espeak-ng

source code

Contents

Chapter 1

Example

As an example, we let the Espeak-ng speech engine pronounce “Salve mundo!” in interlingua (voice: ia):

```
echo "Salve mundo!" | espeak-ng -via -w salve-mundo.wav
```

In Figure 1 we see how the file salve-mundo.wav looks like in Audacity audio editor, with the approximate letter positions added.

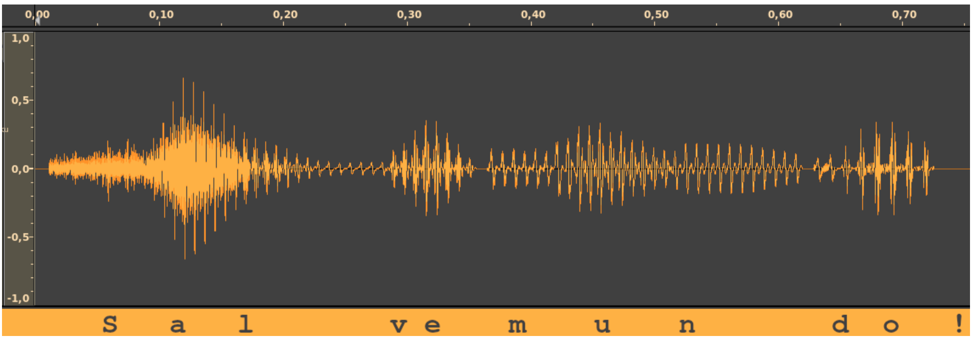

Figure 1. The file salve-mundo.wav in Audacity.

Chapter 2

./src/include/espeak-ng/speak_lib.h

```
#ifndef SPEAK_LIB_H
#define SPEAK_LIB_H

#include <stdio.h>
#include <stddef.h>

#if defined(_WIN32) || defined(_WIN64)
#ifdef LIBESPEAK_NG_EXPORT
#define ESPEAK_API __declspec(dllexport)
#else
#define ESPEAK_API __declspec(dllimport)
#endif
#else
#define ESPEAK_API
#endif

#define ESPEAK_API_REVISION  12

/*
Revision 2
```

```
   Added parameter "options" to eSpeakInitialize()

Revision 3
   Added espeakWORDGAP to  espeak_PARAMETER

Revision 4
   Added flags parameter to espeak_CompileDictionary()

Revision 5
   Added espeakCHARS_16BIT

Revision 6
  Added macros: espeakRATE_MINIMUM, espeakRATE_MAXIMUM,
espeakRATE_NORMAL

Revision 7  24.Dec.2011
  Changed espeak_EVENT structure to add id.string[] for phoneme
mnemonics.
  Added espeakINITIALIZE_PHONEME_IPA option for
espeak_Initialize() to report phonemes as IPA names.

Revision 8  26.Apr.2013
  Added function espeak_TextToPhonemes().

Revision 9  30.May.2013
  Changed function espeak_TextToPhonemes().

Revision 10 29.Aug.2014
  Changed phonememode parameter to espeak_TextToPhonemes() and
espeak_SetPhonemeTrace

Revision 11 (espeak-ng)
  Made ESPEAK_API import/export symbols correctly on Windows.

Revision 12 (espeak-ng)
  Exposed espeak_SetPhonemeCallback. This is available in eSpeak,
but was not exposed in this header.
```

```
*/
         /********************/
         /*  Initialization  */
         /********************/

// values for 'value' in espeak_SetParameter(espeakRATE, value,
0), nominally in words-per-minute
#define espeakRATE_MINIMUM  80
#define espeakRATE_MAXIMUM  450
#define espeakRATE_NORMAL   175

typedef enum {
  espeakEVENT_LIST_TERMINATED = 0, // Retrieval mode: terminates
the event list.
  espeakEVENT_WORD = 1,            // Start of word
  espeakEVENT_SENTENCE = 2,        // Start of sentence
  espeakEVENT_MARK = 3,            // Mark
  espeakEVENT_PLAY = 4,            // Audio element
  espeakEVENT_END = 5,             // End of sentence or clause
  espeakEVENT_MSG_TERMINATED = 6,  // End of message
  espeakEVENT_PHONEME = 7,         // Phoneme, if enabled in
espeak_Initialize()
  espeakEVENT_SAMPLERATE = 8       // internal use, set sample
rate
} espeak_EVENT_TYPE;

typedef struct {
 espeak_EVENT_TYPE type;
 unsigned int unique_identifier; // message identifier (or 0 for
key or character)
 int text_position;    // the number of characters from the start
of the text
 int length;           // word length, in characters (for
espeakEVENT_WORD)
 int audio_position;   // the time in mS within the generated
speech output data
```

```
 int sample;           // sample id (internal use)
 void* user_data;      // pointer supplied by the calling program
 union {
  int number;         // used for WORD and SENTENCE events.
  const char *name;  // used for MARK and PLAY events.  UTF8
string
  char string[8];    // used for phoneme names (UTF8). Terminated
by a zero byte unless the name needs the full 8 bytes.
 } id;
} espeak_EVENT;
```

When a message is supplied to espeak_synth, the request is buffered and espeak_synth returns. When the message is really processed, the callback function will be repetedly called.

In RETRIEVAL mode, the callback function supplies to the calling program the audio data and an event list terminated by 0 (LIST_TERMINATED).

In PLAYBACK mode, the callback function is called as soon as an event happens.

For example suppose that the following message is supplied to espeak_Synth:
"hello, hello."

* Once processed in RETRIEVAL mode, it could lead to 3 calls of the callback function :

** Block 1:
<audio data> +
List of events: SENTENCE + WORD + LIST_TERMINATED

** Block 2:
<audio data> +
List of events: WORD + END + LIST_TERMINATED

```
   ** Block 3:
   no audio data
   List of events: MSG_TERMINATED + LIST_TERMINATED

   * Once processed in PLAYBACK mode, it could lead to 5 calls of
the callback function:

   ** SENTENCE
   ** WORD (call when the sounds are actually played)
   ** WORD
   ** END (call when the end of sentence is actually played.)
   ** MSG_TERMINATED

   The MSG_TERMINATED event is the last event. It can inform the
calling program to clear the user data related to the message.
   So if the synthesis must be stopped, the callback function is
called for each pending message with the MSG_TERMINATED event.

   A MARK event indicates a <mark> element in the text.
   A PLAY event indicates an <audio> element in the text, for
which the calling program should play the named sound file.
*/

typedef enum {
 POS_CHARACTER = 1,
 POS_WORD,
 POS_SENTENCE
} espeak_POSITION_TYPE;

typedef enum {
 /* PLAYBACK mode: plays the audio data, supplies events to the
calling program*/
 AUDIO_OUTPUT_PLAYBACK,

 /* RETRIEVAL mode: supplies audio data and events to the calling
program */
 AUDIO_OUTPUT_RETRIEVAL,
```

```
 /* SYNCHRONOUS mode: as RETRIEVAL but doesn't return until
synthesis is completed */
 AUDIO_OUTPUT_SYNCHRONOUS,

 /* Synchronous playback */
 AUDIO_OUTPUT_SYNCH_PLAYBACK

} espeak_AUDIO_OUTPUT;

typedef enum {
 EE_OK=0,
 EE_INTERNAL_ERROR=-1,
 EE_BUFFER_FULL=1,
 EE_NOT_FOUND=2
} espeak_ERROR;

#define espeakINITIALIZE_PHONEME_EVENTS 0x0001
#define espeakINITIALIZE_PHONEME_IPA   0x0002
#define espeakINITIALIZE_DONT_EXIT     0x8000

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API int espeak_Initialize(espeak_AUDIO_OUTPUT output, int
buflength, const char *path, int options);

   output: the audio data can either be played by eSpeak or
passed back by the SynthCallback function.

   buflength:  The length in mS of sound buffers passed to the
SynthCallback function.
            Value=0 gives a default of 60mS.
            This paramater is only used for
AUDIO_OUTPUT_RETRIEVAL and AUDIO_OUTPUT_SYNCHRONOUS modes.

   path: The directory which contains the espeak-ng-data
```

```
directory, or NULL for the default location.

   options: bit 0:  1=allow espeakEVENT_PHONEME events.
            bit 1:  1= espeakEVENT_PHONEME events give IPA
phoneme names, not eSpeak phoneme names
            bit 15: 1=don't exit if espeak_data is not found
(used for --help)

   Returns: sample rate in Hz, or -1 (EE_INTERNAL_ERROR).
*/

typedef int (t_espeak_callback)(short*, int, espeak_EVENT*);

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API void espeak_SetSynthCallback(t_espeak_callback*
SynthCallback);

   This specifies a function in the calling program which is
called when a buffer of
   speech sound data has been produced.

   The callback function is of the form:

int SynthCallback(short *wav, int numsamples, espeak_EVENT
*events);

   wav:  is the speech sound data which has been produced.
      NULL indicates that the synthesis has been completed.

   numsamples: is the number of entries in wav.  This number may
vary, may be less than
      the value implied by the buflength parameter given in
espeak_Initialize, and may
      sometimes be zero (which does NOT indicate end of
synthesis).
```

```
   events: an array of espeak_EVENT items which indicate word and
sentence events, and
      also the occurance if <mark> and <audio> elements within
the text.  The list of
      events is terminated by an event of type = 0.

   Callback returns: 0=continue synthesis,  1=abort synthesis.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API void espeak_SetUriCallback(int (*UriCallback)(int,
const char*, const char*));

   <audio> tags.  It specifies a callback function which is
called when an <audio> element is
   encountered and allows the calling program to indicate whether
the sound file which
   is specified in the <audio> element is available and is to be
played.

   The callback function is of the form:

int UriCallback(int type, const char *uri, const char *base);

   type:  type of callback event.  Currently only 1= <audio>
element

   uri:   the "src" attribute from the <audio> element

   base:  the "xml:base" attribute (if any) from the <speak>
element

   Return: 1=don't play the sound, but speak the text
alternative.
```

```
           0=place a PLAY event in the event list at the point
where the <audio> element
             occurs.  The calling program can then play the sound
at that point.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API void espeak_SetPhonemeCallback(int
(*PhonemeCallback)(const char *));

         /********************/
         /*    Synthesis     */
         /********************/

#define espeakCHARS_AUTO   0
#define espeakCHARS_UTF8   1
#define espeakCHARS_8BIT   2
#define espeakCHARS_WCHAR  3
#define espeakCHARS_16BIT  4

#define espeakSSML        0x10
#define espeakPHONEMES    0x100
#define espeakENDPAUSE    0x1000
#define espeakKEEP_NAMEDATA 0x2000

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Synth(const void *text,
 size_t size,
 unsigned int position,
 espeak_POSITION_TYPE position_type,
 unsigned int end_position,
 unsigned int flags,
 unsigned int* unique_identifier,
```

```
 void* user_data);

   program in buffers by means of the callback function specified
by espeak_SetSynthCallback(). The command is asynchronous: it is
internally buffered and returns as soon as possible. If
espeak_Initialize was previously called with
AUDIO_OUTPUT_PLAYBACK as argument, the sound data are played by
eSpeak.

   text: The text to be spoken, terminated by a zero character.
It may be either 8-bit characters,
      wide characters (wchar_t), or UTF8 encoding.  Which of
these is determined by the "flags"
      parameter.

   size: Equal to (or greatrer than) the size of the text data,
in bytes.  This is used in order
      to allocate internal storage space for the text.  This
value is not used for
      AUDIO_OUTPUT_SYNCHRONOUS mode.

   position:  The position in the text where speaking starts.
Zero indicates speak from the
      start of the text.

   position_type:  Determines whether "position" is a number of
characters, words, or sentences.
      Values:

   end_position:  If set, this gives a character position at
which speaking will stop.  A value
      of zero indicates no end position.

   flags:  These may be OR'd together:
      Type of character codes, one of:
         espeakCHARS_UTF8     UTF8 encoding
         espeakCHARS_8BIT     The 8 bit ISO-8859 character set
```

```
for the particular language.
         espeakCHARS_AUTO     8 bit or UTF8  (this is the
default)
         espeakCHARS_WCHAR    Wide characters (wchar_t)
         espeakCHARS_16BIT    16 bit characters.

      espeakSSML   Elements within < > are treated as SSML
elements, or if not recognised are ignored.

      espeakPHONEMES  Text within [[ ]] is treated as phonemes
codes (in espeak's Kirshenbaum encoding).

      espeakENDPAUSE  If set then a sentence pause is added at
the end of the text.  If not set then
         this pause is suppressed.

   unique_identifier: This must be either NULL, or point to an
integer variable to
       which eSpeak writes a message identifier number.
       eSpeak includes this number in espeak_EVENT messages which
are the result of
       this call of espeak_Synth().

   user_data: a pointer (or NULL) which will be passed to the
callback function in
       espeak_EVENT messages.

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
```

```
ESPEAK_API espeak_ERROR espeak_Synth_Mark(const void *text,
 size_t size,
 const char *index_mark,
 unsigned int end_position,
 unsigned int flags,
 unsigned int* unique_identifier,
 void* user_data);

   specified by the name of a <mark> element in the text.

   index_mark:  The "name" attribute of a <mark> element within
the text which specified the
      point at which synthesis starts.  UTF8 string.

   For the other parameters, see espeak_Synth()

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Key(const char *key_name);

   If key_name is a single character, it speaks the name of the
character.
   Otherwise, it speaks key_name as a text string.

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
```

```
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Char(wchar_t character);

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

         /***********************/
         /*  Speech Parameters  */
         /***********************/

typedef enum {
  espeakSILENCE=0, /* internal use */
  espeakRATE=1,
  espeakVOLUME=2,
  espeakPITCH=3,
  espeakRANGE=4,
  espeakPUNCTUATION=5,
  espeakCAPITALS=6,
  espeakWORDGAP=7,
  espeakOPTIONS=8,   // reserved for misc. options.  not yet used
  espeakINTONATION=9,

  espeakRESERVED1=10,
  espeakRESERVED2=11,
  espeakEMPHASIS,   /* internal use */
  espeakLINELENGTH, /* internal use */
  espeakVOICETYPE,  // internal, 1=mbrola
  N_SPEECH_PARAM    /* last enum */
} espeak_PARAMETER;
```

```
typedef enum {
  espeakPUNCT_NONE=0,
  espeakPUNCT_ALL=1,
  espeakPUNCT_SOME=2
} espeak_PUNCT_TYPE;

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_SetParameter(espeak_PARAMETER
parameter, int value, int relative);

   relative=0   Sets the absolute value of the parameter.
   relative=1   Sets a relative value of the parameter.

   parameter:
      espeakRATE:    speaking speed in word per minute.  Values
80 to 450.

      espeakVOLUME:  volume in range 0-200 or more.
                     0=silence, 100=normal full volume, greater
values may produce amplitude compression or distortion

      espeakPITCH:   base pitch, range 0-100.  50=normal

      espeakRANGE:   pitch range, range 0-100. 0-monotone,
50=normal

      espeakPUNCTUATION:  which punctuation characters to
announce:
         value in espeak_PUNCT_TYPE (none, all, some),
         see espeak_GetParameter() to specify which characters
are announced.

      espeakCAPITALS: announce capital letters by:
         0=none,
```

```
         1=sound icon,
         2=spelling,
         3 or higher, by raising pitch.  This values gives the
amount in Hz by which the pitch
            of a word raised to indicate it has a capital letter.

      espeakWORDGAP:  pause between words, units of 10mS (at the
default speed)

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API int espeak_GetParameter(espeak_PARAMETER parameter,
int current);

   current=1  Returns the current value of the specified
parameter, as set by SetParameter()
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_SetPunctuationList(const wchar_t
*punctlist);

   value of the Punctuation parameter is set to "some".

   punctlist:  A list of character codes, terminated by a zero
character.
```

```
   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

#define espeakPHONEMES_SHOW    0x01
#define espeakPHONEMES_IPA     0x02
#define espeakPHONEMES_TRACE   0x08
#define espeakPHONEMES_MBROLA  0x10
#define espeakPHONEMES_TIE     0x80

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API void espeak_SetPhonemeTrace(int phonememode, FILE
*stream);

      bits 0-2:
         value=0  No phoneme output (default)
         value=1  Output the translated phoneme symbols for the
text
         value=2  as (1), but produces IPA phoneme names rather
than ascii
      bit 3:   output a trace of how the translation was done
(showing the matching rules and list entries)
      bit 4:   produce pho data for mbrola
      bit 7:   use (bits 8-23) as a tie within multi-letter
phonemes names
      bits 8-23:  separator character, between phoneme names

   stream   output stream for the phoneme symbols (and trace).
If stream=NULL then it uses stdout.
*/

#ifdef __cplusplus
```

```
extern "C"
#endif
ESPEAK_API const char *espeak_TextToPhonemes(const void
**textptr, int textmode, int phonememode);

   It returns a pointer to a character string which contains the
phonemes for the text up to
   end of a sentence, or comma, semicolon, colon, or similar
punctuation.

   textptr: The address of a pointer to the input text which is
terminated by a zero character.
      On return, the pointer has been advanced past the text
which has been translated, or else set
      to NULL to indicate that the end of the text has been
reached.

   textmode: Type of character codes, one of:
         espeakCHARS_UTF8     UTF8 encoding
         espeakCHARS_8BIT     The 8 bit ISO-8859 character set
for the particular language.
         espeakCHARS_AUTO     8 bit or UTF8  (this is the
default)
         espeakCHARS_WCHAR    Wide characters (wchar_t)
         espeakCHARS_16BIT    16 bit characters.

   phoneme_mode
     bit 1:   0=eSpeak's ascii phoneme names, 1= International
Phonetic Alphabet (as UTF-8 characters).
        bit 7:   use (bits 8-23) as a tie within multi-letter
phonemes names
        bits 8-23:  separator character, between phoneme names

*/

#ifdef __cplusplus
extern "C"
```

```
#endif
ESPEAK_API void espeak_CompileDictionary(const char *path, FILE
*log, int flags);

   selected voice.  The required voice should be selected before
calling this function.

   path:  The directory which contains the language's '_rules'
and '_list' files.
          'path' should end with a path separator character
('/').
   log:   Stream for error reports and statistics information. If
log=NULL then stderr will be used.

   flags:  Bit 0: include source line information for debug
purposes (This is displayed with the
          -X command line option).
*/
         /***********************/
         /*   Voice Selection   */
         /***********************/

// voice table
typedef struct {
 const char *name;      // a given name for this voice. UTF8
string.
 const char *languages;       // list of pairs of (byte) priority
+ (string) language (and dialect qualifier)
 const char *identifier;      // the filename for this voice
within espeak-ng-data/voices
 unsigned char gender;  // 0=none 1=male, 2=female,
 unsigned char age;     // 0=not specified, or age in years
 unsigned char variant; // only used when passed as a parameter
to espeak_SetVoiceByProperties
 unsigned char xx1;     // for internal use
 int score;       // for internal use
 void *spare;     // for internal use
```

```
} espeak_VOICE;

  1.  To return the details of the available voices.
  2.  As a parameter to  espeak_SetVoiceByProperties() in order
to specify selection criteria.

   In (1), the "languages" field consists of a list of (UTF8)
language names for which this voice
   may be used, each language name in the list is terminated by a
zero byte and is also preceded by
   a single byte which gives a "priority" number.  The list of
languages is terminated by an
   additional zero byte.

   A language name consists of a language code, optionally
followed by one or more qualifier (dialect)
   names separated by hyphens (eg. "en-uk").  A voice might, for
example, have languages "en-uk" and
   "en".  Even without "en" listed, voice would still be selected
for the "en" language (because
   "en-uk" is related) but at a lower priority.

   The priority byte indicates how the voice is preferred for the
language. A low number indicates a
   more preferred voice, a higher number indicates a less
preferred voice.

   In (2), the "languages" field consists simply of a single
(UTF8) language name, with no preceding
   priority byte.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API const espeak_VOICE **espeak_ListVoices(espeak_VOICE
*voice_spec);
```

```
   The list is terminated by a NULL pointer

   If voice_spec is NULL then all voices are listed.
   If voice spec is given, then only the voices which are
compatible with the voice_spec
   are listed, and they are listed in preference order.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_SetVoiceByFile(const char
*filename);

   "filename" is a UTF8 string.

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_SetVoiceByName(const char *name);

   "name" is a UTF8 string.

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/
```

```
#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_SetVoiceByProperties(espeak_VOICE
*voice_spec);

   fields may be set:

   name     NULL, or a voice name

   languages  NULL, or a single language string (with optional
dialect), eg. "en-uk", or "en"

   gender   0=not specified, 1=male, 2=female

   age      0=not specified, or an age in years

   variant  After a list of candidates is produced, scored and
sorted, "variant" is used to index
            that list and choose a voice.
            variant=0 takes the top voice (i.e. best match).
variant=1 takes the next voice, etc
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_VOICE *espeak_GetCurrentVoice(void);

   This is not affected by temporary voice changes caused by SSML
elements such as <voice> and <s>
*/

#ifdef __cplusplus
extern "C"
#endif
```

```
ESPEAK_API espeak_ERROR espeak_Cancel(void);

   function returns, the audio output is fully stopped and the
synthesizer is ready to
   synthesize a new message.

   Return: EE_OK: operation achieved
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API int espeak_IsPlaying(void);

*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Synchronize(void);

   Return: EE_OK: operation achieved
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Terminate(void);

   Return: EE_OK: operation achieved
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
```

```
#endif
ESPEAK_API const char *espeak_Info(const char **path_data);

   path_data  returns the path to espeak_data
*/
#endif
```

