

# Flite: a small, fast speech synthesis engine

---

System documentation  
Edition 2.0, for Flite version 2.0  
18th November 2014

by Alan W Black and Kevin A. Lenzo

---

Copyright © 2001-2014 Carnegie Mellon University, all rights reserved.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Carnegie Mellon University

# 1 Abstract

This document provides a user manual for flite, a small, fast run-time speech synthesis engine.

This manual is nowhere near complete.

Flite offers text to speech synthesis in a small and efficient binary. It is designed for embedded systems like PDAs as well large server installation which must serve synthesis to many ports. Flite is part of the suite of free speech synthesis tools which include Edinburgh University's Festival Speech Synthesis System <http://www.festvox.org/festival> and Carnegie Mellon University's FestVox project <http://festvox.org>, which provides tools, scripts, and documentation for building new synthetic voices.

Flite is written in ANSI C, and is designed to be portable to almost any platform, including very small hardware.

Flite is really just a synthesis library that can be linked into other programs, it includes two simple voices with the distribution, an old diphone voice and an example limited domain voice which uses the newer unit selection techniques we have been developing. Neither of these voices would be considered production voices but serve as examples, new voices will be released as they are developed.

The latest versions, comments, new voices etc for Flite are available from its home page which may be found at

<http://cmuflite.org>



## 2 Copying

Flite is free software. It is distributed under an X11-like license. Apart from the few exceptions noted below (which still have similarly open licenses) the general license is

Language Technologies Institute  
Carnegie Mellon University  
Copyright (c) 1999-2014  
All Rights Reserved.

Permission is hereby granted, free of charge, to use and distribute this software and its documentation without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of this work, and to permit persons to whom this work is furnished to do so, subject to the following conditions:

1. The code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Any modifications must be clearly marked as such.
3. Original authors' names are not deleted.
4. The authors' names are not used to endorse or promote products derived from this software without specific prior written permission.

CARNEGIE MELLON UNIVERSITY AND THE CONTRIBUTORS TO THIS WORK DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY NOR THE CONTRIBUTORS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.



### 3 Acknowledgements

The initial development of flite was primarily done by awb while travelling, perhaps the name is doubly appropriate as a substantial amount of the coding was done over 30,000ft). During most of that time awb was funded by the Language Technologies Institute at Carnegie Mellon University.

Kevin A. Lenzo was involved in the design, conversion techniques and representations for the voice distributed with flite (as well as being the actual voice itself).

Other contributions are:

- Nagoya Institute of Technology The MLSA, MLPG code comes directly NITECH's hts engine code, though we have done some optimizations.
- Marcela Charfuelan (DFKI) For the mixed-excitation techniques (but no direct code). These originally came from NITECH but we understood the techniques from Marcela's Open Mary Java code and implemented them in our optimized version of MLSA.
- David Huggins-Daines: much of the very early clunits code, porting to multiple platforms, substantial code tidy up and configure/autoconf guidance (up to 2001).
- Cepstral, LLC (<http://cepstral.com>): For supporting DHD to spend time (in early 2001) on flite and passing back the important fixes and enhancements while on a project funded by the Portuguese Foundation for Science and Technology (FCT) Praxis XXI program specifically to produce an open source synthesizer.
- Willie Walker <william.walker@sun.com> and the Sun Speech Group: lots of low level bugs (and fixes).
- Henry Spencer: For the regex code
- University of Edinburgh: for releasing Festival for free, making a companion runtime synthesizer a practical project, much of the design of flite relies on the architecture decisions made in the Festival Speech Synthesis Systems and the Edinburgh Speech Tools.

The duration cart tree and intonation (accent and F0) models for the US English voice were derived from the models in the Festival distribution. which in turn were trained from the Boston University FM Radio Data Corpus.

- Carnegie Mellon University The included lexicon is derived from CMULEX and the letter to sound rules are constructed using the Lenzo and Black techniques for building LTS decision graphs.
- Craig Reese: IDA/Supercomputing Research Center and Joe Campbell: Department of Defense who wrote the ulaw conversion routines in `src/speech/cst_wave_utils.c`





## 4 Installation

Flite consist simple of a set of C files. GNU configure is used to configure the engine and will work on most major architectures.

In general, the following should build the system

```
tar zxvf flite-XXX.tar.gz
cd flite-XXX
./configure
make
```

However you will need to explicitly call GNU make **gmake** if **make** is not GNU make on your system.

The configuration process build a file **config/config** which under some circumstances may need to be edited, e.g. to add unusual options or dealing with cross compilation.

On Linux systems, we also support shared libraries which are useful for keeping space down when multiple different application are linked to the flite libraries. For development we strongly discourage use of shared libraries as it is too easy to either not set them up correctly or accidentally pick up the wrong version. But for installation they are definitely encouraged. That is if you are just going to make and install they are good but unless you know what **LD\_LIBRARY\_PATH** does, it may be better to use static libraries (the default) if you are changing C code or building your own voices.

```
./configure --enable-shared
make
```

This will build both shared and static versions of the libraries but will link the executables to the *shared* libraries thus you will need to install the libraries in a place that your dynamic linker will find them (cf. `/etc/ld.so.conf`) or set **LD\_LIBRARY\_PATH** appropriately.

```
make install
```

Will install the binaries (**bin/flite\***), include files and libraries in appropriate subdirectories of the defined install directory, **/usr/local** by default. You can change this at configure time with

```
./configure --prefix=/opt
```

### 4.1 Windows Support

### 4.2 Window CE Support

*NOTE: as Windows CE is somewhat rare now, we do not guarantee this still works.*

Flite has been successfully compiled by a number of different groups under Windows CE. The system should compile under Embedded Visual Studio but we not have the full details.

