference documentation for the new API is available at https://cmusphinx.github.io/doc/pocketsphinx/.

## Basic usage (hello world)

There are a few key things you need to know when you want to use the API:

1. command-line parsing is done externally (in `<cmd_ln.h>`)
2. everything takes a `ps_decoder_t *` as the first argument

To illustrate the API, we will step through a simple "hello world" example. This example is somewhat specific to Unix regarding the locations of files and the compilation process. We will create a C source file called `hello_ps.c`. To compile it (on Unix), use this command:

```
gcc -o hello_ps hello_ps.c \
    -DMODELDIR=\"`pkg-config --variable=modeldir pocketsphinx`\" \
    `pkg-config --cflags --libs pocketsphinx sphinxbase`
```

Please note that compilation errors mean that you didn't carefully follow the installation guide above. For example pocketsphinx needs to be properly installed to be available through the pkg-config system. To check whether pocketsphinx is installed properly, just run

```
pkg-config --cflags --libs pocketsphinx sphinxbase
```

on the command line and make sure the output looks like this:

```
-I/usr/local/include -I/usr/local/include/sphinxbase -I/usr/local/include/pocketsphinx
        -L/usr/local/lib -lpocketsphinx -lsphinxbase -lsphinxad
```

### Initialization

The first thing we need to do is to create a configuration object, which for historical reasons is called `cmd_ln_t`. Along with the general boilerplate for our C program, our code looks like this:

```
#include <pocketsphinx.h>

int
main(int argc, char *argv[])
{
    ps_decoder_t *ps = NULL;
    cmd_ln_t *config = NULL;

    config = cmd_ln_init(NULL, ps_args(), TRUE,
                         "-hmm", MODELDIR "/en-us/en-us",
                         "-lm", MODELDIR "/en-us/en-us.lm.bin",
                         "-dict", MODELDIR "/en-us/cmudict-en-us.dict",
                         NULL);

    return 0;
}
```

The `cmd_ln_init()` function takes a variable number of null-terminated string arguments, followed by NULL. The first argument is any previous `cmd_ln_t *` which is to be updated. The second argument is an array of argument definitions – the standard set can be obtained by calling `ps_args()`. The third argument is a flag telling the argument parser to be "strict". If this argument is `TRUE`, then duplicate arguments or unknown arguments will cause the parsing process to fail.

The `MODELDIR` macro is defined on the GCC command-line by using the `pkg-config` to obtain the `modeldir` variable from the PocketSphinx configuration. On Windows, you can simply add a preprocessor definition to the code, such as this:

```
#define MODELDIR "c:/sphinx/model"
```

(replace this with wherever your models are installed). In order to initialize the decoder, use `ps_init`:

```
ps = ps_init(config);
```

### Decoding a file stream

Because live audio input is somewhat platform-specific, we will confine ourselves to decoding audio files. There is an audio file helpfully included in the pocketsphinx source code which contains this very sentence. You can find it in `pocketsphinx/test/data/goforward.raw`. Copy it to the current directory. If you want to create your own version of it: it needs to be a single-channel (monaural), little-endian, unheadered 16-bit signed PCM audio file sampled at 16000 Hz.

The main pocketsphinx use case is to read audio data in blocks of memory from a source and feed it to the decoder. To do that, we first open the file and start decoding the utterance using `ps_start_utt()`:

```
rv = ps_start_utt(ps);
```

Next, we read 512 samples at a time from the file, and feed them to the decoder using `ps_process_raw()`:

```
int16 buf[512];
while (!feof(fh)) {
    size_t nsamp;
    nsamp = fread(buf, 2, 512, fh);
    ps_process_raw(ps, buf, nsamp, FALSE, FALSE);
}
```

Afterwards, we need to mark the end of the utterance using `ps_end_utt()`:

```
    rv = ps_end_utt(ps);
```

Last, we retrieve the hypothesis to get our recognition result:

```
    hyp = ps_get_hyp(ps, &score);
    printf("Recognized: %s\n", hyp);
```

We can also retrieve the hypothesis during recognition. If we do so, it will return a partial result.

### Cleaning up

To clean up, simply call `ps_free()` on the object that was returned by `ps_init()`. Free the configuration object with `cmd_ln_free_r`.

### Full code listing

Here is the full listing of our code again:

```
#include <pocketsphinx.h>

int
main(int argc, char *argv[])
{
    ps_decoder_t *ps;
```

```
    cmd_ln_t *config;
    FILE *fh;
    char const *hyp, *uttid;
    int16 buf[512];
    int rv;
    int32 score;

    config = cmd_ln_init(NULL, ps_args(), TRUE,
                         "-hmm", MODELDIR "/en-us/en-us",
                         "-lm", MODELDIR "/en-us/en-us.lm.bin",
                         "-dict", MODELDIR "/en-us/cmudict-en-us.dict",
                         NULL);
    if (config == NULL) {
        fprintf(stderr, "Failed to create config object, see log for details\n");
        return -1;
    }

    ps = ps_init(config);
    if (ps == NULL) {
        fprintf(stderr, "Failed to create recognizer, see log for details\n");
        return -1;
    }

    fh = fopen("goforward.raw", "rb");
    if (fh == NULL) {
        fprintf(stderr, "Unable to open input file goforward.raw\n");
        return -1;
    }

    rv = ps_start_utt(ps);

    while (!feof(fh)) {
        size_t nsamp;
        nsamp = fread(buf, 2, 512, fh);
        rv = ps_process_raw(ps, buf, nsamp, FALSE, FALSE);
    }

    rv = ps_end_utt(ps);
    hyp = ps_get_hyp(ps, &score);
    printf("Recognized: %s\n", hyp);

    fclose(fh);
    ps_free(ps);
    cmd_ln_free_r(config);

    return 0;
}
```

## Advanced usage

For more advanced uses of the API please check the API reference.

- For word segmentations, the API provides an iterator object which is used to iterate over the sequence of words. This iterator object is an abstract type, with some accessors provided to obtain timepoints, scores and, most interestingly, posterior probabilities for each word.
- The confidence of the whole utterance can be accessed with the `ps_get_prob` method.
- You can access the lattice if needed.
- You can configure multiple searches and switch between them in runtime.

### Searches

As a developer you can configure several “search” objects with different grammars and language models and switch between them during runtime to provide interactive experience for the user.

There are multiple possible search modes:

- *keyword*: efficiently looks for a keyphrase and ignores other speech. It Allows to configure the detection threshold.
- *grammar*: recognizes speech according to the JSGF grammar. Unlike keyphrase search, grammar search doesn’t ignore words which are not in the grammar but tries to recognize them.
- *ngram/lm*: recognizes natural speech with a language model.
- *allphone*: recognizes phonemes with a phonetic language model.

Each search has a name and can be referenced by a name. Names are application-specific. The function `ps_set_search` allows to activate the search that was previously added by a name.

In order to add a search, one needs to point to the grammar/language model describing the search. The location of the grammar is specific to the application. If only a simple recognition is required it is sufficient to add a single search or to just configure the required mode using configuration options.

The exact design of a search depends on your application. For example, you might want to listen for an activation keyword first and once this keyword is recognized switch to ngram search to recognize the actual command. Once you recognized the command you can switch to grammar search to recognize the confirmation and then switch back to keyword listening mode to wait for another command.

← Building an application with Sphinx4

Using PocketSphinx on Android →

## Contact us

Forum

Mailing List

Issue Tracker

Telegram

## Links

AlphaCephei

LinkedIn

## Feeds

Posts