Chapter 3

./src/include/espeak-ng/espeak_ng.h

```
#ifndef ESPEAK_NG_H
#define ESPEAK_NG_H

#include <espeak-ng/speak_lib.h>

#ifdef __cplusplus
extern "C"
{
#endif

#if defined(_WIN32) || defined(_WIN64)
#ifdef LIBESPEAK_NG_EXPORT
#define ESPEAK_NG_API __declspec(dllexport)
#else
#define ESPEAK_NG_API __declspec(dllimport)
#endif
#else
#define ESPEAK_NG_API
#endif
```

```
#define ESPEAKNG_DEFAULT_VOICE "en"

typedef enum {
 ENS_GROUP_MASK               = 0x70000000,
 ENS_GROUP_ERRNO              = 0x00000000, /* Values 0-255 map
to errno error codes. */
 ENS_GROUP_ESPEAK_NG          = 0x10000000, /* eSpeak NG error
codes. */

 /* eSpeak NG 1.49.0 */
 ENS_OK                       = 0,
 ENS_COMPILE_ERROR            = 0x100001FF,
 ENS_VERSION_MISMATCH         = 0x100002FF,
 ENS_FIFO_BUFFER_FULL         = 0x100003FF,
 ENS_NOT_INITIALIZED          = 0x100004FF,
 ENS_AUDIO_ERROR              = 0x100005FF,
 ENS_VOICE_NOT_FOUND          = 0x100006FF,
 ENS_MBROLA_NOT_FOUND         = 0x100007FF,
 ENS_MBROLA_VOICE_NOT_FOUND   = 0x100008FF,
 ENS_EVENT_BUFFER_FULL        = 0x100009FF,
 ENS_NOT_SUPPORTED            = 0x10000AFF,
 ENS_UNSUPPORTED_PHON_FORMAT  = 0x10000BFF,
 ENS_NO_SPECT_FRAMES          = 0x10000CFF,
 ENS_EMPTY_PHONEME_MANIFEST   = 0x10000DFF,
 ENS_SPEECH_STOPPED           = 0x10000EFF,

 /* eSpeak NG 1.49.2 */
 ENS_UNKNOWN_PHONEME_FEATURE  = 0x10000FFF,
 ENS_UNKNOWN_TEXT_ENCODING    = 0x100010FF,
} espeak_ng_STATUS;

typedef enum {
 ENOUTPUT_MODE_SYNCHRONOUS = 0x0001,
 ENOUTPUT_MODE_SPEAK_AUDIO = 0x0002,
} espeak_ng_OUTPUT_MODE;

typedef enum {
```

```
 ENGENDER_UNKNOWN = 0,
 ENGENDER_MALE = 1,
 ENGENDER_FEMALE = 2,
 ENGENDER_NEUTRAL = 3,
} espeak_ng_VOICE_GENDER;

typedef struct espeak_ng_ERROR_CONTEXT_ *espeak_ng_ERROR_CONTEXT;

ESPEAK_NG_API void
espeak_ng_ClearErrorContext(espeak_ng_ERROR_CONTEXT *context);

ESPEAK_NG_API void
espeak_ng_GetStatusCodeMessage(espeak_ng_STATUS status,
                               char *buffer,
                               size_t length);

ESPEAK_NG_API void
espeak_ng_PrintStatusCodeMessage(espeak_ng_STATUS status,
                                 FILE *out,
                                 espeak_ng_ERROR_CONTEXT
context);

ESPEAK_NG_API void
espeak_ng_InitializePath(const char *path);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_Initialize(espeak_ng_ERROR_CONTEXT *context);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_InitializeOutput(espeak_ng_OUTPUT_MODE output_mode,
                           int buffer_length,
                           const char *device);

ESPEAK_NG_API int
espeak_ng_GetSampleRate(void);

ESPEAK_NG_API espeak_ng_STATUS
```

```
espeak_ng_SetParameter(espeak_PARAMETER parameter,
                       int value,
                       int relative);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_SetPunctuationList(const wchar_t *punctlist);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_SetVoiceByName(const char *name);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_SetVoiceByFile(const char *filename);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_SetVoiceByProperties(espeak_VOICE *voice_selector);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_Synthesize(const void *text,
                     size_t size,
                     unsigned int position,
                     espeak_POSITION_TYPE position_type,
                     unsigned int end_position,
                     unsigned int flags,
                     unsigned int *unique_identifier,
                     void *user_data);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_SynthesizeMark(const void *text,
                         size_t size,
                         const char *index_mark,
                         unsigned int end_position,
                         unsigned int flags,
                         unsigned int *unique_identifier,
                         void *user_data);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_SpeakKeyName(const char *key_name);
```

```
ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_SpeakCharacter(wchar_t character);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_Cancel(void);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_Synchronize(void);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_Terminate(void);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_CompileDictionary(const char *dsource,
                            const char *dict_name,
                            FILE *log,
                            int flags,
                            espeak_ng_ERROR_CONTEXT *context);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_CompileMbrolaVoice(const char *path,
                             FILE *log,
                             espeak_ng_ERROR_CONTEXT *context);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_CompilePhonemeData(long rate,
                             FILE *log,
                             espeak_ng_ERROR_CONTEXT *context);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_CompileIntonation(FILE *log,
                            espeak_ng_ERROR_CONTEXT *context);

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_CompilePhonemeDataPath(long rate,
                                 const char *source_path,
```

```
                                  const char *destination_path,
                                  FILE *log,
                                  espeak_ng_ERROR_CONTEXT
*context);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 4

./src/include/espeak/speak_lib.h

```
#ifndef SPEAK_LIB_H
#define SPEAK_LIB_H

#include <stdio.h>
#include <stddef.h>

#if defined(_WIN32) || defined(_WIN64)
#ifdef LIBESPEAK_NG_EXPORT
#define ESPEAK_API __declspec(dllexport)
#else
#define ESPEAK_API __declspec(dllimport)
#endif
#else
#define ESPEAK_API
#endif

#define ESPEAK_API_REVISION  12

Revision 2
   Added parameter "options" to eSpeakInitialize()

Revision 3
```

```
   Added espeakWORDGAP to  espeak_PARAMETER

Revision 4
   Added flags parameter to espeak_CompileDictionary()

Revision 5
   Added espeakCHARS_16BIT

Revision 6
  Added macros: espeakRATE_MINIMUM, espeakRATE_MAXIMUM,
espeakRATE_NORMAL

Revision 7  24.Dec.2011
  Changed espeak_EVENT structure to add id.string[] for phoneme
mnemonics.
  Added espeakINITIALIZE_PHONEME_IPA option for
espeak_Initialize() to report phonemes as IPA names.

Revision 8  26.Apr.2013
  Added function espeak_TextToPhonemes().

Revision 9  30.May.2013
  Changed function espeak_TextToPhonemes().

Revision 10 29.Aug.2014
  Changed phonememode parameter to espeak_TextToPhonemes() and
espeak_SetPhonemeTrace

Revision 11 (espeak-ng)
  Made ESPEAK_API import/export symbols correctly on Windows.

Revision 12 (espeak-ng)
  Exposed espeak_SetPhonemeCallback. This is available in eSpeak,
but was not exposed in this header.

*/
         /********************/
```

```
        /*  Initialization  */
        /********************/

// values for 'value' in espeak_SetParameter(espeakRATE, value,
0), nominally in words-per-minute
#define espeakRATE_MINIMUM  80
#define espeakRATE_MAXIMUM  450
#define espeakRATE_NORMAL   175

typedef enum {
  espeakEVENT_LIST_TERMINATED = 0, // Retrieval mode: terminates
the event list.
  espeakEVENT_WORD = 1,            // Start of word
  espeakEVENT_SENTENCE = 2,        // Start of sentence
  espeakEVENT_MARK = 3,            // Mark
  espeakEVENT_PLAY = 4,            // Audio element
  espeakEVENT_END = 5,             // End of sentence or clause
  espeakEVENT_MSG_TERMINATED = 6,  // End of message
  espeakEVENT_PHONEME = 7,         // Phoneme, if enabled in
espeak_Initialize()
  espeakEVENT_SAMPLERATE = 8       // internal use, set sample
rate
} espeak_EVENT_TYPE;

typedef struct {
 espeak_EVENT_TYPE type;
 unsigned int unique_identifier; // message identifier (or 0 for
key or character)
 int text_position;    // the number of characters from the start
of the text
 int length;           // word length, in characters (for
espeakEVENT_WORD)
 int audio_position;   // the time in mS within the generated
speech output data
 int sample;           // sample id (internal use)
 void* user_data;      // pointer supplied by the calling program
 union {
```

```
  int number;        // used for WORD and SENTENCE events.
  const char *name;  // used for MARK and PLAY events.  UTF8
string
  char string[8];    // used for phoneme names (UTF8). Terminated
by a zero byte unless the name needs the full 8 bytes.
 } id;
} espeak_EVENT;
```

When a message is supplied to espeak_synth, the request is buffered and espeak_synth returns. When the message is really processed, the callback function will be repetedly called.

In RETRIEVAL mode, the callback function supplies to the calling program the audio data and an event list terminated by 0 (LIST_TERMINATED).

In PLAYBACK mode, the callback function is called as soon as an event happens.

For example suppose that the following message is supplied to espeak_Synth:
"hello, hello."

* Once processed in RETRIEVAL mode, it could lead to 3 calls of the callback function :

** Block 1:
<audio data> +
List of events: SENTENCE + WORD + LIST_TERMINATED

** Block 2:
<audio data> +
List of events: WORD + END + LIST_TERMINATED

** Block 3:
no audio data
List of events: MSG_TERMINATED + LIST_TERMINATED

```
   * Once processed in PLAYBACK mode, it could lead to 5 calls of
the callback function:

   ** SENTENCE
   ** WORD (call when the sounds are actually played)
   ** WORD
   ** END (call when the end of sentence is actually played.)
   ** MSG_TERMINATED

   The MSG_TERMINATED event is the last event. It can inform the
calling program to clear the user data related to the message.
   So if the synthesis must be stopped, the callback function is
called for each pending message with the MSG_TERMINATED event.

   A MARK event indicates a <mark> element in the text.
   A PLAY event indicates an <audio> element in the text, for
which the calling program should play the named sound file.
*/

typedef enum {
 POS_CHARACTER = 1,
 POS_WORD,
 POS_SENTENCE
} espeak_POSITION_TYPE;

typedef enum {
 /* PLAYBACK mode: plays the audio data, supplies events to the
calling program*/
 AUDIO_OUTPUT_PLAYBACK,

 /* RETRIEVAL mode: supplies audio data and events to the calling
program */
 AUDIO_OUTPUT_RETRIEVAL,

 /* SYNCHRONOUS mode: as RETRIEVAL but doesn't return until
synthesis is completed */
```

```
 AUDIO_OUTPUT_SYNCHRONOUS,

 /* Synchronous playback */
 AUDIO_OUTPUT_SYNCH_PLAYBACK

} espeak_AUDIO_OUTPUT;

typedef enum {
 EE_OK=0,
 EE_INTERNAL_ERROR=-1,
 EE_BUFFER_FULL=1,
 EE_NOT_FOUND=2
} espeak_ERROR;

#define espeakINITIALIZE_PHONEME_EVENTS 0x0001
#define espeakINITIALIZE_PHONEME_IPA   0x0002
#define espeakINITIALIZE_DONT_EXIT     0x8000

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API int espeak_Initialize(espeak_AUDIO_OUTPUT output, int
buflength, const char *path, int options);

   output: the audio data can either be played by eSpeak or
passed back by the SynthCallback function.

   buflength:  The length in mS of sound buffers passed to the
SynthCallback function.
            Value=0 gives a default of 60mS.
            This paramater is only used for
AUDIO_OUTPUT_RETRIEVAL and AUDIO_OUTPUT_SYNCHRONOUS modes.

   path: The directory which contains the espeak-ng-data
directory, or NULL for the default location.

   options: bit 0:  1=allow espeakEVENT_PHONEME events.
```

```
            bit 1:  1= espeakEVENT_PHONEME events give IPA
phoneme names, not eSpeak phoneme names
            bit 15: 1=don't exit if espeak_data is not found
(used for --help)

   Returns: sample rate in Hz, or -1 (EE_INTERNAL_ERROR).
*/

typedef int (t_espeak_callback)(short*, int, espeak_EVENT*);

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API void espeak_SetSynthCallback(t_espeak_callback*
SynthCallback);

   This specifies a function in the calling program which is
called when a buffer of
   speech sound data has been produced.

   The callback function is of the form:

int SynthCallback(short *wav, int numsamples, espeak_EVENT
*events);

   wav:  is the speech sound data which has been produced.
      NULL indicates that the synthesis has been completed.

   numsamples: is the number of entries in wav.  This number may
vary, may be less than
      the value implied by the buflength parameter given in
espeak_Initialize, and may
      sometimes be zero (which does NOT indicate end of
synthesis).

   events: an array of espeak_EVENT items which indicate word and
sentence events, and
```

```
      also the occurance if <mark> and <audio> elements within
the text.  The list of
      events is terminated by an event of type = 0.