The system as distributed does compile under the gcc **mingw32ce** toolchain available from <http://cegcc.sourceforge.net/>. The current version can be compiled and run under WinCE with a primitive application called **flowm**. **flowm** is a simple application that allows playing of typed-in text, or full text to speech on a file. Files should be a simple ascii

text files `*.txt`. The application allows the setting of the byte position to start synthesis from.

Assuming you have `mingw32ce` installed you can configure as

```
./configure --target=arm-wince
make
```

The resulting binary is given in `wince/flowm.exe`. If you copy this onto your Windows Mobile device and run it, it should allow you to speak typed-in text and any `*.txt` files you have on your device.

The application uses `cmu_us_kal` as the voice for default. Although it is possible to include the clusterген voices, they may be too slow to be really practical. An 8KHz clusterген voice with a reduced order to 13 gives a voices that runs acceptably on an hp2755 (624MHz) but still marginal on an AT&T Tilt (400MHz).

Building 8KHz clusterген voices is currently a bit of hack. We take the standard waveforms and resample them to 8KHz, then relabel the sample rate to be 16KHz. Then build the voice as normal (as if the speaker spoke twice as fast. You may need to have tune the F0 parameters in `etc/f0.params`. This seems to basically work. Then after the waveform is synthesized (still in the "chipmunk" domain) we then playit back at 8KHz. This effectively means we generate half the number of samples and the frames are really at 10ms. A second reduction is an option on the basic `build_flite` command. A second argument can specify order reduction, thus instead of the standard 25 static parameters (plus its deltas) we can reduce this to 13 and still get acceptable results

```
./bin/build_flite cg 13
cd flite
make
```

Importantly this uses less space, and uses less time to synthesis. These `SPEECH_HACKS` in `src/cg/cst_mlsa.c` are switched on by default when `UNDER_CE` is defined.

The reduced order properly extracts the statics (and stddev) and deltas (and stddev) from the predicted parameter clusters and makes it as if those were the sizes of parameters that were used to the train the voice.

## 4.3 PalmOS Support

*NOTE: as PalmOS is somewhat rare now, we do not guarantee this still works.*

Support for PalmOS was removed from 1.9, I no longer have any working PalmOS devices. But this section remains for people who do, but they may need to update something to make this work.

Starting with 1.3 we have initial support for PalmOS using the free development tools. The compilation method assumes the target device is running PalmOS 5.0 (or later) on an ARM processor. Following convention in the Palm world, the app that the user interacts with is actually a m68k application compiled with the m68 gcc cross compiler, the resulting code is interpreted by the PalmOS 5.0 device. The core flite code is in native ARM, and hence uses the ARM gcc cross compiler. An interesting amount of support code is required to get all this work properly.

The user app is called `flop` (FLite on Palm) and like most apps written by awb, is functional, but ugly. You should not let a short-sighted Scotsman, who still thinks command

line interfaces are cool, design a graphical app. But it does work and can read typed-in text. The `armflite.ro` resources are designed with the idea that proper applications will be written using it as a library.

The `flop.prc` application is distributed separately so it can be used without having to install all these tools. But if you want to PalmOS development here is what you need to do to compile Flite for PalmOS and the flop application.

There are number of different application development environments for Palm, here I only describe the Unix based one as this is what was used. You will need the PalmOS SDK 5.0 from palmOne <http://www.palmone.com/us/developers/>. This is free but does require registration. Out of the lots of different files you can get for palmOne you will eventually find `palmos-sdk-5.0r3-1.noarch.rpm`, install that on your linux machine

```
rpm -i palmos-sdk-5.0r3-1.noarch.rpm
```

You will also need the various gcc based cross compilers <http://prc-tools.sourceforge.net/>

```
prc-tools-2.3-1.i386.rpm
```

```
prc-tools-arm-2.3-1.i386.rpm
```

```
prc-tools-htmldocs-2.3-1.noarch.rpm
```

The Palm Resource compiler <http://pilrc.sourceforge.net/>

```
pilrc-3.1-1.i386.rpm
```

And maybe the emulator <http://www.palmos.com/dev/tools/emulator/>

```
pose-3.5-2.i386.rpm
```

```
pose-skins-1.9-1.noarch.rpm
```

```
pose-skins-handspring-3.1H4-1.noarch.rpm
```

Though as POSE doesn't support ARM code, Simulator does but that only works under Windows, POSE is only useful for debugging the m68k parts of the app.

Install these

```
rpm -i prc-tools-2.3-1.i386.rpm
```

```
rpm -i prc-tools-arm-2.3-1.i386.rpm
```

```
rpm -i prc-tools-htmldocs-2.3-1.noarch.rpm
```

```
rpm -i pilrc-3.1-1.i386.rpm
```

```
rpm -i pose-3.5-2.i386.rpm
```

```
rpm -i pose-skins-1.9-1.noarch.rpm
```

```
rpm -i pose-skins-handspring-3.1H4-1.noarch.rpm
```

We also need the prc-tools to know which SDK is available

```
palmdev-prep
```

In addition we use Greg Parker's PEAL <http://www.sealiesoftware.com/peal/> ELF ARM loader. You need to download this and compile and install it yourself, so that `peal-postlink` is in your path. Greg was very helpful and even added support for large data segments for this work (though in the end we don't actually use them). Some peal code is in our distribution (which is valid under his licence) but if you use a different version of peal you may need to ensure they are matched, by updating the peal code in `palm/`. We used version `peal-2004-12-29`.