   Callback returns: 0=continue synthesis,  1=abort synthesis.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API void espeak_SetUriCallback(int (*UriCallback)(int,
const char*, const char*));

   <audio> tags.  It specifies a callback function which is
called when an <audio> element is
   encountered and allows the calling program to indicate whether
the sound file which
   is specified in the <audio> element is available and is to be
played.

   The callback function is of the form:

int UriCallback(int type, const char *uri, const char *base);

   type:  type of callback event.  Currently only 1= <audio>
element

   uri:   the "src" attribute from the <audio> element

   base:  the "xml:base" attribute (if any) from the <speak>
element

   Return: 1=don't play the sound, but speak the text
alternative.
           0=place a PLAY event in the event list at the point
where the <audio> element
             occurs.  The calling program can then play the sound
```

```
at that point.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API void espeak_SetPhonemeCallback(int
(*PhonemeCallback)(const char *));

         /********************/
         /*    Synthesis     */
         /********************/

#define espeakCHARS_AUTO   0
#define espeakCHARS_UTF8   1
#define espeakCHARS_8BIT   2
#define espeakCHARS_WCHAR  3
#define espeakCHARS_16BIT  4

#define espeakSSML        0x10
#define espeakPHONEMES    0x100
#define espeakENDPAUSE    0x1000
#define espeakKEEP_NAMEDATA 0x2000

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Synth(const void *text,
 size_t size,
 unsigned int position,
 espeak_POSITION_TYPE position_type,
 unsigned int end_position,
 unsigned int flags,
 unsigned int* unique_identifier,
 void* user_data);

   program in buffers by means of the callback function specified
```

```
by espeak_SetSynthCallback(). The command is asynchronous: it is
internally buffered and returns as soon as possible. If
espeak_Initialize was previously called with
AUDIO_OUTPUT_PLAYBACK as argument, the sound data are played by
eSpeak.

   text: The text to be spoken, terminated by a zero character.
It may be either 8-bit characters,
      wide characters (wchar_t), or UTF8 encoding.  Which of
these is determined by the "flags"
      parameter.

   size: Equal to (or greatrer than) the size of the text data,
in bytes.  This is used in order
      to allocate internal storage space for the text.  This
value is not used for
      AUDIO_OUTPUT_SYNCHRONOUS mode.

   position:  The position in the text where speaking starts.
Zero indicates speak from the
      start of the text.

   position_type:  Determines whether "position" is a number of
characters, words, or sentences.
      Values:

   end_position:  If set, this gives a character position at
which speaking will stop.  A value
      of zero indicates no end position.

   flags:  These may be OR'd together:
      Type of character codes, one of:
         espeakCHARS_UTF8     UTF8 encoding
         espeakCHARS_8BIT     The 8 bit ISO-8859 character set
for the particular language.
         espeakCHARS_AUTO     8 bit or UTF8  (this is the
default)
```

```
         espeakCHARS_WCHAR    Wide characters (wchar_t)
         espeakCHARS_16BIT    16 bit characters.

      espeakSSML   Elements within < > are treated as SSML
elements, or if not recognised are ignored.

      espeakPHONEMES  Text within [[ ]] is treated as phonemes
codes (in espeak's Kirshenbaum encoding).

      espeakENDPAUSE  If set then a sentence pause is added at
the end of the text.  If not set then
         this pause is suppressed.

   unique_identifier: This must be either NULL, or point to an
integer variable to
       which eSpeak writes a message identifier number.
       eSpeak includes this number in espeak_EVENT messages which
are the result of
       this call of espeak_Synth().

   user_data: a pointer (or NULL) which will be passed to the
callback function in
       espeak_EVENT messages.

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Synth_Mark(const void *text,
 size_t size,
 const char *index_mark,
```

```
 unsigned int end_position,
 unsigned int flags,
 unsigned int* unique_identifier,
 void* user_data);

   specified by the name of a <mark> element in the text.

   index_mark:  The "name" attribute of a <mark> element within
the text which specified the
      point at which synthesis starts.  UTF8 string.

   For the other parameters, see espeak_Synth()

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Key(const char *key_name);

   If key_name is a single character, it speaks the name of the
character.
   Otherwise, it speaks key_name as a text string.

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
```

```
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Char(wchar_t character);

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

         /***********************/
         /*  Speech Parameters  */
         /***********************/

typedef enum {
  espeakSILENCE=0, /* internal use */
  espeakRATE=1,
  espeakVOLUME=2,
  espeakPITCH=3,
  espeakRANGE=4,
  espeakPUNCTUATION=5,
  espeakCAPITALS=6,
  espeakWORDGAP=7,
  espeakOPTIONS=8,   // reserved for misc. options.  not yet used
  espeakINTONATION=9,

  espeakRESERVED1=10,
  espeakRESERVED2=11,
  espeakEMPHASIS,   /* internal use */
  espeakLINELENGTH, /* internal use */
  espeakVOICETYPE,  // internal, 1=mbrola
  N_SPEECH_PARAM    /* last enum */
} espeak_PARAMETER;

typedef enum {
  espeakPUNCT_NONE=0,
```

```
  espeakPUNCT_ALL=1,
  espeakPUNCT_SOME=2
} espeak_PUNCT_TYPE;

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_SetParameter(espeak_PARAMETER
parameter, int value, int relative);

   relative=0   Sets the absolute value of the parameter.
   relative=1   Sets a relative value of the parameter.

   parameter:
      espeakRATE:    speaking speed in word per minute.  Values
80 to 450.

      espeakVOLUME:  volume in range 0-200 or more.
                     0=silence, 100=normal full volume, greater
values may produce amplitude compression or distortion

      espeakPITCH:   base pitch, range 0-100.  50=normal

      espeakRANGE:   pitch range, range 0-100. 0-monotone,
50=normal

      espeakPUNCTUATION:  which punctuation characters to
announce:
         value in espeak_PUNCT_TYPE (none, all, some),
         see espeak_GetParameter() to specify which characters
are announced.

      espeakCAPITALS: announce capital letters by:
         0=none,
         1=sound icon,
         2=spelling,
         3 or higher, by raising pitch.  This values gives the
```

```
amount in Hz by which the pitch
            of a word raised to indicate it has a capital letter.

      espeakWORDGAP:  pause between words, units of 10mS (at the
default speed)

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API int espeak_GetParameter(espeak_PARAMETER parameter,
int current);

   current=1  Returns the current value of the specified
parameter, as set by SetParameter()
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_SetPunctuationList(const wchar_t
*punctlist);

   value of the Punctuation parameter is set to "some".

   punctlist:  A list of character codes, terminated by a zero
character.

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
```

```
again.
    EE_INTERNAL_ERROR.
*/

#define espeakPHONEMES_SHOW    0x01
#define espeakPHONEMES_IPA     0x02
#define espeakPHONEMES_TRACE   0x08
#define espeakPHONEMES_MBROLA  0x10
#define espeakPHONEMES_TIE     0x80

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API void espeak_SetPhonemeTrace(int phonememode, FILE
*stream);

      bits 0-2:
         value=0  No phoneme output (default)
         value=1  Output the translated phoneme symbols for the
text
         value=2  as (1), but produces IPA phoneme names rather
than ascii
      bit 3:   output a trace of how the translation was done
(showing the matching rules and list entries)
      bit 4:   produce pho data for mbrola
      bit 7:   use (bits 8-23) as a tie within multi-letter
phonemes names
      bits 8-23:  separator character, between phoneme names

   stream   output stream for the phoneme symbols (and trace).
If stream=NULL then it uses stdout.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API const char *espeak_TextToPhonemes(const void
```

```
**textptr, int textmode, int phonememode);

   It returns a pointer to a character string which contains the
phonemes for the text up to
   end of a sentence, or comma, semicolon, colon, or similar
punctuation.

   textptr: The address of a pointer to the input text which is
terminated by a zero character.
      On return, the pointer has been advanced past the text
which has been translated, or else set
      to NULL to indicate that the end of the text has been
reached.

   textmode: Type of character codes, one of:
         espeakCHARS_UTF8     UTF8 encoding
         espeakCHARS_8BIT     The 8 bit ISO-8859 character set
for the particular language.
         espeakCHARS_AUTO     8 bit or UTF8  (this is the
default)
         espeakCHARS_WCHAR    Wide characters (wchar_t)
         espeakCHARS_16BIT    16 bit characters.

   phoneme_mode
     bit 1:   0=eSpeak's ascii phoneme names, 1= International
Phonetic Alphabet (as UTF-8 characters).
        bit 7:   use (bits 8-23) as a tie within multi-letter
phonemes names
        bits 8-23:  separator character, between phoneme names

*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API void espeak_CompileDictionary(const char *path, FILE
*log, int flags);
```

```
   selected voice.  The required voice should be selected before
calling this function.

   path:  The directory which contains the language's '_rules'
and '_list' files.
          'path' should end with a path separator character
('/').
   log:   Stream for error reports and statistics information. If
log=NULL then stderr will be used.

   flags:  Bit 0: include source line information for debug
purposes (This is displayed with the
          -X command line option).
*/
         /***********************/
         /*   Voice Selection   */
         /***********************/

// voice table
typedef struct {
 const char *name;      // a given name for this voice. UTF8
string.
 const char *languages;       // list of pairs of (byte) priority
+ (string) language (and dialect qualifier)
 const char *identifier;      // the filename for this voice
within espeak-ng-data/voices
 unsigned char gender;  // 0=none 1=male, 2=female,
 unsigned char age;     // 0=not specified, or age in years
 unsigned char variant; // only used when passed as a parameter
to espeak_SetVoiceByProperties
 unsigned char xx1;     // for internal use
 int score;       // for internal use
 void *spare;     // for internal use
} espeak_VOICE;
```

  1. To return the details of the available voices.

```
  2.  As a parameter to  espeak_SetVoiceByProperties() in order
to specify selection criteria.

   In (1), the "languages" field consists of a list of (UTF8)
language names for which this voice
   may be used, each language name in the list is terminated by a
zero byte and is also preceded by
   a single byte which gives a "priority" number.  The list of
languages is terminated by an
   additional zero byte.

   A language name consists of a language code, optionally
followed by one or more qualifier (dialect)
   names separated by hyphens (eg. "en-uk").  A voice might, for
example, have languages "en-uk" and
   "en".  Even without "en" listed, voice would still be selected
for the "en" language (because
   "en-uk" is related) but at a lower priority.

   The priority byte indicates how the voice is preferred for the
language. A low number indicates a
   more preferred voice, a higher number indicates a less
preferred voice.

   In (2), the "languages" field consists simply of a single
(UTF8) language name, with no preceding
   priority byte.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API const espeak_VOICE **espeak_ListVoices(espeak_VOICE
*voice_spec);

   The list is terminated by a NULL pointer
```

```
   If voice_spec is NULL then all voices are listed.
   If voice spec is given, then only the voices which are
compatible with the voice_spec
   are listed, and they are listed in preference order.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_SetVoiceByFile(const char
*filename);

   "filename" is a UTF8 string.

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_SetVoiceByName(const char *name);

   "name" is a UTF8 string.

   Return: EE_OK: operation achieved
           EE_BUFFER_FULL: the command can not be buffered;
             you may try after a while to call the function
again.
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
```

```
#endif
ESPEAK_API espeak_ERROR espeak_SetVoiceByProperties(espeak_VOICE
*voice_spec);

   fields may be set:

   name      NULL, or a voice name

   languages  NULL, or a single language string (with optional
dialect), eg. "en-uk", or "en"

   gender    0=not specified, 1=male, 2=female

   age       0=not specified, or an age in years

   variant  After a list of candidates is produced, scored and
sorted, "variant" is used to index
            that list and choose a voice.
            variant=0 takes the top voice (i.e. best match).
variant=1 takes the next voice, etc
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_VOICE *espeak_GetCurrentVoice(void);

   This is not affected by temporary voice changes caused by SSML
elements such as <voice> and <s>
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Cancel(void);

   function returns, the audio output is fully stopped and the
```

```
synthesizer is ready to
   synthesize a new message.

   Return: EE_OK: operation achieved
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API int espeak_IsPlaying(void);

*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Synchronize(void);

   Return: EE_OK: operation achieved
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API espeak_ERROR espeak_Terminate(void);

   Return: EE_OK: operation achieved
    EE_INTERNAL_ERROR.
*/

#ifdef __cplusplus
extern "C"
#endif
ESPEAK_API const char *espeak_Info(const char **path_data);
```

```
   path_data  returns the path to espeak_data
*/
#endif
```

Chapter 5

./src/libespeak-ng/setlengths.h

```
#ifndef ESPEAK_NG_SETLENGTHS_H
#define ESPEAK_NG_SETLENGTHS_H

#include "translate.h"

#ifdef __cplusplus
extern "C"
{
#endif

void CalcLengths(Translator *tr);

espeak_ng_STATUS SetParameter(int parameter,
 int value,
 int relative
 );

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 6

./src/libespeak-ng/error.h

```
#ifndef ESPEAK_NG_ERROR_API
#define ESPEAK_NG_ERROR_API

#include <espeak-ng/espeak_ng.h>

#ifdef __cplusplus
extern "C"
{
#endif

typedef enum
{
 ERROR_CONTEXT_FILE,
 ERROR_CONTEXT_VERSION,
} espeak_ng_CONTEXT_TYPE;

typedef struct espeak_ng_ERROR_CONTEXT_
{
 espeak_ng_CONTEXT_TYPE type;
 char *name;
 int version;
 int expected_version;
```

```
} espeak_ng_ERROR_CONTEXT_;

espeak_ng_STATUS
create_file_error_context(espeak_ng_ERROR_CONTEXT *context,
                          espeak_ng_STATUS status,
                          const char *filename);

espeak_ng_STATUS
create_version_mismatch_error_context(espeak_ng_ERROR_CONTEXT
*context,
                                      const char *path,
                                      int version,
                                      int expected_version);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 7

./src/libespeak-ng/intonation.h

```
#ifndef ESPEAK_NG_INTONATION_H
#define ESPEAK_NG_INTONATION_H

#include "translate.h"

#ifdef __cplusplus
extern "C"
{
#endif

void CalcPitches(Translator *tr, int clause_type);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 8

./src/libespeak-ng/readclause.h

```
#ifndef ESPEAK_NG_READCLAUSE_H
#define ESPEAK_NG_READCLAUSE_H

#include "translate.h"

#ifdef __cplusplus
extern "C"
{
#endif

typedef struct {
 int type;
 int parameter[N_SPEECH_PARAM];
} PARAM_STACK;

extern PARAM_STACK param_stack[];

// Tests if all bytes of str up to size are null
int is_str_totally_null(const char* str, int size);

int clause_type_from_codepoint(uint32_t c);
int towlower2(unsigned int c, Translator *translator); //
```

```
Supports Turkish I
int Eof(void);
const char *WordToString2(unsigned int word);
int Read4Bytes(FILE *f);
int LoadSoundFile2(const char *fname);
int AddNameData(const char *name,
                int wide);
int ReadClause(Translator *tr,
  char *buf,
  short *charix,
  int *charix_top,
  int n_buf,
  int *tone_type,
  char *voice_change);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 9

./src/libespeak-ng/dictionary.h

```
#ifndef ESPEAK_NG_DICTIONARY_H
#define ESPEAK_NG_DICTIONARY_H

#include "compiledict.h"
#include "synthesize.h"
#include "translate.h"

#ifdef __cplusplus
extern "C"
{
#endif

extern ESPEAK_NG_API void strncpy0(char *to, const char *from,
int size);
int LoadDictionary(Translator *tr, const char *name, int
no_error);
int HashDictionary(const char *string);
const char *EncodePhonemes(const char *p, char *outptr, int
*bad_phoneme);
void DecodePhonemes(const char *inptr, char *outptr);
char *WritePhMnemonic(char *phon_out, PHONEME_TAB *ph,
PHONEME_LIST *plist, int use_ipa, int *flags);
```

```
const char *GetTranslatedPhonemeString(int phoneme_mode);
int IsVowel(Translator *tr, int letter);
int Unpronouncable(Translator *tr, char *word, int posn);
void ChangeWordStress(Translator *tr, char *word, int
new_stress);
void SetWordStress(Translator *tr, char *output, unsigned int
*dictionary_flags, int tonic, int control);
void AppendPhonemes(Translator *tr, char *string, int size, const
char *ph);
int TranslateRules(Translator *tr, char *p_start, char *phonemes,
int ph_size, char *end_phonemes, int word_flags, unsigned int
*dict_flags);
int TransposeAlphabet(Translator *tr, char *text);
int Lookup(Translator *tr, const char *word, char *ph_out);
int LookupDictList(Translator *tr, char **wordptr, char *ph_out,
unsigned int *flags, int end_flags, WORD_TAB *wtab);
int LookupFlags(Translator *tr, const char *word, unsigned int
**flags_out);
int RemoveEnding(Translator *tr, char *word, int end_type, char
*word_copy);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 10