The other palm specific function we require is `par` <http://www.djw.org/product/palm/par/> which is part of the `prc.tgz` distribution. We use `par` to construct resources from raw

binary files. There are other programs that can do this but we found this one adequate. Again you must compile this and ensure `par` is in your path. Note no part of `par` ends up in the distributed system.

Given all of the above you should be able to compile the Palm code and the `flop` application.

```
./configure --target=arm-palmos
make
```

The resulting application should be in `palm/flop/flop.prc` which can then be installed on your Plam device

```
pilot-xfer -i palm/flop/flop.prc
```

Setting up the tools, and getting a working Linux/Palm conduit is not particularly easy but it is possible. Although some attempt was made to use the Simulator, (PalmOS 5.0/ARM simulator) under Windows, it never really contributed to the development. The POSE (m68k) emulator though was use to develop the `flop` application itself.

### 4.3.1 Some notes on the PalmOS port

Throughout the PalmOS developer documentation they continually remind you that a Palm device is not a full computer, its an extention of the desktop. But seeing devices like the Treo 600 can easily make one forget and want the device to do real computational work. PalmOS is designed for small light weight devices so it is easy to start hitting the boundaries of its capabilities when trying to port larger applications.

PalmOS5.0 still has interesting limitations, in the m68k domain, `int`'s are 16 bit and using memory segments greater than 65K require special work. Quaint as these are, they do significantly affect the port. At first we thought that only the key computationally expensive parts would be in ARM (so-called `armlets`) but trying to compile the whole flite code in m68k with long/short distinctions and sub-64K code segment limitations was just too hard.

Thus all the Flite code, USEnglish, Lexicon and diphone databases actually are compiled in the ARM domain. There is however no system support in the ARM domain so call backs to m68k system functions are necessary. With care calls to system functions can be significantly limited so only a few call backs needed to be written. These are in `palm/pocore/`. I believe CodeWarrior has better support for this, but in this case we rolled our own (though help from other open source examples was important).

We manage the m68k/ARM interface through PEAL, which is basically a linker for ARM code and calling mechanism from m68k. PEAL deals with globals and splitting the code into 65K chunks automatically.

Flite does however have a number of large data segments, in the lexicon and the voice database itself. PEAL can deal with this but it loads large segments by copying them into the dynamic heap, which on most Palm device is less than 2M. This isn't big enough.

Thus we changed Flite to restrict the number of large data sgements it used (and also did some new compression on them). The five segments: the lts rules, the lexical entries, the voice LPC coefficients, the voice residuals and the voice residual index are now treated a data segments that are split into 65400 sized segments and loaded into feature memory space, which is in the storage heap and typically much bigger. This means we do need

about 2-3 megabyte free on the device to run. We did look into just indexing the 65400 byte segments directly but that looked like being too much work, and we're only going to be able to run on 16M sized Palms anyway (there aren't any 8M ARM Palms with audio, expect maybe some SmartPhones).

Using Flite from m68k land involves getting a `flite_info` structure from `flite_init()`. This contains a bunch of fields that be set and sent to the ARM domain Flite synthesizer proper within which other output fields may be set and returned. This isn't a very general structure, but is adequate. Note the necessary byte swapping (for the top level fields) is done for the this structure, before calling the ARM native `arm_flite_synth_text` and swapped back again after returning.

Display, playing audio, pointy-clicky event thingies are all done in the m68K domain.

### 4.3.2 Using the PalmOS

There are three basic functions that access the ARM flite functions: `flite_init()`, `flite_synth_text()` and `flite_end()`.



## 5 Flite Design

### 5.1 Background

Flite was primarily developed to address one of the most common complaints about the Festival Speech Synthesis System. Festival is large and slow, even with the software bloat common amongst most products and that that bloat has helped machines get faster, have more memory and large disks, still Festival is criticized for its size.

Although sometimes this complaint is unfair, it is valid and although much work was done to ensure Festival can be trimmed and run fast it still requires substantial resources per utterance to run. After some investigation to see if Festival itself could be trimmed down it became clear because there was a core set of functions that were sufficient for synthesis that a new implementation containing only those aspects that were necessary would be easier than trimming down Festival itself.

Given that a new implementation was being considered a number of problems with Festival could also be addressed at the same time. Festival is not thread-safe, and although it runs under Windows, in server mode it relies on the Unix-centric view of fast forks with copy-on-write shared memory for servicing clients. This is a perfectly safe and practical solution for Unix systems, but under Windows where threads are the more common feature used for servicing multiple events and forking is expensive, a non-thread safe program can't be used as efficiently.

Festival is written in C++ which was a good decision at the time and perfectly suitable for a large program. However what was discovered over the years of development is that C++ is not a portable language. Different C++ compilers are quite different and it takes significant amount of work to ensure compatibility of the code base over multiple compilers. What makes this worse is that new versions of each compiler are incompatible and changes are required. At first this looked like we were producing bad quality code but after 10 years it is clear that it is also that the compilers are still maturing. Thus it is clear that Festival and the Edinburgh Speech Tools will continue to require constant support as new versions of compilers are released.

A second problem with C++ is the size and efficiency of the code produced. Proponents of C++ may rightly argue that Festival and the Edinburgh Speech Tools aren't properly designed, but irrespective if that is true or not, it is true that the size of the code is much larger and slower than it need be for what it does. Throughout the design there is a constant trade-off between elegancy and efficiency which unfortunately at times in Festival requires untidy solutions of copying data out of objects processing it and copying back because direct access (particularly in some signal processing routines) is just too inefficient.

Another major criticism of Festival is the use of Scheme as the interpreter language. Even though it is a simple to implement language that is adequate for Festival's needs and can be easily included in the distribution, people still hate it. Often these people do learn to use it and appreciate how run time configurability is very desirable and that new voices may be added without recompilation. Scheme does have garbage collection which makes leaky programs much harder to write and as some of the intended audience for developing in Festival will not be hard core programmers a safe programming language seems very desirable.

After taking into consideration all of the above it was decided to develop Flite as a new system written in ANSI C. C is much more portable than C++ as well as offering much lower level control of the size of the objects and data structure it uses.

Flite is not intended as a research and development platform for speech synthesis, Festival is and will continue to be the best platform for that. Flite however is designed as a run-time engine when an application needs to be delivered. It specifically addresses two communities. First as a engine for small devices such as PDAs and telephones where the memory and CPU power are limited and in some cases do not even have a conventional operating system.

The second community is for those running synthesis servers for many clients. Here although large fixed databases are acceptable, the size of memory required per utterance and speed in which they can be synthesized is crucial.

However in spite of the decision to build a new synthesis engine we see this as being tightly coupled into the existing free software synthesis tools or Festival and the FestVox voice building suite. Flite offers a companion run-time engine. Our intended mode of development is to build new voices in FestVox and debug and tune them in Festival. Then for deployment the FestVox format voice may be (semi-)automatically compiled into a form that can be used by Flite.

In case some people feel that development of a small run-time synthesizer is not an appropriate thing to do within a University and is more suited to commercial development, we have a few points which they should be aware of that to our mind justify this work.

We have long felt that research in speech and language should have an identifiable link to ultimate commercial use. In providing a platform that can be used in consumer products that falls within the same framework as our research we can better understand what research issues are actually important to the improvement our work.

In considering small useful synthesizers it forces a more explicit definition of what is necessary in a synthesizer and also how we can trade size, flexibility and speed with the quality of synthesized output. Defining that relationship is a research issue.

We are also advocates of speech technology within other research areas and the ability to offer support on new platforms such as PDAs and wearables allows for more interesting speech applications such as speech-to-speech translation, robots, and interactive personal digital assistants, that will prove new and interesting areas of research. Thus having a platform that others around us can more easily integrate into their research makes our work more satisfying.

## 5.2 Key Decisions

The basic architecture of Festival is good. It is well proven. Paul Taylor, Alan W. Black and Richard Caley spent many hours debating low level aspects of representation and structure that would both be adequate for current theories but also allow for future theories too. The heterogeneous relation graphs (HRG) are theoretically adequate, computationally efficient and well proven. Thus both because HRGs have such a background and that Flite is to be compatible with voices and models developed in Festival, Flite uses HRGs as its basic utterance representation structure.

Most of a synthesizer is in its data (lexicons, unit database etc), the actual synthesis code is pretty small. In Festival most of that data exists in external files which are loaded on



demand. This is obviously slow and memory expensive (you need both a copy on the data on disk and in memory). As one of the principal targets for Flite is very small machines we wanted to allow that core data to be in ROM, and be appropriately mapped into RAM without any explicit loading (some OS's call this XIP – execute in place). This can be done by various memory mapping functions (in Unix its called mmap) and is the core technique used in shared libraries (called DLLs in some parts of the world). Thus the data should be in a format that it can be directly accessed. If you are going to directly access data you need to ensure the byte layout is appropriate for the architecture you are running on, byte order and address width become crucial if you want to avoid any extra conversion code at access time (like byte swapping).

At first it was considered that synthesis data would be converted in binary files which could be mmap'ed into the runtime systems but building appropriate binaries files for architectures is quite a job. However the C compiler does this in a standard way. Therefore the mode of operation for data within Flite is to convert it to C code (actually C structures) and use the C compiler to generate the appropriate binary structures.

Using the C compiler is a good portable solution but it as these structures can be very big this can tax the C compiler somewhat. Also because this data is not going to change at run time it can all be declared `const`. Which means (in Unix) it will be in the text segment and hence read only (this can be ROM on platforms which have that distinction). For structures to be `const` all their subparts must also be `const` thus all relevant parts must be in the same file, hence the unit databases files can be quite big.

Of course, this all presumes that you have a C compiler robust enough to compile these files, hardware smart enough to treat flash ROM as memory rather than disk, or an operating system smart enough to demand-page executables. Certain "popular" operating systems and compilers fail in at least one of these respects, and therefore we have provided the flexibility to use memory-mapped file I/O on voice databases, where available, or simply to load them all into memory.



## 6 Structure

The flite distribution consists of two distinct parts:

- The flite library containing the core synthesis code
- Voice(s) for flite. These contain three sub-parts
  - Language models: text processing, prosody models etc.
  - Lexicon and letter to sound rules
  - Unit database and voice definition

### 6.1 `cst_val`

This is a basic simple object which can contain ints, floats, strings and other objects. It also allows for lists using the Scheme/Lisp, `car/cdr` architecture (as that is the most efficient way to represent arbitrary trees and lists).

The `cst_val` structure is carefully designed to take up only 8 bytes (or 16 on a 64-bit machine). The multiple union structure that it can contain is designed so there are no conflicts. However it depends on the fact that a pointer to a `cst_val` is guaranteed to lie on an even address boundary (which is true for all architectures I know of). Thus the distinction between `cons` (i.e. list) objects and atomic values can be determined by the odd/evenness of the least significant bits of the first address in a `cst_val`. In some circles this is considered hacky, in others elegant. This was done in flite to ensure that the most common structure is 8 bytes rather than 12 which saves significantly on memory.