./src/libespeak-ng/ssml.h

```
#ifndef ESPEAK_NG_SSML_API
#define ESPEAK_NG_SSML_API

#include <stdbool.h>
#include <wchar.h>

#include <espeak-ng/speak_lib.h>

#ifdef __cplusplus
extern "C"
{
#endif

// stack for language and voice properties
// frame 0 is for the defaults, before any ssml tags.
typedef struct {
        int tag_type;
        int voice_variant_number;
        int voice_gender;
        int voice_age;
        char voice_name[40];
        char language[20];
```

```
} SSML_STACK;

#define N_PARAM_STACK  20

#define SSML_SPEAK        1
#define SSML_VOICE        2
#define SSML_PROSODY      3
#define SSML_SAYAS        4
#define SSML_MARK         5
#define SSML_SENTENCE     6
#define SSML_PARAGRAPH    7
#define SSML_PHONEME      8
#define SSML_SUB          9
#define SSML_STYLE       10
#define SSML_AUDIO       11
#define SSML_EMPHASIS    12
#define SSML_BREAK       13
#define SSML_IGNORE_TEXT 14
#define HTML_BREAK       15
#define HTML_NOSPACE     16   // don't insert a space for this
element, so it doesn't break a word
#define SSML_CLOSE       0x20 // for a closing tag, OR this with
the tag type

int ProcessSsmlTag(wchar_t *xml_buf,
                   char *outbuf,
                   int *outix,
                   int n_outbuf,
                   bool self_closing,
                   const char *xmlbase,
                   bool *audio_text,
                   char *current_voice_id,
                   espeak_VOICE *base_voice,
                   char *base_voice_variant_name,
                   bool *ignore_text,
                   bool *clear_skipping_text,
                   int *sayas_mode,
```

```
                   int *sayas_start,
                   SSML_STACK *ssml_stack,
                   int *n_ssml_stack,
                   int *n_param_stack,
                   int *speech_parameters);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 11

./src/libespeak-ng/translate.h

```
#ifndef ESPEAK_NG_TRANSLATE_H
#define ESPEAK_NG_TRANSLATE_H

#include <stdbool.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/encoding.h>

#ifdef __cplusplus
extern "C"
{
#endif

#define L(c1, c2) (c1<<8)+c2 // combine two characters into an
integer for translator name
#define L3(c1, c2, c3) (c1<<16)+(c2<<8) + c3 // combine three
characters into an integer for translator name

#define CTRL_EMBEDDED    0x01 // control character at the start
of an embedded command
#define REPLACED_E       'E' // 'e' replaced by silent e
```

```
#define N_WORD_PHONEMES  200 // max phonemes in a word
#define N_WORD_BYTES     160 // max bytes for the UTF8 characters
in a word
#define N_CLAUSE_WORDS   300 // max words in a clause
#define N_TR_SOURCE      800 // the source text of a single
clause (UTF8 bytes)

#define N_RULE_GROUP2    120 // max num of two-letter rule chains
#define N_HASH_DICT     1024
#define N_LETTER_GROUPS   95 // maximum is 127-32

// dictionary flags, word 1
// bits 0-3  stressed syllable,  bit 6=unstressed
#define FLAG_SKIPWORDS        0x80
#define FLAG_PREPAUSE        0x100

#define FLAG_STRESS_END      0x200 // full stress if at end of
clause
#define FLAG_STRESS_END2     0x400 // full stress if at end of
clause, or only followed by unstressed
#define FLAG_UNSTRESS_END    0x800 // reduce stress at end of
clause
#define FLAG_SPELLWORD      0x1000 // re-translate the word as
individual letters, separated by spaces
#define FLAG_ACCENT_BEFORE  0x1000 // say this accent name before
the letter name
#define FLAG_ABBREV         0x2000 // spell as letters, even with
a vowel, OR use specified pronunciation rather than split into
letters
#define FLAG_DOUBLING       0x4000 // doubles the following
consonant

#define BITNUM_FLAG_ALT         14 // bit number of
FLAG_ALT_TRANS - 1
#define FLAG_ALT_TRANS      0x8000 // language specific
#define FLAG_ALT2_TRANS    0x10000 // language specific
#define FLAG_ALT3_TRANS    0x20000 // language specific
```

```
#define FLAG_ALT4_TRANS    0x40000 // language specific
#define FLAG_ALT5_TRANS    0x80000 // language specific
#define FLAG_ALT6_TRANS   0x100000 // language specific
#define FLAG_ALT7_TRANS   0x200000 // language specific

#define FLAG_COMBINE      0x800000 // combine with the next word
#define FLAG_ALLOW_DOT  0x01000000 // ignore '.' after word
(abbreviation)
#define FLAG_NEEDS_DOT  0x02000000 // only if the word is
followed by a dot
#define FLAG_WAS_UNPRONOUNCABLE  0x04000000  // the
unpronounceable routine was used
#define FLAG_MAX3       0x08000000 // limit to 3 repeats
#define FLAG_PAUSE1     0x10000000 // shorter prepause
#define FLAG_TEXTMODE   0x20000000 // word translates to
replacement text, not phonemes
#define BITNUM_FLAG_TEXTMODE    29

#define FLAG_FOUND_ATTRIBUTES 0x40000000 // word was found in the
dictionary list (has attributes)
#define FLAG_FOUND            0x80000000 // pronunciation was
found in the dictionary list

// dictionary flags, word 2
#define FLAG_VERBF              0x1 // verb follows
#define FLAG_VERBSF             0x2 // verb follows, may have -s
suffix
#define FLAG_NOUNF              0x4 // noun follows
#define FLAG_PASTF              0x8 // past tense follows
#define FLAG_VERB              0x10 // pronunciation for verb
#define FLAG_NOUN              0x20 // pronunciation for noun
#define FLAG_PAST              0x40 // pronunciation for past
tense
#define FLAG_VERB_EXT         0x100 // extend the 'verb follows'
#define FLAG_CAPITAL          0x200 // pronunciation if initial
letter is upper case
#define FLAG_ALLCAPS          0x400 // only if the word is all
```

```
capitals
#define FLAG_ACCENT          0x800 // character name is base-
character name + accent name
#define FLAG_HYPHENATED     0x1000 // multiple-words, but needs
hyphen between parts 1 and 2
#define FLAG_SENTENCE       0x2000 // only if the clause is a
sentence
#define FLAG_ONLY           0x4000
#define FLAG_ONLY_S         0x8000
#define FLAG_STEM          0x10000 // must have a suffix
#define FLAG_ATEND         0x20000 // use this pronunciation if
at end of clause
#define FLAG_ATSTART       0x40000 // use this pronunciation if
at start of clause
#define FLAG_NATIVE        0x80000 // not if we've switched
translators
#define FLAG_LOOKUP_SYMBOL 0x40000000 // to indicate called from
Lookup()

#define BITNUM_FLAG_ALLCAPS    0x2a
#define BITNUM_FLAG_HYPHENATED 0x2c
#define BITNUM_FLAG_ONLY       0x2e
#define BITNUM_FLAG_ONLY_S     0x2f

// wordflags, flags in source word
#define FLAG_ALL_UPPER     0x1   // no lower case letters in the
word
#define FLAG_FIRST_UPPER   0x2   // first letter is upper case
#define FLAG_UPPERS        0x3   // FLAG_ALL_UPPER |
FLAG_FIRST_UPPER
#define FLAG_HAS_PLURAL    0x4   // upper-case word with s or 's
lower-case ending
#define FLAG_PHONEMES      0x8   // word is phonemes
#define FLAG_LAST_WORD     0x10  // last word in clause
#define FLAG_EMBEDDED      0x40  // word is preceded by embedded
commands
#define FLAG_HYPHEN        0x80
```

```
#define FLAG_NOSPACE       0x100 // word is not seperated from
previous word by a space
#define FLAG_FIRST_WORD    0x200 // first word in clause
#define FLAG_FOCUS         0x400 // the focus word of a clause
#define FLAG_EMPHASIZED    0x800
#define FLAG_EMPHASIZED2   0xc00 // FLAG_FOCUS | FLAG_EMPHASIZED
#define FLAG_DONT_SWITCH_TRANSLATOR  0x1000
#define FLAG_SUFFIX_REMOVED  0x2000
#define FLAG_HYPHEN_AFTER    0x4000
#define FLAG_ORDINAL       0x8000   // passed to
TranslateNumber() to indicate an ordinal number
#define FLAG_HAS_DOT       0x10000  // dot after this word
#define FLAG_COMMA_AFTER   0x20000  // comma after this word
#define FLAG_MULTIPLE_SPACES 0x40000  // word is preceded by
multiple spaces, newline, or tab
#define FLAG_INDIVIDUAL_DIGITS 0x80000  // speak number as
individual digits
#define FLAG_DELETE_WORD     0x100000   // don't speak this word,
it has been spoken as part of the previous word
#define FLAG_CHAR_REPLACED   0x200000   // characters have been
replaced by .replace in the *_rules
#define FLAG_TRANSLATOR2     0x400000   // retranslating using a
different language
#define FLAG_PREFIX_REMOVED  0x800000   // a prefix has been
removed from this word

#define FLAG_SUFFIX_VOWEL  0x08000000 // remember an initial
vowel from the suffix
#define FLAG_NO_TRACE      0x10000000 // passed to
TranslateRules() to suppress dictionary lookup printout
#define FLAG_NO_PREFIX     0x20000000
#define FLAG_UNPRON_TEST   0x80000000 // do unpronounability test
on the beginning of the word

// prefix/suffix flags (bits 8 to 14, bits 16 to 22) don't use
0x8000, 0x800000
#define SUFX_E        0x0100   // e may have been added
```

```
#define SUFX_I        0x0200   // y may have been changed to i
#define SUFX_P        0x0400   // prefix
#define SUFX_V        0x0800   // suffix means use the verb form
pronunciation
#define SUFX_D        0x1000   // previous letter may have been
doubled
#define SUFX_F        0x2000   // verb follows
#define SUFX_Q        0x4000   // don't retranslate
#define SUFX_T        0x10000   // don't affect the stress
position in the stem
#define SUFX_B        0x20000  // break, this character breaks
the word into stem and suffix (used with SUFX_P)
#define SUFX_A        0x40000  // remember that the suffix starts
with a vowel
#define SUFX_M        0x80000  // bit 19, allow multiple suffixes

#define SUFX_UNPRON     0x8000   // used to return $unpron flag
from *_rules

#define FLAG_ALLOW_TEXTMODE  0x02  // allow dictionary to
translate to text rather than phonemes
#define FLAG_SUFX       0x04
#define FLAG_SUFX_S     0x08
#define FLAG_SUFX_E_ADDED 0x10

// codes in dictionary rules
#define RULE_PRE         1
#define RULE_POST        2
#define RULE_PHONEMES    3
#define RULE_PH_COMMON   4 // At start of rule. Its phoneme
string is used by subsequent rules
#define RULE_CONDITION   5 // followed by condition number (byte)
#define RULE_GROUP_START 6
#define RULE_GROUP_END   7
#define RULE_PRE_ATSTART 8 // as RULE_PRE but also match with
'start of word'
#define RULE_LINENUM     9 // next 2 bytes give a line number,
```

```
for debugging purposes

#define RULE_STRESSED     10 // &
#define RULE_DOUBLE       11 // %
#define RULE_INC_SCORE    12 // +
#define RULE_DEL_FWD      13 // #
#define RULE_ENDING       14 // S
#define RULE_DIGIT        15 // D digit
#define RULE_NONALPHA     16 // Z non-alpha
#define RULE_LETTERGP     17 // A B C H F G Y   letter group
number
#define RULE_LETTERGP2    18 // L + letter group number
#define RULE_CAPITAL      19 // !   word starts with a capital
letter
#define RULE_REPLACEMENTS 20 // section for character
replacements
#define RULE_SYLLABLE     21 // @
#define RULE_SKIPCHARS    23 // J
#define RULE_NO_SUFFIX    24 // N
#define RULE_NOTVOWEL     25 // K
#define RULE_IFVERB       26 // V
#define RULE_DOLLAR       28 // $ commands
#define RULE_NOVOWELS     29 // X no vowels up to word boundary
#define RULE_SPELLING     31 // W while spelling letter-by-letter
#define RULE_LAST_RULE    31
// Rule codes above 31 are the ASCII code representation of the
character
// used to specify the rule.
#define RULE_SPACE        32 // ascii space
#define RULE_DEC_SCORE    60 // <

#define DOLLAR_UNPR     0x01
#define DOLLAR_NOPREFIX 0x02
#define DOLLAR_LIST     0x03

#define LETTERGP_A      0
#define LETTERGP_B      1
```

```
#define LETTERGP_C      2
#define LETTERGP_H      3
#define LETTERGP_F      4
#define LETTERGP_G      5
#define LETTERGP_Y      6
#define LETTERGP_VOWEL2 7

// Punctuation types returned by ReadClause()
//@{

#define CLAUSE_PAUSE                    0x00000FFF // pause (x
10mS)
#define CLAUSE_INTONATION_TYPE          0x00007000 // intonation
type
#define CLAUSE_OPTIONAL_SPACE_AFTER     0x00008000 // don't need
space after the punctuation
#define CLAUSE_TYPE                     0x000F0000 // phrase type
#define CLAUSE_PUNCTUATION_IN_WORD      0x00100000 // punctuation
character can be inside a word (Armenian)
#define CLAUSE_SPEAK_PUNCTUATION_NAME 0x00200000 // speak the
name of the punctuation character
#define CLAUSE_DOT_AFTER_LAST_WORD      0x00400000 // dot after the
last word
#define CLAUSE_PAUSE_LONG               0x00800000 // x 320mS to
the CLAUSE_PAUSE value

#define CLAUSE_INTONATION_FULL_STOP     0x00000000
#define CLAUSE_INTONATION_COMMA         0x00001000
#define CLAUSE_INTONATION_QUESTION      0x00002000
#define CLAUSE_INTONATION_EXCLAMATION 0x00003000
#define CLAUSE_INTONATION_NONE          0x00004000

#define CLAUSE_TYPE_NONE                0x00000000
#define CLAUSE_TYPE_EOF                 0x00010000
#define CLAUSE_TYPE_VOICE_CHANGE        0x00020000
#define CLAUSE_TYPE_CLAUSE              0x00040000
#define CLAUSE_TYPE_SENTENCE            0x00080000
```

```
#define CLAUSE_NONE         ( 0 | CLAUSE_INTONATION_NONE        |
CLAUSE_TYPE_NONE)
#define CLAUSE_PARAGRAPH    (70 | CLAUSE_INTONATION_FULL_STOP   |
CLAUSE_TYPE_SENTENCE)
#define CLAUSE_EOF          (40 | CLAUSE_INTONATION_FULL_STOP   |
CLAUSE_TYPE_SENTENCE | CLAUSE_TYPE_EOF)
#define CLAUSE_VOICE        ( 0 | CLAUSE_INTONATION_NONE        |
CLAUSE_TYPE_VOICE_CHANGE)
#define CLAUSE_PERIOD       (40 | CLAUSE_INTONATION_FULL_STOP   |
CLAUSE_TYPE_SENTENCE)
#define CLAUSE_COMMA        (20 | CLAUSE_INTONATION_COMMA       |
CLAUSE_TYPE_CLAUSE)
#define CLAUSE_SHORTCOMMA   ( 4 | CLAUSE_INTONATION_COMMA       |
CLAUSE_TYPE_CLAUSE)
#define CLAUSE_SHORTFALL    ( 4 | CLAUSE_INTONATION_FULL_STOP   |
CLAUSE_TYPE_CLAUSE)
#define CLAUSE_QUESTION     (40 | CLAUSE_INTONATION_QUESTION    |
CLAUSE_TYPE_SENTENCE)
#define CLAUSE_EXCLAMATION  (45 | CLAUSE_INTONATION_EXCLAMATION |
CLAUSE_TYPE_SENTENCE)
#define CLAUSE_COLON        (30 | CLAUSE_INTONATION_FULL_STOP   |
CLAUSE_TYPE_CLAUSE)
#define CLAUSE_SEMICOLON    (30 | CLAUSE_INTONATION_COMMA       |
CLAUSE_TYPE_CLAUSE)

//@}

#define SAYAS_CHARS         0x12
#define SAYAS_GLYPHS        0x13
#define SAYAS_SINGLE_CHARS  0x14
#define SAYAS_KEY           0x24
#define SAYAS_DIGITS        0x40 // + number of digits
#define SAYAS_DIGITS1       0xc1

#define CHAR_EMPHASIS    0x0530 // this is an unused character
code
```

```
#define CHAR_COMMA_BREAK 0x0557 // unused character code

// Rule:
// [4] [match] [1 pre] [2 post] [3 phonemes] 0
//     match 1 pre 2 post 0     - use common phoneme string
//     match 1 pre 2 post 3 0   - empty phoneme string

typedef const char *constcharptr;

// used to mark words with the source[] buffer
typedef struct {
 unsigned int flags;
 unsigned short start;
 unsigned char pre_pause;
 unsigned short sourceix;
 unsigned char length;
} WORD_TAB;

typedef struct {
 const char *name;
 int offset;
 unsigned short range_min, range_max;
 int language;
 int flags;
} ALPHABET;