All `cst_val`'s except those of type `cons` are reference counted. A few functions generate new lists of `cst_val`'s which the user should be careful about as they need to explicitly delete them (notably the lexicon lookup function that returns a list of phonemes). Everything that is added to an utterance will be deleted (and/or dereferenced) when the utterance is deleted.

Like Festival, user types can be added to the `cst_vals`. In Festival this can be done on the fly but because this requires the updating of some list when each new type is added, this wouldn't be thread safe. Thus an explicit method of defining user types is done in `src/utls/cst_val_user.c`. This is not as neat as defining on the fly or using a registration function but it is thread safe and these user types won't change often.



## 7 APIs

Flite is a library that we expected will be embedded into other applications. Included with the distribution is a small example executable that allows synthesis of strings of text and text files from the command line.

You may want to look at Bard <http://festvox.org/bard/>, an ebook reader with a tight coupling to flite as a synthesizer. This is the most elaborate use of the Flite API within our suite of programs.

### 7.1 flite binary

The example flite binary may be suitable for very simple applications. Unlike Festival its start up time is very short (less than 25ms on a PIII 500MHz) making it practical (on larger machines) to call it each time you need to synthesize something.

```
flite TEXT OUTPUTTYPE
```

If TEXT contains a space it is treated as a string of text and converted to speech, if it does not contain a space TEXT is treated as a file name and the contents of that file are converted to speech. The option `-t` specifies TEXT is to be treat as text (not a filename) and `-f` forces treatment as a file. Thus

```
flite -t hello
```

will say the word "hello" while

```
flite hello
```

will say the content of the file `hello`. Likewise

```
flite "hello world."
```

will say the words "hello world" while

```
flite -f "hello world"
```

will say the contents of a file `hello world`. If no argument is specified text is read from standard input.

The second argument OUTPUTTYPE is the name of a file the output is written to, or if it is `play` then it is played to the audio device directly. If it is `none` then the audio is created but discarded, this is used for benchmarking. If it is `stream` then the audio is streamed through a call back function (though this is not particularly useful in the command line version. If OUTPUTTYPE is omitted, `play` is assumed. You can also explicitly set the outputtype with the `-o` flag.

```
flite -f doc/alice -o alice.wav
```

### 7.2 Voice selection

All the voices in the distribution are collected into a single simple list in the global variable `flite_voice_list`. You can select a voice from this list from the command line

```
flite -voice awb -f doc/alice -o alice.wav
```

And list which voices are currently supported in the binary with

```
flite -lv
```

The voices which get linked together are those listed in the **VOICES** in the **main/Makefile**. You can change that as you require.

Voices may also be dynamically loaded from files as well as built in. The argument to the **-voice** option may be pathname to a dumped (Clustergen) voice. This may be a Unix pathname or a URL (only protocols **http** and **file** are supported. For example

```
flite -voice file://cmu_us_awb.flitevox -f doc/alice -o alice.wav
flite -voice http://festvox.org/voices/cmu_us_ksp.flitevox -f doc/alice -o alice.wav
```

Voices will be loaded once and added to **flite\_voice\_list**. Although these voices are often small (a few megabytes) there will still be some time required to read them in the first time. The voices are not mapped, they are read into newly created structures.

This loading function is currently only supported for Clustergen voices.

### 7.3 C example

Each voice in Flite is held in a structure, a pointer to which is returned by the voice registration function. In the standard distribution, the example diphone voice is **cmu\_us\_kal**.

Here is a simple C program that uses the flite library

```
#include <flite/flite.h>

cst_voice * register_cmu_us_kal(const char *voxdir);

int main(int argc, char **argv)
{
    cst_voice *v;

    if (argc != 2)
    {
        fprintf(stderr,"usage: flite_test FILE\n");
        exit(-1);
    }

    flite_init();

    v = register_cmu_us_kal(NULL);

    flite_file_to_speech(argv[1],v,"play");

}
```

Assuming the shell variable **FLITEDIR** is set to the flite directory the following will compile the system (with appropriate changes for your platform if necessary).

```
gcc -Wall -g -o flite_test flite_test.c -I$FLITEDIR/include
-L$FLITEDIR/lib -lflite_cmu_us_kal -lflite_usenglish
-lflite_cmulex -lflite -lm
```

## 7.4 Public Functions

Although, of course you are welcome to call lower level functions, there are a few key functions that will satisfy most users of flite.

```
void flite_init(void);
```

This must be called before any other flite function can be called. As of Flite 1.1, it actually does nothing at all, but there is no guarantee that this will remain true.

```
cst_wave *flite_text_to_wave(const char *text, cst_voice *voice);
```

Returns a waveform (as defined in `include/cst_wave.h`) synthesized from the given text string by the given voice.

```
float flite_file_to_speech(const char *filename, cst_voice *voice, const char *outtype);
```

synthesizes all the sentences in the file `filename` with given voice. Output (at present) can only reasonably be, `play` or `none`. If the feature `file_start_position` with an integer, that point is used as start position in the file to be synthesized.

```
float flite_text_to_speech(const char *text, cst_voice *voice, const char *outtype);
```

synthesizes the text in string point to by `text`, with the given voice. `outtype` may be a filename where the generated waveform is written to, or `"play"` and it will be sent to the audio device, or `"none"` and it will be discarded. The return value is the number of seconds of speech generated.

```
cst_utterance *flite_synth_text(const char *text, cst_voice *voice);
```

synthesize the given text with the given voice and returns an utterance from it for further processing and access.

```
cst_utterance *flite_synth_phones(const char *phones, cst_voice *voice);
```

synthesize the given phones with the given voice and returns an utterance from it for further processing and access.

```
cst_voice *flite_voice_select(const char *name);
```

returns a pointer to the voice named `name`. Will return NULL if there is not match, if `name == NULL` then the first voice in the voice list is returned. If `name` is a url (starting with `file:` or `http:`, that file will be accessed and the voice will be downloaded from there.

```
float flite_ssml_file_to_speech(const char *filename, cst_voice *voice, const char *outtype);
```

Will read the file as ssml, not all ssml tags are supported but many are, unsupported ones are ignored. Voice selection works by naming the internal name of the voice, or the name may be a url and the voice will be loaded. The audio tag is supported for loading waveform files, again urls are supported.

```
float flite_ssml_text_to_speech(const char *text, cst_voice *voice, const char *outtype);
```

Will treat the text as ssml.

```
int flite_voice_add_lex_addenda(cst_voice *v, const cst_string *lexfile);
```

loads the pronunciations from `lexfile` into the lexicon identified in the given voice (which will cause all other voices using that lexicon to also get this new addenda list. An example lexicon file is given in `flite/tools/examples.lex`. Words may be in double quotes, an optional part of speech tag may be give. A colon separates the headword/postag from the list of phonemes. Stress values (if used in the lexicon) must be specified. Bad phonemes will be complained about on standard out.

## 7.5 Streaming Synthesis

In 1.4 support was added for streaming synthesis. Basically you may provide a call back function that will be called with waveform data immediately when it is available. This potentially can reduce the delay between sending text to the synthesized and having audio available.

The support is through a call back function of type

```
int audio_stream_chunk(const cst_wave *w, int start, int size,
                      int last, cst_audio_streaming_info *asi)
```

If the utterance feature `streaming_info` is set (which can be set in a voice or in an utterance). The LPC or MLSA resynthesis functions will call the provided function as buffers become available. The LPC and MLSA waveform synthesis functions are used for diphones, limited domain, unit selection and clusterngen voices. Note explicit support is required for streaming so new waveform synthesis function may not have the functionality.

An example streaming function is provided in `src/audio/au_streaming.c` and is used by the example flite main program when `stream` is given as the playing option. (Though in the command line program the function it isn't really useful.)

In order to use streaming you must provide call back function in your particular thread. This is done by adding features to the voice in your thread. Suppose your function was declared as

```
int example_audio_stream_chunk(const cst_wave *w, int start, int size,
                              int last, void *user)
```

You can add this function as the streaming function through the statement

```
cst_audio_streaming_info *asi;
...
asi = new_audio_streaming_info();
asi->asc = example_audio_stream_chunk;
feat_set(voice->features,
         "streaming_info",
         audio_streaming_info_val(asi));
```

You may also optionally include your own pointer to any information you additionally want to pass to your function. For example

```
typedef my_callback_struct {
    cst_audiodev *fd;
    int count;
};
```



```
cst_audio_streaming_info *asi;

...

mcs = cst_alloc(my_callback_struct,1);
mcs->fd=NULL;
mcs->count=1;

asi = new_audio_streaming_info();
asi->asc = example_audio_stream_chunk;
asi->userdata = mcs;
feat_set(voice->features,
         "streaming_info",
         audio_streaming_info_val(asi));
```

Another example is given in `testsuite/by_word_main.c` which shows a call back function that also prints the token as it is being synthesized. The `utt` field in the `cst_audio_streaming_info` structure will be set to the current utterance. Please note that the `item` field in the `cst_audio_streaming_info` structure is for your convenience and is not set by anyone at all. The previous sentence exists in the documentation so that I can point at it, when user's fail to read it.



## 8 Converting FestVox Voices

As of 1.2 initial scripts have been added to aid the conversion of FestVox voices to Flite. In general the conversion cannot be automatic. For example all specific Scheme code written for a voice needs to be hand converted to C to work in Flite, this can be a major task.

Simple conversion scripts are given as examples of the stages you need to go through. These are designed to work on standard (English) diphone sets, and simple limited domain voices. The conversion technique will almost certainly fail for large unit selection voices due to limitations in the C compiler (more discussion below). In 1.4 we have also added support for converting clustergen voices too (which is a little easier, see section below).

### 8.1 Cocantenative Voice Building

Conversion is basically taking the description of units (clunit catalogue or diphone index) and constructing some C files that can be compiled to form a usable database. Using the C compiler to generate the object files has the advantage that we do not need to worry about byte order, alignment and object formats as the C compiler for the particular target platform should be able to generate the right code.

Before you start ensure you have successfully built and run your FestVox voice in Festival. Flite is not designed as a voice building/debugging tool it is just a delivery vehicle for finalized voices so you should first ensure you are satisfied with the quality of Festival voices before you start converting it for Flite.

The following basic stages are required:

- Setup the directories and copy the conversion scripts
- Build the LPC files
- Build the MCEP files (for ldom/clunits)
- Convert LPC (MCEP) into STS (short term signal) files
- Convert the catalogue/diphone index
- Compile the generated C code

The conversion assumes the environment variable `FLITEDIR` is set, for example

```
export FLITEDIR=/home/awb/projects/flite/
```

The basic flite conversion takes place within a FestVox voice directory. Thus all of the conversion scripts expect that the standard files are available. The first task is to build some new directories and copy in the build scripts. The scripts are copied rather than linked from the Flite directories as you may need to change these for your particular voices.

```
$FLITEDIR/tools/setup_flite
```

This will read `etc/voice.defs`, which should have been created by the FestVox build process (except in very old versions of FestVox).