// alphabet flags
#define AL_DONT_NAME    0x01 // don't speak the alphabet name
#define AL_NOT_LETTERS  0x02 // don't use the language for
speaking letters
#define AL_WORDS        0x04 // use the language to speak words
#define AL_NOT_CODE     0x08 // don't speak the character code
#define AL_NO_SYMBOL    0x10 // don't repeat "symbol" or
"character"

#define N_LOPTS       21
#define LOPT_DIERESES  1
```

```
// 1=remove [:] from unstressed syllables, 2= remove from
unstressed or non-penultimate syllables
// bit 4=0, if stress < 4,  bit 4=1, if not the highest stress in
the word
#define LOPT_IT_LENGTHEN 2

// 1=german
#define LOPT_PREFIXES 3

// non-zero, change voiced/unoiced to match last consonant in a
cluster
// bit 0=use regressive voicing
// bit 1=LANG=cz,bg  don't propagate over [v]
// bit 2=don't propagate acress word boundaries
// bit 3=LANG=pl,  propagate over liquids and nasals
// bit 4=LANG=cz,sk  don't progagate to [v]
// bit 8=devoice word-final consonants
#define LOPT_REGRESSIVE_VOICING 4

// 0=default, 1=no check, other allow this character as an extra
initial letter (default is 's')
#define LOPT_UNPRONOUNCABLE 5

// select length_mods tables,  (length_mod_tab) +
(length_mod_tab0 * 100)
#define LOPT_LENGTH_MODS 6

// increase this to prevent sonorants being shortened before
shortened (eg. unstressed) vowels
#define LOPT_SONORANT_MIN 7

// bit 0: don't break vowels at word boundary
#define LOPT_WORD_MERGE 8

// max. amplitude for vowel at the end of a clause
#define LOPT_MAXAMP_EOC 9
```

```
// bit 0=reduce even if phonemes are specified in the **_list
file
// bit 1=don't reduce the strongest vowel in a word which is
marked 'unstressed'
#define LOPT_REDUCE 10

// LANG=cs,sk  combine some prepositions with the following word,
if the combination has N or fewer syllables
// bits 0-3  N syllables
// bit 4=only if the second word has $alt attribute
// bit 5=not if the second word is end-of-sentence
#define LOPT_COMBINE_WORDS 11

// change [t] when followed by unstressed vowel
#define LOPT_REDUCE_T 12

// 1 = allow capitals inside a word
// 2 = stressed syllable is indicated by capitals
#define LOPT_CAPS_IN_WORD 13

// bit 0=Italian "syntactic doubling" of consoants in the word
after a word marked with $double attribute
// bit 1=also after a word which ends with a stressed vowel
#define LOPT_IT_DOUBLING 14

// Call ApplySpecialAttributes() if $alt or $alt2 is set for a
word
// bit 1: stressed syllable: $alt change [e],[o] to [E],[O],
$alt2 change [E],[O] to [e],[o]
#define LOPT_ALT 15

// pause for bracket (default=4), pause when annoucing bracket
names (default=2)
#define LOPT_BRACKET_PAUSE 16

// bit 1, don't break clause before annoucning . ? !
#define LOPT_ANNOUNCE_PUNCT 17
```

```
// recognize long vowels (0 = don't recognize)
#define LOPT_LONG_VOWEL_THRESHOLD 18

// bit 0:  Don't allow suffices if there is no previous syllable
#define LOPT_SUFFIX 19

// bit 0  Apostrophe at start of word is part of the word
// bit 1  Apostrophe at end of word is part of the word
#define LOPT_APOSTROPHE 20

// stress_rule
#define STRESSPOSN_1L 0 // 1st syllable
#define STRESSPOSN_2L 1 // 2nd syllable
#define STRESSPOSN_2R 2 // penultimate
#define STRESSPOSN_1R 3 // final syllable
#define STRESSPOSN_3R 4 // antipenultimate

typedef struct {
// bits0-2  separate words with (1=pause_vshort, 2=pause_short,
3=pause, 4=pause_long 5=[?] phonemme)
// bit 3=don't use linking phoneme
// bit4=longer pause before STOP, VSTOP,FRIC
// bit5=length of a final vowel doesn't depend on the next
phoneme
 int word_gap;
 int vowel_pause;
 int stress_rule; // 1=first syllable, 2=penultimate,  3=last

#define S_NO_DIM            0x02
#define S_FINAL_DIM         0x04
#define S_FINAL_DIM_ONLY    0x06
// bit1=don't set diminished stress,
// bit2=mark unstressed final syllables as diminished

// bit3=set consecutive unstressed syllables in unstressed words
to diminished, but not in stressed words
```

```
#define S_FINAL_NO_2        0x10
// bit4=don't allow secondary stress on last syllable

#define S_NO_AUTO_2         0x20
// bit5-don't use automatic secondary stress

#define S_2_TO_HEAVY        0x40
// bit6=light syllable followed by heavy, move secondary stress
to the heavy syllable. LANG=Finnish

#define S_FIRST_PRIMARY     0x80
// bit7=if more than one primary stress, make the subsequent
primaries to secondary stress

#define S_FINAL_VOWEL_UNSTRESSED    0x100
// bit8=don't apply default stress to a word-final vowel

#define S_FINAL_SPANISH     0x200
// bit9=stress last syllable if it doesn't end in vowel or "s" or
"n"  LANG=Spanish

#define S_2_SYL_2           0x1000
// bit12= In a 2-syllable word, if one has primary stress then
give the other secondary stress

#define S_INITIAL_2         0x2000
// bit13= If there is only one syllable before the primary
stress, give it a secondary stress

#define S_MID_DIM           0x10000
// bit 16= Set (not first or last) syllables to diminished stress

#define S_PRIORITY_STRESS   0x20000
// bit17= "priority" stress reduces other primary stress to
"unstressed" not "secondary"
```

```
#define S_EO_CLAUSE1        0x40000
// bit18= don't lengthen short vowels more than long vowels at
end-of-clause

#define S_FINAL_LONG         0x80000
// bit19=stress on final syllable if it has a long vowel, but
previous syllable has a short vowel

#define S_HYPEN_UNSTRESS    0x100000
// bit20= hyphenated words, 2nd part is unstressed

#define S_NO_EOC_LENGTHEN   0x200000
// bit21= don't lengthen vowels at end-of-clause

// bit15= Give stress to the first unstressed syllable

 int stress_flags;
 int unstressed_wd1; // stress for $u word of 1 syllable
 int unstressed_wd2; // stress for $u word of >1 syllable
 int param[N_LOPTS];
 int param2[N_LOPTS];
 unsigned char *length_mods;
 unsigned char *length_mods0;

#define NUM_THOUS_SPACE  0x4
#define NUM_DECIMAL_COMMA 0x8
#define NUM_SWAP_TENS    0x10
#define NUM_AND_UNITS    0x20
#define NUM_HUNDRED_AND  0x40
#define NUM_SINGLE_AND   0x80
#define NUM_SINGLE_STRESS 0x100
#define NUM_SINGLE_VOWEL 0x200
#define NUM_OMIT_1_HUNDRED 0x400
#define NUM_1900         0x800
#define NUM_ALLOW_SPACE  0x1000
#define NUM_DFRACTION_1  0x2000
#define NUM_DFRACTION_2  0x4000
```

```
#define NUM_DFRACTION_3  0x6000
#define NUM_DFRACTION_4  0x8000
#define NUM_DFRACTION_5  0xa000
#define NUM_DFRACTION_6  0xc000
#define NUM_DFRACTION_7  0xe000    // lang=si, alternative form
of number for decimal fraction digits (except the last)
#define NUM_ORDINAL_DOT   0x10000
#define NUM_NOPAUSE       0x20000
#define NUM_AND_HUNDRED   0x40000
#define NUM_THOUSAND_AND  0x80000
#define NUM_VIGESIMAL       0x100000
#define NUM_OMIT_1_THOUSAND 0x200000
#define NUM_ZERO_HUNDRED    0x400000
#define NUM_HUNDRED_AND_DIGIT   0x800000
#define NUM_ROMAN          0x1000000
#define NUM_ROMAN_CAPITALS 0x2000000
#define NUM_ROMAN_AFTER    0x4000000
#define NUM_ROMAN_ORDINAL  0x8000000
#define NUM_SINGLE_STRESS_L  0x10000000

 // bits0-1=which numbers routine to use.
 // bit2=  thousands separator must be space
 // bit3=  , decimal separator, not .
 // bit4=use three-and-twenty rather than twenty-three
 // bit5='and' between tens and units
 // bit6=add "and" after hundred or thousand
 // bit7=don't have "and" both after hundreds and also between
tens and units
 // bit8=only one primary stress in tens+units
 // bit9=only one vowel betwen tens and units
 // bit10=omit "one" before "hundred"
 // bit11=say 19** as nineteen hundred
 // bit12=allow space as thousands separator (in addition to
langopts.thousands_sep)
 // bits13-15  post-decimal-digits 0=single digits, 1=(LANG=it)
2=(LANG=pl) 3=(LANG=ro)
```

```
 // bit16= dot after number indicates ordinal
 // bit17= don't add pause after a number
 // bit18= 'and' before hundreds
 // bit19= 'and' after thousands if there are no hundreds
 // bit20= vigesimal number, if tens are not found
 // bit21= omit "one" before "thousand"
 // bit22= say "zero" before hundred
 // bit23= add "and" after hundreds and thousands, only if there
are digits and no tens

 // bit24= recognize roman numbers
 // bit25= Roman numbers only if upper case
 // bit26= say "roman" after the number, not before
 // bit27= Roman numbers are ordinal numbers
 // bit28= only one primary stress in tens+units (on the tens)
 int numbers;

#define NUM2_THOUSANDS_VAR1     0x40
#define NUM2_THOUSANDS_VAR2     0x80
#define NUM2_THOUSANDS_VAR3     0xc0
#define NUM2_THOUSANDS_VAR4     0x100
#define NUM2_THOUSANDS_VAR5     0x140

#define NUM2_SWAP_THOUSANDS     0x200
#define NUM2_ORDINAL_NO_AND     0x800
#define NUM2_MULTIPLE_ORDINAL   0x1000
#define NUM2_NO_TEEN_ORDINALS   0x2000
#define NUM2_MYRIADS            0x4000
#define NUM2_ENGLISH_NUMERALS   0x8000
#define NUM2_PERCENT_BEFORE     0x10000
#define NUM2_OMIT_1_HUNDRED_ONLY 0x20000
#define NUM2_ORDINAL_AND_THOUSANDS 0x40000
#define NUM2_ORDINAL_DROP_VOWEL  0x80000 // currently only for
tens and units
#define NUM2_ZERO_TENS          0x100000
 // bits 1-4  use variant form of numbers before
thousands,millions,etc.
```

```
 // bits 6-8  use different forms of thousand, million, etc (M MA
MB)
 // bit9=(LANG=rw) say "thousand" and "million" before its
number, not after
 // bit11=(LANG=es,an) don't say 'and' between tens and units for
ordinal numbers
 // bit12=(LANG=el,es) use ordinal form of hundreds and tens as
well as units
 // bit13=(LANG=pt) don't use 11-19 numbers to make ordinals
 // bit14=(LANG=ko)  use myriads (groups of 4 digits) not
thousands (groups of 3)
 // bit15=(LANG=ne)  speak (non-replaced) English numerals in
English
 // bit16=(LANG=si)  say "%" before the number
 // bit17=(LANG=ml)  omit "one" before hundred only if there are
no previous digits
 // bit18=(LANG=ta)  same variant for ordinals and thousands (#o
= #a)
 // bit19=(LANG=te)  drop final vowel from cardial number before
adding ordinal suffix
 // bit20=(LANG=zh)  say zero tens
 int numbers2;

// Bit 2^n is set if 10^n separates a number grouping (max n=31).
//                                      0         1         2
3
//                                  n =
01234567890123456789012345678901
#define BREAK_THOUSANDS   0x49249248 // b  b  b  b  b  b  b  b  b
b  b  // 10,000,000,000,000,000,000,000,000,000,000
#define BREAK_MYRIADS     0x11111110 // b   b   b   b   b   b   b
b    // 1000,0000,0000,0000,0000,0000,0000,0000
#define BREAK_LAKH        0xaaaaaaa8 // b  b b b b b b b b b b b
b b b b // 10,00,00,00,00,00,00,00,00,00,00,00,00,00,00,000
#define BREAK_LAKH_BN     0x24924aa8 // b  b b b b b  b  b  b  b
b  b   // 100,000,000,000,000,000,000,00,00,00,00,000
#define BREAK_LAKH_DV     0x000014a8 // b  b b b  b b
```

```
// 100,00,000,00,00,000
#define BREAK_LAKH_HI     0x00014aa8 // b  b b b b b  b b
// 100,00,000,00,00,00,00,000
#define BREAK_LAKH_UR     0x000052a8 // b  b b b b  b b
// 100,00,000,00,00,00,000
#define BREAK_INDIVIDUAL  0x00000018 // b  bb
// 100,0,000

 int break_numbers;  // which digits to break the number into
thousands, millions, etc (Hindi has 100,000 not 1,000,000)
 int max_roman;
 int min_roman;
 int thousands_sep;
 int decimal_sep;
 int max_digits;    // max number of digits which can be spoken
as an integer number (rather than individual digits)
 const char *ordinal_indicator;   // UTF-8 string
 const unsigned char *roman_suffix;    // add this (ordinal)
suffix to Roman numbers (LANG=an)

 // bit 0, accent name before the letter name, bit 1 "capital"
after letter name
 int accents;

 int tone_language;          // 1=tone language
 int intonation_group;
 unsigned char tunes[6];
 int long_stop;          // extra mS pause for a lengthened stop
 char max_initial_consonants;
 char spelling_stress;   // 0=default, 1=stress first letter
 char tone_numbers;
 char ideographs;      // treat as separate words
 bool textmode;          // the meaning of FLAG_TEXTMODE is
reversed (to save data when *_list file is compiled)
 char dotless_i;         // uses letter U+0131
 int listx;    // compile *_listx after *list
 const unsigned char *replace_chars;      // characters to be
```

```
substitutes
 int our_alphabet;           // offset for main alphabet (if not
set in letter_bits_offset)
 int alt_alphabet;       // offset for another language to
recognize
 int alt_alphabet_lang;  // language for the alt_alphabet
 int max_lengthmod;
 int lengthen_tonic;   // lengthen the tonic syllable
 int suffix_add_e;      // replace a suffix (which has the SUFX_E
flag) with this character
} LANGUAGE_OPTIONS;

typedef struct {
 LANGUAGE_OPTIONS langopts;
 int translator_name;
 int transpose_max;
 int transpose_min;
 const char *transpose_map;
 char dictionary_name[40];

 char phonemes_repeat[20];
 int phonemes_repeat_count;
 int phoneme_tab_ix;

 unsigned char stress_amps[8];
 unsigned char stress_amps_r[8];
 short stress_lengths[8];
 int dict_condition;    // conditional apply some pronunciation
rules and dict.lookups
 int dict_min_size;
 espeak_ng_ENCODING encoding;
 const wchar_t *char_plus_apostrophe;  // single chars +
apostrophe treated as words
 const wchar_t *punct_within_word;   // allow these punctuation
characters within words
 const unsigned short *chars_ignore;
```

```
// holds properties of characters: vowel, consonant, etc for
pronunciation rules
 unsigned char letter_bits[256];
 int letter_bits_offset;
 const wchar_t *letter_groups[8];

 /* index1=option, index2 by 0=. 1=, 2=?, 3=! 4=none */
 #define INTONATION_TYPES 8
 #define PUNCT_INTONATIONS 6
 unsigned char
punct_to_tone[INTONATION_TYPES][PUNCT_INTONATIONS];

 char *data_dictrules;     // language_1   translation rules file
 char *data_dictlist;      // language_2   dictionary lookup file
 char *dict_hashtab[N_HASH_DICT];   // hash table to index
dictionary lookup file
 char *letterGroups[N_LETTER_GROUPS];

 // groups1 and groups2 are indexes into data_dictrules, set up
by InitGroups()
 // the two-letter rules for each letter must be consecutive in
the language_rules source

 char *groups1[256];         // translation rule lists, index by
single letter
 char *groups3[128];         // index by offset letter
 char *groups2[N_RULE_GROUP2];   // translation rule lists,
indexed by two-letter pairs
 unsigned int groups2_name[N_RULE_GROUP2];  // the two letter
pairs for groups2[]
 int n_groups2;              // number of groups2[] entries used

 unsigned char groups2_count[256];    // number of 2 letter
groups for this initial letter
 unsigned char groups2_start[256];    // index into groups2
 const short *frequent_pairs;   // list of frequent pairs of
letters, for use in compressed *_list
```

```
 int expect_verb;
 int expect_past;    // expect past tense
 int expect_verb_s;
 int expect_noun;
 int prev_last_stress;
 char *clause_end;

 int word_vowel_count;     // number of vowels so far
 int word_stressed_count;  // number of vowels so far which could
be stressed

 int clause_upper_count;   // number of upper case letters in the
clause
 int clause_lower_count;   // number of lower case letters in the
clause

 int prepause_timeout;
 int end_stressed_vowel;  // word ends with stressed vowel
 int prev_dict_flags[2];     // dictionary flags from previous
word
 int clause_terminator;
} Translator;