If you don't have a `etc/voice.defs` you can construct one with `festvox/src/general/guess_voice_defs` in the Festvox distribution, or generate one by hand making it look like

```
FV_INST=cmu
FV_LANG=us
FV_NAME=ked_timit
```

```
FV_TYPE=clunits
FV_VOICENAME=$FV_INST_"$FV_LANG_"$FV_NAME
FV_FULLVOICENAME=$FV_VOICENAME_"$FV_TYPE"
```

The main script build building the Flite voice is `bin/build_flite` which will eventually build sufficient C code in `flite/` that can be compiled with the constructed `flite/Makefile` to give you a library that can be linked into applications and also an example `flite` binary with the constructed voice built-in.

You can run all of these stages, except the final make, together by running the the build script with no arguments

```
./bin/build_flite
```

But as things may not run smoothly, we will go through the stages explicitly.

The first stage is to build the LPC files, this may have already been done as part of the diphone building process (though probably not in the `ldom/clunit` case). In our experience it is very important that the records be of similar power, as mis-matched power can often cause overflows in the resulting flite (and sometimes Festival) voices. Thus, for diphone voices, it is important to run the power normalization techniques described in the FestVox document. The Flite LPC build process also builds a parameter file of the ranges of the LPC parameters used in later coding of the files, so even if you have already built your LPC files you should still do this again

```
./bin/build_flite lpc
```

For `ldom`, and `clunit` voices (but not for diphone voices) we also need the Mel-frequency Cepstral Coefficients. These are assumed to have been cleared and are in `mcep/` as they are necessary for running the voice in Festival. This stage simply constructs information about the range of the `mcep` parameters.

```
./bin/build_flite mcep
```

The next stage is to construct the STS files. Short Term Signals (STS) are built for each pitch period in the database. These are ascii files (one for each utterance file in the database, with LPC coefficients, and ulaw encoded residuals for each pitch period. These are built using a binary executable built as part of the Flite build (`flite/tools/find_sts`.

```
./bin/build_flite sts
```

Note that the flite code expects waveform files to be in Microsoft RIFF format and cannot deal with files in other formats. Some earlier versions of the Edinburgh Speech Tools used NIST as the default header format. This is likely to cause flite and its related programs not work. So do ensure your waveform files are in riff format (`ch_wave -info wav/*` will tell you the format). And the following will convert all your wave files

```
mv wav wav.nist
mkdir wav
cd wav.nist
for i in *.wav
do
    ch_wave -otype riff -o ../wav/$i $i
done
```

The next stage is to convert the index to the required C format. For diphone voices this takes the `dic/*.est` index files, for `clunit/ldom` voices it takes the

`festival/clunit/VOICE.catalogue` and `festival/trees/VOICE.tree` files. This process uses a binary executable built as part of the Flite build process (`flite/tools/flite_sort`) to sort the indices into the same sorting order required for flite to run. (Using `unix sort` may or may not give the same result due to definitions of lexicographic order so we use the very same function in C that will be used in flite to ensure that a consistent order is given.)

```
./bin/build_flite idx
```

All the necessary C files should now have been built in `flite/` and you may compile them by

```
cd flite
make
```

This should give a library and an executable called `flite` that can run as

```
./flite "Hello World"
```

Assuming a general voice. For `ldom` voices it will only be able to say things in its domain. This `flite` binary offers the same options as standard the standard `flite` binary compiled in the Flite build but with your voice rather than the distributed voices.

Almost certainly this process will not run smoothly for you. Building voices is still a very hard thing to do and problems will probably exist.

This build process does not deal with customization for the given voices. Thus you will need to edit `flite/VOICE.c` to set intonation ranges and duration stretch for your particular voice.

For example in our `cmu_us_sls_diphone` voice (a US English female diphone voice). We had to change the default parameters from

```
feat_set_float(v->features,"int_f0_target_mean",110.0);
feat_set_float(v->features,"int_f0_target_stddev",15.0);

feat_set_float(v->features,"duration_stretch",1.0);
```

to

```
feat_set_float(v->features,"int_f0_target_mean",167.0);
feat_set_float(v->features,"int_f0_target_stddev",25.0);

feat_set_float(v->features,"duration_stretch",1.0);
```

Note this conversion is limited. Because it depends on the C compiler to do the final conversion into binary object format (a good idea in general for portability), you can easily generate files too big for the C compiler to deal with. We have spent some time investigating this so the largest possible voices can be converted but it is still too limited for our larger voices. In general the limitation seems to be best quantified by the number of pitch periods in the database. After about 100k pitch periods the files get too big to handle. There are probably solutions to this but we have not yet investigated them. This limitation doesn't seem to be an issue with the diphone voices as they are typically much smaller than unit selection voices.

## 8.2 Statistical Voice Building (Clustergen)

The process of building from a clustergen (cg) voice is also supported. It is assumed the environment variable `FLITEDIR` is set

```
export FLITEDIR=/home/awb/projects/flite/
```

After you build the clustergen voice you can convert by first setting up the skeleton files in the `flite/` directory

```
$FLITEDIR/tools/setup_flite
```

Assuming `etc/voice.defs` properly identifies the voice the cg templates will be compiled in.

The conversion itself is actually much faster than a clunit build (there is less to actually convert).

```
./bin/build_flite cg
```

Will convert then necessary models into files in the `flite/` directory. The you can compile it with

```
cd flite
make
./flite_cmu_us_awb "Hello world"
```

Note that the voice that is to be converted *\*must\** be a standard clustergen voice with `f0`, `mceps`, `delta mceps` (optionally strengths for mixed excitation) and voicing in its combined coeffs files. The method could be changed to deal with other possibilities but it will only work for default build method.