#define OPTION_EMPHASIZE_ALLCAPS  0x100
#define OPTION_EMPHASIZE_PENULTIMATE 0x200
extern int option_tone_flags;
extern int option_phonemes;
extern int option_phoneme_events;
extern int option_linelength;     // treat lines shorter than
this as end-of-clause
extern int option_capitals;
extern int option_punctuation;
extern int option_endpause;
extern int option_ssml;
extern int option_phoneme_input;   // allow [[phonemes]] in input
text
```

```
extern int option_sayas;
extern int option_wordgap;

extern int count_characters;
extern int count_sentences;
extern int skip_characters;
extern int skip_words;
extern int skip_sentences;
extern bool skipping_text;
extern int end_character_position;
extern int clause_start_char;
extern int clause_start_word;
extern char *namedata;
extern int pre_pause;

#define N_MARKER_LENGTH 50   // max.length of a mark name
extern char skip_marker[N_MARKER_LENGTH];

#define N_PUNCTLIST  60
extern wchar_t option_punctlist[N_PUNCTLIST];  // which
punctuation characters to announce

extern Translator *translator;
extern Translator *translator2;
extern char dictionary_name[40];
extern char ctrl_embedded;    // to allow an alternative CTRL for
embedded commands
extern espeak_ng_TEXT_DECODER *p_decoder;
extern int dictionary_skipwords;

extern int (*uri_callback)(int, const char *, const char *);
extern int (*phoneme_callback)(const char *);
extern void SetLengthMods(Translator *tr, int value);

#define LEADING_2_BITS 0xC0 // 0b11000000
#define UTF8_TAIL_BITS 0x80 // 0b10000000
```

```c
ESPEAK_NG_API int utf8_in(int *c, const char *buf);
int utf8_in2(int *c, const char *buf, int backwards);
int utf8_out(unsigned int c, char *buf);
int utf8_nbytes(const char *buf);

int lookupwchar(const unsigned short *list, int c);
int lookupwchar2(const unsigned short *list, int c);
char *strchr_w(const char *s, int c);
int IsBracket(int c);
void InitNamedata(void);
void InitText(int flags);
void InitText2(void);
int IsDigit(unsigned int c);
int IsDigit09(unsigned int c);
int IsAlpha(unsigned int c);
int isspace2(unsigned int c);
ALPHABET *AlphabetFromChar(int c);

Translator *SelectTranslator(const char *name);
int SetTranslator2(const char *name);
void DeleteTranslator(Translator *tr);
void ProcessLanguageOptions(LANGUAGE_OPTIONS *langopts);

void print_dictionary_flags(unsigned int *flags, char *buf, int
buf_len);

void ApplySpecialAttribute2(Translator *tr, char *phonemes, int
dict_flags);

int TranslateWord(Translator *tr, char *word1, WORD_TAB *wtab,
char *word_out);
void TranslateClause(Translator *tr, int *tone, char
**voice_change);

void SetVoiceStack(espeak_VOICE *v, const char *variant_name);

extern FILE *f_trans; // for logging
```

```
#ifdef __cplusplus
}
#endif

#endif
```

Chapter 12

./src/libespeak-ng/klatt.h

```
#ifndef ESPEAK_NG_KLATT_H
#define ESPEAK_NG_KLATT_H

#include "speech.h"
#include "synthesize.h"

#ifdef __cplusplus
extern "C"
{
#endif

#define CASCADE_PARALLEL 1         /* Type of synthesis model */
#define ALL_PARALLEL     2

#define IMPULSIVE        1         /* Type of voicing source */
#define NATURAL          2
#define SAMPLED          3
#define SAMPLED2         4

typedef long flag;

typedef struct {
```

```
 double a;
 double b;
 double c;
 double p1;
 double p2;
 double a_inc;
 double b_inc;
 double c_inc;
} resonator_t, *resonator_ptr;

typedef struct {
 flag synthesis_model; /* cascade-parallel or all-parallel */
 flag outsl;     /* Output waveform selector
*/
 long samrate;   /* Number of output samples per second
*/
 long FLPhz;     /* Frequeny of glottal downsample low-pass
filter */
 long BLPhz;     /* Bandwidth of glottal downsample low-pass
filter */
 flag glsource;  /* Type of glottal source */
 int f0_flutter; /* Percentage of f0 flutter 0-100 */
 long nspfr;     /* number of samples per frame */
 long nper;      /* Counter for number of samples in a pitch
period */
 long ns;
 long T0;        /* Fundamental period in output samples times 4
*/
 long nopen;     /* Number of samples in open phase of period
*/
 long nmod;      /* Position in period to begin noise amp. modul
*/
 long nrand;     /* Varible used by random number generator
*/
 double pulse_shape_a; /* Makes waveshape of glottal pulse when
open   */
 double pulse_shape_b; /* Makes waveshape of glottal pulse when
```

```
open   */
 double minus_pi_t;
 double two_pi_t;
 double onemd;
 double decay;
 double amp_bypas; /* AB converted to linear gain              */
 double amp_voice; /* AVdb converted to linear gain            */
 double par_amp_voice; /* AVpdb converted to linear gain       */
 double amp_aspir; /* AP converted to linear gain              */
 double amp_frica; /* AF converted to linear gain              */
 double amp_breth; /* ATURB converted to linear gain           */
 double amp_gain0; /* G0 converted to linear gain              */
 int num_samples; /* number of glottal samples */
 double sample_factor; /* multiplication factor for glottal
samples */
 short *natural_samples; /* pointer to an array of glottal
samples */
 long original_f0; /* original value of f0 not modified by
flutter */

 int fadeout;       // set to 64 to cause fadeout over 64 samples
 int scale_wav;     // depends on the voicing source

#define N_RSN 20
#define Rnz  0   // nasal zero, anti-resonator
#define R1c  1
#define R2c  2
#define R3c  3
#define R4c  4
#define R5c  5
#define R6c  6
#define R7c  7
#define R8c  8
#define Rnpc 9   // nasal pole

#define Rparallel 10
#define Rnpp 10
```

```
#define R1p  11
#define R2p  12
#define R3p  13
#define R4p  14
#define R5p  15
#define R6p  16

#define RGL  17
#define RLP  18
#define Rout 19

 resonator_t rsn[N_RSN];  // internal storage for resonators
 resonator_t rsn_next[N_RSN];

} klatt_global_t, *klatt_global_ptr;

#define F_NZ   0  // nasal zero formant
#define F1     1
#define F2     2
#define F3     3
#define F4     4
#define F5     5
#define F6     6
#define F_NP   9  // nasal pole formant

typedef struct {
 int F0hz10; /* Voicing fund freq in Hz
*/
 int AVdb;   /* Amp of voicing in dB,            0 to   70
*/
 int Fhz[10];  // formant Hz, F_NZ to F6 to F_NP
 int Bhz[10];
 int Ap[10];   /* Amp of parallel formants in dB,    0 to   80
*/
 int Bphz[10]; /* Parallel formants bw in Hz,       40 to 1000
*/
```

```
 int ASP;    /* Amp of aspiration in dB,         0 to   70
*/
 int Kopen;  /* # of samples in open period,     10 to   65
*/
 int Aturb;  /* Breathiness in voicing,          0 to   80
*/
 int TLTdb;  /* Voicing spectral tilt in dB,     0 to   24
*/
 int AF;     /* Amp of frication in dB,          0 to   80
*/
 int Kskew;  /* Skewness of alternate periods,   0 to   40 in
sample#/2  */

 int AB;     /* Amp of bypass fric. in dB,       0 to   80
*/
 int AVpdb;  /* Amp of voicing,  par in dB,      0 to   70
*/
 int Gain0;  /* Overall gain, 60 dB is unity,    0 to   60
*/

 int AVdb_tmp;      // copy of AVdb, which is changed within
parwave()
 int Fhz_next[10];    // Fhz for the next chunk, so we can do
interpolation of resonator (a,b,c) parameters
 int Bhz_next[10];
} klatt_frame_t, *klatt_frame_ptr;

typedef struct {
 int freq;     // Hz
 int bw;   // klatt bandwidth
 int ap;   // parallel amplitude
 int bp;   // parallel bandwidth
 double freq1; // floating point versions of the above
 double bw1;
 double ap1;
 double bp1;
 double freq_inc;    // increment by this every 64 samples
```

```
 double bw_inc;
 double ap_inc;
 double bp_inc;
}  klatt_peaks_t;

void KlattInit(void);
void KlattReset(int control);
int Wavegen_Klatt2(int length, int resume, frame_t *fr1, frame_t
*fr2);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 13

./src/libespeak-ng/phonemelist.h

```
#ifndef ESPEAK_NG_PHONEMELIST_H
#define ESPEAK_NG_PHONEMELIST_H

#include "synthesize.h"
#include "translate.h"

#ifdef __cplusplus
extern "C"
{
#endif

#ifdef __cplusplus
}
#endif

void MakePhonemeList(Translator *tr,
 int post_pause,
 bool start_sentence,
 int *n_ph_list2,
 PHONEME_LIST2 *ph_list2);

#endif
```

Chapter 14

./src/libespeak-ng/synthesize.h

```
#ifndef ESPEAK_NG_SYNTHESIZE_H
#define ESPEAK_NG_SYNTHESIZE_H

#ifdef __cplusplus
extern "C"
{
#endif

#include <stdint.h>
#include <stdbool.h>
#include <espeak-ng/espeak_ng.h>
#include "phoneme.h"
#include "voice.h"

#define espeakINITIALIZE_PHONEME_IPA 0x0002 // move this to
speak_lib.h, after eSpeak version 1.46.02

#define N_PHONEME_LIST 1000 // enough for source[N_TR_SOURCE]
full of text, else it will truncate

#define N_SEQ_FRAMES  25 // max frames in a spectrum sequence
(real max is ablut 8)
```

```
#define STEPSIZE      64 // 2.9mS at 22 kHz sample rate

// flags set for frames within a spectrum sequence
#define FRFLAG_KLATT           0x01 // this frame includes extra
data for Klatt synthesizer
#define FRFLAG_VOWEL_CENTRE    0x02 // centre point of vowel
#define FRFLAG_LEN_MOD         0x04 // reduce effect of length
adjustment
#define FRFLAG_BREAK_LF        0x08 // but keep f3 upwards
#define FRFLAG_BREAK           0x10 // don't merge with next
frame
#define FRFLAG_BREAK_2         0x18 // FRFLAG_BREAK_LF or
FRFLAG_BREAK
#define FRFLAG_FORMANT_RATE    0x20 // Flag5 allow increased rate
of change of formant freq
#define FRFLAG_MODULATE        0x40 // Flag6 modulate amplitude
of some cycles to give trill
#define FRFLAG_DEFER_WAV       0x80 // Flag7 defer mixing WAV
until the next frame
#define FRFLAG_LEN_MOD2      0x4000 // reduce effect of length
adjustment, used for the start of a vowel
#define FRFLAG_COPIED        0x8000 // This frame has been copied
into temporary rw memory

#define SFLAG_SEQCONTINUE      0x01 // a liquid or nasal after a
vowel, but not followed by a vowel
#define SFLAG_EMBEDDED         0x02 // there are embedded
commands before this phoneme
#define SFLAG_SYLLABLE         0x04 // vowel or syllabic
consonant
#define SFLAG_LENGTHEN         0x08 // lengthen symbol : included
after this phoneme
#define SFLAG_DICTIONARY       0x10 // the pronunciation of this
word was listed in the xx_list dictionary
#define SFLAG_SWITCHED_LANG    0x20 // this word uses phonemes
from a different language
#define SFLAG_PROMOTE_STRESS   0x40 // this unstressed word can
```

```
be promoted to stressed

#define SFLAG_PREV_PAUSE     0x1000 // consider previous phoneme
as pause
#define SFLAG_NEXT_PAUSE     0x2000 // consider next phoneme as
pause

// embedded command numbers
#define EMBED_P     1 // pitch
#define EMBED_S     2 // speed (used in setlengths)
#define EMBED_A     3 // amplitude/volume
#define EMBED_R     4 // pitch range/expression
#define EMBED_H     5 // echo/reverberation
#define EMBED_T     6 // different tone for announcing
punctuation (not used)
#define EMBED_I     7 // sound icon
#define EMBED_S2    8 // speed (used in synthesize)
#define EMBED_Y     9 // say-as commands
#define EMBED_M    10 // mark name
#define EMBED_U    11 // audio uri
#define EMBED_B    12 // break
#define EMBED_F    13 // emphasis
#define EMBED_C    14 // capital letter indication

#define N_EMBEDDED_VALUES    15
extern int embedded_value[N_EMBEDDED_VALUES];
extern int embedded_default[N_EMBEDDED_VALUES];

#define N_PEAKS2  9 // plus Notch and Fill (not yet implemented)
#define N_MARKERS 8

#define N_KLATTP   10 // this affects the phoneme data file
format
#define N_KLATTP2  14 // used in vowel files, with extra
parameters for future extensions

#define KLATT_AV      0
```

```
#define KLATT_FNZ     1 // nasal zero freq
#define KLATT_Tilt    2
#define KLATT_Aspr    3
#define KLATT_Skew    4

#define KLATT_Kopen   5
#define KLATT_AVp     6
#define KLATT_Fric    7
#define KLATT_FricBP  8
#define KLATT_Turb    9

typedef struct { // 64 bytes
 short frflags;
 short ffreq[7];
 unsigned char length;
 unsigned char rms;
 unsigned char fheight[8];
 unsigned char fwidth[6];   // width/4  f0-5
 unsigned char fright[3];   // width/4  f0-2
 unsigned char bw[4];       // Klatt bandwidth BNZ /2, f1,f2,f3
 unsigned char klattp[5];   // AV, FNZ, Tilt, Aspr, Skew
 unsigned char klattp2[5];  // continuation of klattp[],  Avp,
Fric, FricBP, Turb
 unsigned char klatt_ap[7]; // Klatt parallel amplitude
 unsigned char klatt_bp[7]; // Klatt parallel bandwidth  /2
 unsigned char spare;       // pad to multiple of 4 bytes
} frame_t; // with extra Klatt parameters for parallel resonators

typedef struct { // 44 bytes
 short frflags;
 short ffreq[7];
 unsigned char length;
 unsigned char rms;
 unsigned char fheight[8];
 unsigned char fwidth[6];  // width/4  f0-5
 unsigned char fright[3];  // width/4  f0-2
 unsigned char bw[4];      // Klatt bandwidth BNZ /2, f1,f2,f3
```

```
 unsigned char klattp[5];  // AV, FNZ, Tilt, Aspr, Skew
} frame_t2; // without the extra Klatt parameters

typedef struct {
 unsigned char *pitch_env;
 int pitch;      // pitch Hz*256
 int pitch_ix;   // index into pitch envelope (*256)
 int pitch_inc;  // increment to pitch_ix
 int pitch_base; // Hz*256 low, before modified by envelope
 int pitch_range; // Hz*256 range of envelope

 unsigned char *mix_wavefile; // wave file to be added to
synthesis
 int n_mix_wavefile; // length in bytes
 int mix_wave_scale; // 0=2 byte samples
 int mix_wave_amp;
 int mix_wavefile_ix;
 int mix_wavefile_max; // length of available WAV data (in bytes)
 int mix_wavefile_offset;

 int amplitude;
 int amplitude_v;
 int amplitude_fmt; // percentage amplitude adjustment for
formant synthesis
} WGEN_DATA;

typedef struct {
 double a;
 double b;
 double c;
 double x1;
 double x2;
} RESONATOR;

typedef struct {
 short length_total; // not used
 unsigned char n_frames;
```

```
 unsigned char sqflags;
 frame_t2 frame[N_SEQ_FRAMES]; // max. frames in a spectrum
sequence
} SPECT_SEQ; // sequence of espeak formant frames

typedef struct {
 short length_total; // not used
 unsigned char n_frames;
 unsigned char sqflags;
 frame_t frame[N_SEQ_FRAMES]; // max. frames in a spectrum
sequence
} SPECT_SEQK; // sequence of klatt formants frames

typedef struct {
 short length;
 short frflags;
 frame_t *frame;
} frameref_t;

// a clause translated into phoneme codes (first stage)
typedef struct {
 unsigned short synthflags; // NOTE Put shorts on 32bit
boundaries, because of RISC OS compiler bug?
 unsigned char phcode;
 unsigned char stresslevel;
 unsigned short sourceix;  // ix into the original source text
string, only set at the start of a word
 unsigned char wordstress; // the highest level stress in this
word
 unsigned char tone_ph;    // tone phoneme to use with this vowel
} PHONEME_LIST2;

#define PHLIST_START_OF_WORD     1
#define PHLIST_END_OF_CLAUSE     2
#define PHLIST_START_OF_SENTENCE 4
#define PHLIST_START_OF_CLAUSE   8
```

```
typedef struct {
 // The first section is a copy of PHONEME_LIST2
 unsigned short synthflags;
 unsigned char phcode;
 unsigned char stresslevel;
 unsigned short sourceix;  // ix into the original source text
string, only set at the start of a word
 unsigned char wordstress; // the highest level stress in this
word
 unsigned char tone_ph;    // tone phoneme to use with this vowel

 PHONEME_TAB *ph;
 unsigned int length;  // length_mod
 unsigned char env;    // pitch envelope number
 unsigned char type;
 unsigned char prepause;
 unsigned char postpause;
 unsigned char amp;
 unsigned char newword;   // bit flags, see
PHLIST_(START|END)_OF_*
 unsigned char pitch1;
 unsigned char pitch2;
 unsigned char std_length;
 unsigned int phontab_addr;
 int sound_param;
} PHONEME_LIST;

#define pd_FMT    0
#define pd_WAV    1
#define pd_VWLSTART 2
#define pd_VWLEND 3
#define pd_ADDWAV 4

#define N_PHONEME_DATA_PARAM 16
#define pd_INSERTPHONEME   i_INSERT_PHONEME
#define pd_APPENDPHONEME   i_APPEND_PHONEME
#define pd_CHANGEPHONEME   i_CHANGE_PHONEME
```

```
#define pd_CHANGE_NEXTPHONEME  i_REPLACE_NEXT_PHONEME
#define pd_LENGTHMOD       i_SET_LENGTH

#define pd_FORNEXTPH     0x2
#define pd_DONTLENGTHEN  0x4
#define pd_REDUCELENGTHCHANGE 0x8
typedef struct {
 int pd_control;
 int pd_param[N_PHONEME_DATA_PARAM];  // set from group 0
instructions
 int sound_addr[5];
 int sound_param[5];
 int vowel_transition[4];
 int pitch_env;
 int amp_env;
 char ipa_string[18];
} PHONEME_DATA;

typedef struct {
 int fmt_control;
 int use_vowelin;
 int fmt_addr;
 int fmt_length;
 int fmt_amp;
 int fmt2_addr;
 int fmt2_lenadj;
 int wav_addr;
 int wav_amp;
 int transition0;
 int transition1;
 int std_length;
} FMT_PARAMS;

typedef struct {
 PHONEME_LIST prev_vowel;
} WORD_PH_DATA;
```

```
// instructions

#define INSTN_RETURN         0x0001
#define INSTN_CONTINUE       0x0002

// Group 0 instrcutions with 8 bit operand.  These values go into
bits 8-15 of the instruction
#define i_CHANGE_PHONEME 0x01
#define i_REPLACE_NEXT_PHONEME 0x02
#define i_INSERT_PHONEME 0x03
#define i_APPEND_PHONEME 0x04
#define i_APPEND_IFNEXTVOWEL 0x05
#define i_VOICING_SWITCH 0x06
#define i_PAUSE_BEFORE   0x07
#define i_PAUSE_AFTER    0x08
#define i_LENGTH_MOD     0x09
#define i_SET_LENGTH     0x0a
#define i_LONG_LENGTH    0x0b
#define i_ADD_LENGTH     0x0c
#define i_IPA_NAME       0x0d

#define i_CHANGE_IF      0x10  // 0x10 to 0x14

// conditions and jumps
#define i_CONDITION  0x2000
#define i_OR         0x1000  // added to i_CONDITION
#define i_NOT        0x0003

#define i_JUMP       0x6000
#define i_JUMP_FALSE 0x6800
#define i_SWITCH_NEXTVOWEL 0x6a00
#define i_SWITCH_PREVVOWEL 0x6c00
#define MAX_JUMP     255  // max jump distance

// multi-word instructions
#define i_CALLPH     0x9100
#define i_PITCHENV   0x9200
```

```
#define i_AMPENV     0x9300
#define i_VOWELIN    0xa100
#define i_VOWELOUT   0xa200
#define i_FMT        0xb000
#define i_WAV        0xc000
#define i_VWLSTART   0xd000
#define i_VWLENDING  0xe000
#define i_WAVADD     0xf000

// conditions
#define CONDITION_IS_PHONEME_TYPE 0x00
#define CONDITION_IS_PLACE_OF_ARTICULATION 0x20
#define CONDITION_IS_PHFLAG_SET 0x40
#define CONDITION_IS_OTHER 0x80

// other conditions (stress)
#define STRESS_IS_DIMINISHED    0       // diminished, unstressed
within a word
#define STRESS_IS_UNSTRESSED    1       // unstressed, weak
#define STRESS_IS_NOT_STRESSED  2       // default, not stressed
#define STRESS_IS_SECONDARY     3       // secondary stress
#define STRESS_IS_PRIMARY       4       // primary (main) stress
#define STRESS_IS_PRIORITY      5       // replaces primary
markers
#define STRESS_IS_EMPHASIZED 6       // emphasized

// other conditions
#define isAfterStress  9
#define isNotVowel    10
#define isFinalVowel  11
#define isVoiced      12 // voiced consonant, or vowel
#define isFirstVowel  13
#define isSecondVowel 14
#define isTranslationGiven 16 // phoneme translation given in
**_list or as [[...]]
#define isBreak        17 // pause phoneme or (stop/vstop/vfric
not followed by vowel or (liquid in same word))
```

```
#define isWordStart    18
#define isWordEnd      19

#define i_StressLevel  0x800

typedef struct {
 int name;
 int length;
 char *data;
 char *filename;
} SOUND_ICON;

typedef struct {
 int pause_factor;
 int clause_pause_factor;
 unsigned int min_pause;
 int wav_factor;
 int lenmod_factor;
 int lenmod2_factor;
 int min_sample_len;
 int loud_consonants;
 int fast_settings[8];
} SPEED_FACTORS;

typedef struct {
 char name[12];
 unsigned char flags[4];
 signed char head_extend[8];

 unsigned char prehead_start;
 unsigned char prehead_end;
 unsigned char stressed_env;
 unsigned char stressed_drop;
 unsigned char secondary_drop;
 unsigned char unstressed_shape;

 unsigned char onset;
```

```
 unsigned char head_start;
 unsigned char head_end;
 unsigned char head_last;

 unsigned char head_max_steps;
 unsigned char n_head_extend;

 signed char unstr_start[3]; // for: onset, head, last
 signed char unstr_end[3];

 unsigned char nucleus0_env; // pitch envelope, tonic syllable is
at end, no tail
 unsigned char nucleus0_max;
 unsigned char nucleus0_min;

 unsigned char nucleus1_env; // when followed by a tail
 unsigned char nucleus1_max;
 unsigned char nucleus1_min;
 unsigned char tail_start;
 unsigned char tail_end;

 unsigned char split_nucleus_env;
 unsigned char split_nucleus_max;
 unsigned char split_nucleus_min;
 unsigned char split_tail_start;
 unsigned char split_tail_end;
 unsigned char split_tune;

 unsigned char spare[8];
 int spare2; // the struct length should be a multiple of 4 bytes
} TUNE;

extern int n_tunes;
extern TUNE *tunes;

// phoneme table
extern PHONEME_TAB *phoneme_tab[N_PHONEME_TAB];
```

```
// list of phonemes in a clause
extern int n_phoneme_list;
extern PHONEME_LIST phoneme_list[N_PHONEME_LIST+1];
extern unsigned int embedded_list[];

extern unsigned char env_fall[128];
extern unsigned char env_rise[128];
extern unsigned char env_frise[128];

#define MAX_PITCH_VALUE  101
extern unsigned char pitch_adjust_tab[MAX_PITCH_VALUE+1];

// queue of commands for wavegen
#define WCMD_KLATT  1
#define WCMD_KLATT2 2
#define WCMD_SPECT  3
#define WCMD_SPECT2 4
#define WCMD_PAUSE  5
#define WCMD_WAVE    6
#define WCMD_WAVE2   7
#define WCMD_AMPLITUDE 8
#define WCMD_PITCH  9
#define WCMD_MARKER 10
#define WCMD_VOICE   11
#define WCMD_EMBEDDED 12
#define WCMD_MBROLA_DATA 13
#define WCMD_FMT_AMPLITUDE 14
#define WCMD_SONIC_SPEED 15

#define N_WCMDQ   170
#define MIN_WCMDQ  25   // need this many free entries before
adding new phoneme

extern intptr_t wcmdq[N_WCMDQ][4];
extern int wcmdq_head;
extern int wcmdq_tail;
```

```
void MarkerEvent(int type, unsigned int char_position, int value,
int value2, unsigned char *out_ptr);

extern unsigned char *wavefile_data;
extern int samplerate;
extern int samplerate_native;

extern int wavefile_ix;
extern int wavefile_amp;
extern int vowel_transition[4];

#define N_ECHO_BUF 5500   // max of 250mS at 22050 Hz
extern int echo_head;
extern int echo_tail;
extern int echo_amp;
extern short echo_buf[N_ECHO_BUF];

void SynthesizeInit(void);
int  Generate(PHONEME_LIST *phoneme_list, int *n_ph, bool
resume);
void MakeWave2(PHONEME_LIST *p, int n_ph);
int  SpeakNextClause(int control);
void SetSpeed(int control);
void SetEmbedded(int control, int value);
int FormantTransition2(frameref_t *seq, int *n_frames, unsigned
int data1, unsigned int data2, PHONEME_TAB *other_ph, int which);

void Write4Bytes(FILE *f, int value);

#if HAVE_SONIC_H
void DoSonicSpeed(int value);
#endif

#define ENV_LEN  128    // length of pitch envelopes
#define PITCHfall   0  // standard pitch envelopes
#define PITCHrise   2
```

```
#define N_ENVELOPE_DATA   20
extern unsigned char *envelope_data[N_ENVELOPE_DATA];

extern int formant_rate[];         // max rate of change of each
formant
extern SPEED_FACTORS speed;

extern unsigned char *out_ptr;
extern unsigned char *out_start;
extern unsigned char *out_end;
extern espeak_EVENT *event_list;
extern t_espeak_callback *synth_callback;
extern const int version_phdata;

#define N_SOUNDICON_TAB  80   // total entries in soundicon_tab
#define N_SOUNDICON_SLOTS 4    // number of slots reserved for
dynamic loading of audio files
extern int n_soundicon_tab;
extern SOUND_ICON soundicon_tab[N_SOUNDICON_TAB];

void DoEmbedded(int *embix, int sourceix);
void DoMarker(int type, int char_posn, int length, int value);
void DoPhonemeMarker(int type, int char_posn, int length, char
*name);
int DoSample3(PHONEME_DATA *phdata, int length_mod, int amp);
int DoSpect2(PHONEME_TAB *this_ph, int which, FMT_PARAMS
*fmt_params,  PHONEME_LIST *plist, int modulation);
int PauseLength(int pause, int control);
const char *WordToString(unsigned int word);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 15

./src/libespeak-ng/mbrowrap.h

```
#ifndef MBROWRAP_H
#define MBROWRAP_H

#ifdef __cplusplus
extern "C"
{
#endif

#if !defined(_WIN32) && !defined(_WIN64)
#define WINAPI
typedef int BOOL;
#endif

extern int (WINAPI *init_MBR)(char *voice_path);

extern void (WINAPI *close_MBR)(void);

extern void (WINAPI *reset_MBR)(void);

extern int (WINAPI *read_MBR)(short *buffer, int nb_samples);

extern int (WINAPI *write_MBR)(char *data);
```

```
extern int (WINAPI *flush_MBR)(void);

extern int (WINAPI *getFreq_MBR)(void);

extern void (WINAPI *setVolumeRatio_MBR)(float value);

extern char * (WINAPI *lastErrorStr_MBR)(char *buffer, int
bufsize);

extern void (WINAPI *setNoError_MBR)(int no_error);

BOOL load_MBR(void);
void unload_MBR(void);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 16

./src/libespeak-ng/fifo.h

```
// Helps to add espeak commands in a first-in first-out queue
// and run them asynchronously.

#ifndef ESPEAK_NG_FIFO_H
#define ESPEAK_NG_FIFO_H

#include <espeak-ng/espeak_ng.h>
#include "espeak_command.h"

#ifdef __cplusplus
extern "C"
{
#endif

// Initialize the fifo component.
// First function to be called.
void fifo_init(void);

// Add an espeak command.
//
// Note: this function fails if too many commands are already
buffered.
```

```
// In such a case, the calling function could wait and then add
again its command.
espeak_ng_STATUS fifo_add_command(t_espeak_command *c);

// Add two espeak commands in a single transaction.
//
// Note: this function fails if too many commands are already
buffered.
// In such a case, the calling function could wait and then add
again these commands.
espeak_ng_STATUS fifo_add_commands(t_espeak_command *c1,
t_espeak_command *c2);

// The current running command must be stopped and the awaiting
commands are cleared.
espeak_ng_STATUS fifo_stop(void);

// Is there a running command?
// Returns 1 if yes; 0 otherwise.
int fifo_is_busy(void);

// Terminate the fifo component.
// Last function to be called.
void fifo_terminate(void);

// Indicates if the running command is still enabled.
//
// Note: this function is mainly called by the SynthCallback
(speak_lib.cpp)
// It indicates if the actual wave sample can still be played. It
is helpful for
// stopping speech as soon as a cancel command is applied.
//
// Returns 1 if yes, or 0 otherwise.
int fifo_is_command_enabled(void);

#ifdef __cplusplus
```

```
}
#endif

#endif
```

Chapter 17

./src/libespeak-ng/compiledict.h

```
#ifndef ESPEAK_NG_COMPILEDICT_H
#define ESPEAK_NG_COMPILEDICT_H

#ifdef __cplusplus
extern "C"
{
#endif

char *DecodeRule(const char *group_chars,
  int group_length,
  char *rule,
  int control);

void print_dictionary_flags(unsigned int *flags,
  char *buf,
  int buf_len);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 18

./src/libespeak-ng/ieee80.h

```
#ifndef IEEE_H
#define IEEE_H

#ifndef applec
 typedef double defdouble;
#else applec
 typedef long double defdouble;
#endif applec

defdouble ConvertFromIeeeSingle(  char *bytes);
defdouble ConvertFromIeeeDouble(  char *bytes);
defdouble ConvertFromIeeeExtended(char *bytes);

void ConvertToIeeeSingle(  defdouble num, char *bytes);
void ConvertToIeeeDouble(  defdouble num, char *bytes);
void ConvertToIeeeExtended(defdouble num, char *bytes);

#endif
```

Chapter 19

./src/libespeak-ng/espeak_command.h

```
#ifndef ESPEAK_NG_COMMAND_H
#define ESPEAK_NG_COMMAND_H

#include <espeak-ng/espeak_ng.h>

#ifdef __cplusplus
extern "C"
{
#endif

typedef enum {
 ET_TEXT,
 ET_MARK,
 ET_KEY,
 ET_CHAR,
 ET_PARAMETER,
 ET_PUNCTUATION_LIST,
 ET_VOICE_NAME,
 ET_VOICE_SPEC,
 ET_TERMINATED_MSG
```

```
} t_espeak_type;

typedef struct {
 unsigned int unique_identifier;
 void *text;
 unsigned int position;
 espeak_POSITION_TYPE position_type;
 unsigned int end_position;
 unsigned int flags;
 void *user_data;
} t_espeak_text;

typedef struct {
 unsigned int unique_identifier;
 void *text;
 const char *index_mark;
 unsigned int end_position;
 unsigned int flags;
 void *user_data;
} t_espeak_mark;

typedef struct {
 unsigned int unique_identifier;
 void *user_data;
 wchar_t character;
} t_espeak_character;

typedef struct {
 unsigned int unique_identifier;
 void *user_data;
 const char *key_name;
} t_espeak_key;

typedef struct {
 unsigned int unique_identifier;
 void *user_data;
} t_espeak_terminated_msg;
```

```
typedef struct {
 espeak_PARAMETER parameter;
 int value;
 int relative;
} t_espeak_parameter;

typedef enum {
 CS_UNDEFINED, // The command has just been created
 CS_PENDING,   // stored in the fifo
 CS_PROCESSED  // processed
} t_command_state;

typedef struct {
 t_espeak_type type;
 t_command_state state;

 union command {
  t_espeak_text my_text;
  t_espeak_mark my_mark;
  t_espeak_key my_key;
  t_espeak_character my_char;
  t_espeak_parameter my_param;
  const wchar_t *my_punctuation_list;
  const char *my_voice_name;
  espeak_VOICE my_voice_spec;
  t_espeak_terminated_msg my_terminated_msg;
 } u;
} t_espeak_command;

t_espeak_command *create_espeak_text(const void *text, size_t
size, unsigned int position, espeak_POSITION_TYPE position_type,
unsigned int end_position, unsigned int flags, void *user_data);

t_espeak_command *create_espeak_mark(const void *text, size_t
size, const char *index_mark, unsigned int end_position, unsigned
int flags, void *user_data);
```