The generated library `libflite_cmu_us_awb.a` may be linked with other programs like any other flite voice. The binary generated `flite_cmu_us_awb` links in only one voice (unlike the flite binary in the full flite distribution).

A single flat file contain the cg voice can also be generated that can be loaded at run time into the flite binary. You can dump this file from the initial constructed flite binary

```
./flite_cmu_us_awb -voicedump cmu_us_awb.flitevox
```

The file `cmu_us_awb.flitevox` may now be references (with pathname/url) on the flite command line and used by the synthesizer

```
./flite -voice cmu_us_awb.flitevox "Hello World"
```

## 8.3 Lexicon Conversion

As of 1.3 the script for converting the CMU lexicon (as distributed as part of Festival) is included. `make_cmulex` will use the version of CMULEX unpacked in the current directory to build a new lexicon. Also in 1.3. a more sophisticated compression technique is used to reduce the lexicon size. The lexicon is pruned, removing those words which the letter to sound rule models get correct. Also the letters and phones are separately huffman coded to produce a smaller lexicon.

## 8.4 Language Conversion

This is by far the weakest part as this is the most open ended. There are basic tools in the `flite/tools/` directory that include Scheme code to convert various Scheme structures to C include CART tree conversion and Lisp list conversion. The other major source of help here is the existing language examples in `flite/lang/usenglish/`.

Adding new language support is far from automatic, but there are core scripts for setting up new Flite support for languages and lexicons. There are also scripts for converting (Festival) phoneset definitions to C and converting Festival lexicons to LTS rules and compressed lexicons in C.

But beyond that you are sort of on your own. The largest gap here is text normalization. We do not yet have a standardize model for text normalization with well defined models for which we could write conversion scripts.

However here is a step by step attempt to show you what to do when building support for a new language/lexicon.

Suppose we need to create support for Pashto, and already have a festival voice running, and want it now to run in flite. Converting the voice itself (unitselection or clustergen) is fairly robust, but you will also need C libraries for `cmu_pashto_lang` and `cmu_pashto_lex`. The first stage is to create the basic template files for these. In the core `flite/` source directory type

```
./tools/make_new_lang_lex pashto
```

This will create language and lex template files in `lang/cmu_pashto_lang/` and `cmu_pashto_lex`.

Then in firectory `lang/cmu_pashto_lang/` type

```
festival $FLITEDIR/tools/make_phoneset.scm
```

```
...
```

```
festival> (phonesettoC "cmu_pashto" (car (load "PATHTO/cmu_pashto_transtac_phoneset
```

This will create `cmu_pashto_lang_phoneset.[ch]`. You must the add these explicitly to the Makefile.

Then in `lang/cmu_pashto_lex/` you have to build the C version of the lexicon and letter to sound rules. The core script is in `flite/tools/build_lex`.

```
mkdir lex
```

```
cd lex
```

```
cp -p $FLITEDIR/tooks/build_lex .
```

Edit `build_lex` to give it the name of your lexicon name, and compiled lexicon from your voice.

```
LEXNAME=cmu_pashto
```

```
LEXFILE=lexicon.out
```

You should (I think) remove the first line “MNCL” from your `lexicon.out` file, note this *must* be the compiled lexicon not the list of entries you compiled from as it expects the ordering, and the syllabification.

```
./build_lex setup
```

Build the letter to sound rules (probably again)

```
./build_lex lts
```

Convert the compiled letter to sound rules to C. This converts the decision trees to decision graphs and runs WFST minimization of them to get a more efficient set of structures. This puts the generated C files in `c/`.

```
./build_lex lts2c
```

Now convert the lexical entries themselves

```
./build_lex lex
```

Again the generate C files will be put in `c/`.

Now we generated a Huffman codes compressed lexicon to reduce the lexicon size, merging frequent letter sequences and phone sequences.

```
./build_lex compresslex
```

The copy the `.c` and `.h` files to `lang/cmu_pashto_lex/` [something about compressed and non-compressed??]



## 9 Porting to new platforms

byte order, unions, compiler restrictions



## 10 Future developments



# Table of Contents

<b>1</b>	<b>Abstract</b>	<b>1</b>
<b>2</b>	<b>Copying</b>	<b>3</b>
<b>3</b>	<b>Acknowledgements</b>	<b>5</b>
<b>4</b>	<b>Installation</b>	<b>7</b>
4.1	Windows Support	7
4.2	Window CE Support	7
4.3	PalmOS Support	8
4.3.1	Some notes on the PalmOS port	10
4.3.2	Using the PalmOS	11
<b>5</b>	<b>Flite Design</b>	<b>13</b>
5.1	Background	13
5.2	Key Decisions	14
<b>6</b>	<b>Structure</b>	<b>17</b>
6.1	cst_val	17
<b>7</b>	<b>APIs</b>	<b>19</b>
7.1	flite binary	19
7.2	Voice selection	19
7.3	C example	20
7.4	Public Functions	21
7.5	Streaming Synthesis	22
<b>8</b>	<b>Converting FestVox Voices</b>	<b>25</b>
8.1	Cocantenative Voice Building	25
8.2	Statistical Voice Building (Clustergen)	28
8.3	Lexicon Conversion	28
8.4	Language Conversion	29
<b>9</b>	<b>Porting to new platforms</b>	<b>31</b>
<b>10</b>	<b>Future developments</b>	<b>33</b>