```
t_espeak_command *create_espeak_terminated_msg(unsigned int
unique_identifier, void *user_data);

t_espeak_command *create_espeak_key(const char *key_name, void
*user_data);

t_espeak_command *create_espeak_char(wchar_t character, void
*user_data);

t_espeak_command *create_espeak_parameter(espeak_PARAMETER
parameter, int value, int relative);

t_espeak_command *create_espeak_punctuation_list(const wchar_t
*punctlist);

t_espeak_command *create_espeak_voice_name(const char *name);

t_espeak_command *create_espeak_voice_spec(espeak_VOICE
*voice_spec);

void process_espeak_command(t_espeak_command *the_command);

int delete_espeak_command(t_espeak_command *the_command);

espeak_ng_STATUS sync_espeak_Synth(unsigned int
unique_identifier, const void *text,
                                   unsigned int position,
espeak_POSITION_TYPE position_type,
                                   unsigned int end_position,
unsigned int flags, void *user_data);
espeak_ng_STATUS sync_espeak_Synth_Mark(unsigned int
unique_identifier, const void *text,
                                        const char *index_mark,
unsigned int end_position,
                                        unsigned int flags, void
*user_data);
```

```
espeak_ng_STATUS sync_espeak_Key(const char *key);
espeak_ng_STATUS sync_espeak_Char(wchar_t character);
void sync_espeak_SetPunctuationList(const wchar_t *punctlist);
void sync_espeak_SetParameter(espeak_PARAMETER parameter, int
value, int relative);
espeak_ng_STATUS SetParameter(int parameter, int value, int
relative);

int sync_espeak_terminated_msg(unsigned int unique_identifier,
void *user_data);

#ifdef __cplusplus
}
#endif

// >
#endif
```

Chapter 20

./src/libespeak-ng/phoneme.h

```
#ifndef ESPEAK_NG_PHONEME_H
#define ESPEAK_NG_PHONEME_H

#include <espeak-ng/espeak_ng.h>

#ifdef __cplusplus
extern "C"
{
#endif

// See docs/phonemes.md for the list of supported features.
typedef enum {
# define FEATURE_T(a, b, c) ((a << 16) | (b << 8) | (c))
 // invalid phoneme feature name
 inv = 0,
 // manner of articulation
 nas = FEATURE_T('n', 'a', 's'),
 stp = FEATURE_T('s', 't', 'p'),
 afr = FEATURE_T('a', 'f', 'r'),
 frc = FEATURE_T('f', 'r', 'c'),
 flp = FEATURE_T('f', 'l', 'p'),
 trl = FEATURE_T('t', 'r', 'l'),
```

```
apr = FEATURE_T('a', 'p', 'r'),
clk = FEATURE_T('c', 'l', 'k'),
ejc = FEATURE_T('e', 'j', 'c'),
imp = FEATURE_T('i', 'm', 'p'),
vwl = FEATURE_T('v', 'w', 'l'),
lat = FEATURE_T('l', 'a', 't'),
sib = FEATURE_T('s', 'i', 'b'),
// place of articulation
blb = FEATURE_T('b', 'l', 'b'),
lbd = FEATURE_T('l', 'b', 'd'),
bld = FEATURE_T('b', 'l', 'd'),
dnt = FEATURE_T('d', 'n', 't'),
alv = FEATURE_T('a', 'l', 'v'),
pla = FEATURE_T('p', 'l', 'a'),
rfx = FEATURE_T('r', 'f', 'x'),
alp = FEATURE_T('a', 'l', 'p'),
pal = FEATURE_T('p', 'a', 'l'),
vel = FEATURE_T('v', 'e', 'l'),
lbv = FEATURE_T('l', 'b', 'v'),
uvl = FEATURE_T('u', 'v', 'l'),
phr = FEATURE_T('p', 'h', 'r'),
glt = FEATURE_T('g', 'l', 't'),
// voice
vcd = FEATURE_T('v', 'c', 'd'),
vls = FEATURE_T('v', 'l', 's'),
// vowel height
hgh = FEATURE_T('h', 'g', 'h'),
smh = FEATURE_T('s', 'm', 'h'),
umd = FEATURE_T('u', 'm', 'd'),
mid = FEATURE_T('m', 'i', 'd'),
lmd = FEATURE_T('l', 'm', 'd'),
sml = FEATURE_T('s', 'm', 'l'),
low = FEATURE_T('l', 'o', 'w'),
// vowel backness
fnt = FEATURE_T('f', 'n', 't'),
cnt = FEATURE_T('c', 'n', 't'),
bck = FEATURE_T('b', 'c', 'k'),
```

```
// rounding
unr = FEATURE_T('u', 'n', 'r'),
rnd = FEATURE_T('r', 'n', 'd'),
// articulation
lgl = FEATURE_T('l', 'g', 'l'),
idt = FEATURE_T('i', 'd', 't'),
apc = FEATURE_T('a', 'p', 'c'),
lmn = FEATURE_T('l', 'm', 'n'),
// air flow
egs = FEATURE_T('e', 'g', 's'),
igs = FEATURE_T('i', 'g', 's'),
// phonation
brv = FEATURE_T('b', 'r', 'v'),
slv = FEATURE_T('s', 'l', 'v'),
stv = FEATURE_T('s', 't', 'v'),
crv = FEATURE_T('c', 'r', 'v'),
glc = FEATURE_T('g', 'l', 'c'),
// rounding and labialization
ptr = FEATURE_T('p', 't', 'r'),
cmp = FEATURE_T('c', 'm', 'p'),
mrd = FEATURE_T('m', 'r', 'd'),
lrd = FEATURE_T('l', 'r', 'd'),
// syllabicity
syl = FEATURE_T('s', 'y', 'l'),
nsy = FEATURE_T('n', 's', 'y'),
// consonant release
asp = FEATURE_T('a', 's', 'p'),
nrs = FEATURE_T('n', 'r', 's'),
lrs = FEATURE_T('l', 'r', 's'),
unx = FEATURE_T('u', 'n', 'x'),
// coarticulation
pzd = FEATURE_T('p', 'z', 'd'),
vzd = FEATURE_T('v', 'z', 'd'),
fzd = FEATURE_T('f', 'z', 'd'),
nzd = FEATURE_T('n', 'z', 'd'),
rzd = FEATURE_T('r', 'z', 'd'),
// tongue root
```

```
 atr = FEATURE_T('a', 't', 'r'),
 rtr = FEATURE_T('r', 't', 'r'),
 // fortis and lenis
 fts = FEATURE_T('f', 't', 's'),
 lns = FEATURE_T('l', 'n', 's'),
 // length
 est = FEATURE_T('e', 's', 't'),
 hlg = FEATURE_T('h', 'l', 'g'),
 lng = FEATURE_T('l', 'n', 'g'),
 elg = FEATURE_T('e', 'l', 'g'),
# undef FEATURE_T
} phoneme_feature_t;

phoneme_feature_t phoneme_feature_from_string(const char
*feature);

// phoneme types
#define phPAUSE   0
#define phSTRESS  1
#define phVOWEL   2
#define phLIQUID  3
#define phSTOP    4
#define phVSTOP   5
#define phFRICATIVE 6
#define phVFRICATIVE 7
#define phNASAL   8
#define phVIRTUAL 9
#define phDELETED 14
#define phINVALID 15

// places of articulation (phARTICULATION)
#define phPLACE_BILABIAL 1
#define phPLACE_LABIODENTAL 2
#define phPLACE_DENTAL 3
#define phPLACE_ALVEOLAR 4
#define phPLACE_RETROFLEX 5
#define phPLACE_PALATO_ALVEOLAR 6
```

```
#define phPLACE_PALATAL 7
#define phPLACE_VELAR 8
#define phPLACE_LABIO_VELAR 9
#define phPLACE_UVULAR 10
#define phPLACE_PHARYNGEAL 11
#define phPLACE_GLOTTAL 12

// phflags
#define phFLAGBIT_UNSTRESSED 1
#define phFLAGBIT_VOICELESS 3
#define phFLAGBIT_VOICED 4
#define phFLAGBIT_SIBILANT 5
#define phFLAGBIT_NOLINK 6
#define phFLAGBIT_TRILL 7
#define phFLAGBIT_PALATAL 9
#define phFLAGBIT_BRKAFTER 14 // [*] add a post-pause
#define phARTICULATION 0xf0000 // bits 16-19
#define phFLAGBIT_NONSYLLABIC 20 // don't count this vowel as a
syllable when finding the stress position
#define phFLAGBIT_LONG 21
#define phFLAGBIT_LENGTHENSTOP 22 // make the pre-pause slightly
longer
#define phFLAGBIT_RHOTIC 23
#define phFLAGBIT_NOPAUSE 24
#define phFLAGBIT_PREVOICE 25 // for voiced stops
#define phFLAGBIT_FLAG1 28
#define phFLAGBIT_FLAG2 29
#define phFLAGBIT_LOCAL 31 // used during compilation

// phoneme properties
#define phUNSTRESSED   (1U << phFLAGBIT_UNSTRESSED)
#define phVOICELESS    (1U << phFLAGBIT_VOICELESS)
#define phVOICED       (1U << phFLAGBIT_VOICED)
#define phSIBILANT     (1U << phFLAGBIT_SIBILANT)
#define phNOLINK       (1U << phFLAGBIT_NOLINK)
#define phTRILL        (1U << phFLAGBIT_TRILL)
#define phPALATAL      (1U << phFLAGBIT_PALATAL)
```

```
#define phBRKAFTER     (1U << phFLAGBIT_BRKAFTER)
#define phNONSYLLABIC  (1U << phFLAGBIT_NONSYLLABIC)
#define phLONG         (1U << phFLAGBIT_LONG)
#define phLENGTHENSTOP (1U << phFLAGBIT_LENGTHENSTOP)
#define phRHOTIC       (1U << phFLAGBIT_RHOTIC)
#define phNOPAUSE      (1U << phFLAGBIT_NOPAUSE)
#define phPREVOICE     (1U << phFLAGBIT_PREVOICE)
#define phFLAG1        (1U << phFLAGBIT_FLAG1)
#define phFLAG2        (1U << phFLAGBIT_FLAG2)
#define phLOCAL        (1U << phFLAGBIT_LOCAL)

// fixed phoneme code numbers, these can be used from the program
code
#define phonCONTROL     1
#define phonSTRESS_U    2
#define phonSTRESS_D    3
#define phonSTRESS_2    4
#define phonSTRESS_3    5
#define phonSTRESS_P    6
#define phonSTRESS_P2   7    // priority stress within a word
#define phonSTRESS_PREV 8
#define phonPAUSE       9
#define phonPAUSE_SHORT 10
#define phonPAUSE_NOLINK 11
#define phonLENGTHEN    12
#define phonSCHWA       13
#define phonSCHWA_SHORT 14
#define phonEND_WORD    15
#define phonDEFAULTTONE 17
#define phonCAPITAL     18
#define phonGLOTTALSTOP 19
#define phonSYLLABIC    20
#define phonSWITCH      21
#define phonX1          22      // a language specific action
#define phonPAUSE_VSHORT 23
#define phonPAUSE_LONG  24
#define phonT_REDUCED   25
```

```
#define phonSTRESS_TONIC 26
#define phonPAUSE_CLAUSE 27
#define phonVOWELTYPES   28  // 28 to 33

#define N_PHONEME_TABS     150     // number of phoneme tables
#define N_PHONEME_TAB      256     // max phonemes in a phoneme
table
#define N_PHONEME_TAB_NAME  32     // must be multiple of 4

// main table of phonemes, index by phoneme number (1-254)

typedef struct {
 unsigned int mnemonic;       // Up to 4 characters.  The first
char is in the l.s.byte
 unsigned int phflags;        // bits 16-19 place of articulation
 unsigned short program;      // index into phondata file
 unsigned char code;          // the phoneme number
 unsigned char type;          // phVOWEL, phPAUSE, phSTOP etc
 unsigned char start_type;
 unsigned char end_type;      // vowels: endtype; consonant:
voicing switch
 unsigned char std_length;    // for vowels, in mS/2;  for
phSTRESS phonemes, this is the stress/tone type
 unsigned char length_mod;    // a length_mod group number, used
to access length_mod_tab
} PHONEME_TAB;

espeak_ng_STATUS
phoneme_add_feature(PHONEME_TAB *phoneme,
                    phoneme_feature_t feature);

// Several phoneme tables may be loaded into memory. phoneme_tab
points to
// one for the current voice
extern int n_phoneme_tab;
extern int current_phoneme_table;
extern PHONEME_TAB *phoneme_tab[N_PHONEME_TAB];
```

```
extern unsigned char phoneme_tab_flags[N_PHONEME_TAB];  // bit 0:
not inherited

typedef struct {
 char name[N_PHONEME_TAB_NAME];
 PHONEME_TAB *phoneme_tab_ptr;
 int n_phonemes;
 int includes;            // also include the phonemes from this
other phoneme table
} PHONEME_TAB_LIST;

// table of phonemes to be replaced with different phonemes, for
the current voice
#define N_REPLACE_PHONEMES   60
typedef struct {
 unsigned char old_ph;
 unsigned char new_ph;
 char type;   // 0=always replace, 1=only at end of word
} REPLACE_PHONEMES;

extern int n_replace_phonemes;
extern REPLACE_PHONEMES replace_phonemes[N_REPLACE_PHONEMES];

// Table of phoneme programs and lengths.  Used by MakeVowelLists
typedef struct {
 unsigned int addr;
 unsigned int length;
} PHONEME_PROG_LOG;

#define PH(c1, c2) (c2<<8)+c1          // combine two characters
into an integer for phoneme name
#define PH3(c1, c2, c3) (c3<<16)+(c2<<8)+c1
#define PhonemeCode2(c1, c2) PhonemeCode((c2<<8)+c1)

extern PHONEME_TAB_LIST phoneme_tab_list[N_PHONEME_TABS];
extern int phoneme_tab_number;
```

```
#ifdef __cplusplus
}
#endif

#endif
```

Chapter 21

./src/libespeak-ng/numbers.h

```
#ifndef ESPEAK_NG_NUMBERS_H
#define ESPEAK_NG_NUMBERS_H

#include "translate.h"

#ifdef __cplusplus
extern "C"
{
#endif

void LookupAccentedLetter(Translator *tr, unsigned int letter,
char *ph_buf);
void LookupLetter(Translator *tr, unsigned int letter, int
next_byte, char *ph_buf1, int control);
int IsSuperscript(int letter);
void SetSpellingStress(Translator *tr, char *phonemes, int
control, int n_chars);
int TranslateRoman(Translator *tr, char *word, char *ph_out,
WORD_TAB *wtab);
int TranslateNumber(Translator *tr, char *word1, char *ph_out,
unsigned int *flags, WORD_TAB *wtab, int control);
int TranslateLetter(Translator *tr, char *word, char *phonemes,
```

```
int control, ALPHABET *current_alphabet);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 22

./src/libespeak-ng/voice.h

```
#ifndef ESPEAK_NG_VOICE_H
#define ESPEAK_NG_VOICE_H

#include <espeak-ng/espeak_ng.h>

#ifdef __cplusplus
extern "C"
{
#endif

#define N_PEAKS   9

typedef struct {
 char v_name[40];
 char language_name[20];

 int phoneme_tab_ix; // phoneme table number
 int pitch_base; // Hz<<12
 int pitch_range; // standard = 0x1000

 int speedf1;
 int speedf2;
```

```
 int speedf3;

 int speed_percent;      // adjust the WPM speed by this
percentage
 int flutter;
 int roughness;
 int echo_delay;
 int echo_amp;
 int n_harmonic_peaks;  // highest formant which is formed from
adding harmonics
 int peak_shape;        // alternative shape for formant peaks
(0=standard 1=squarer)
 int voicing;           // 100% = 64, level of formant-
synthesized sound
 int formant_factor;    // adjust nominal formant frequencies by
this  because of the voice's pitch (256ths)
 int consonant_amp;     // amplitude of unvoiced consonants
 int consonant_ampv;    // amplitude of the noise component of
voiced consonants
 int samplerate;
 int klattv[8];

 // parameters used by Wavegen
 short freq[N_PEAKS];    // 100% = 256
 short height[N_PEAKS];  // 100% = 256
 short width[N_PEAKS];   // 100% = 256
 short freqadd[N_PEAKS]; // Hz

 // copies without temporary adjustments from embedded commands
 short freq2[N_PEAKS];    // 100% = 256
 short height2[N_PEAKS];  // 100% = 256
 short width2[N_PEAKS];   // 100% = 256

 int breath[N_PEAKS];  // amount of breath for each formant.
breath[0] indicates whether any are set.
 int breathw[N_PEAKS]; // width of each breath formant
```

```
 // This table provides the opportunity for tone control.
 // Adjustment of harmonic amplitudes, steps of 8Hz
 // value of 128 means no change
 #define N_TONE_ADJUST  1000
 unsigned char tone_adjust[N_TONE_ADJUST];   //  8Hz steps * 1000
= 8kHz

} voice_t;

extern espeak_VOICE current_voice_selected;

extern voice_t *voice;
extern int tone_points[12];

const char *SelectVoice(espeak_VOICE *voice_select, int *found);
espeak_VOICE *SelectVoiceByName(espeak_VOICE **voices, const char
*name);
voice_t *LoadVoice(const char *voice_name, int control);
voice_t *LoadVoiceVariant(const char *voice_name, int variant);
espeak_ng_STATUS DoVoiceChange(voice_t *v);
void WavegenSetVoice(voice_t *v);
void ReadTonePoints(char *string, int *tone_pts);
void VoiceReset(int control);
void FreeVoiceList(void);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 23

./src/libespeak-ng/speech.h

```
#ifndef ESPEAK_NG_SPEECH_H
#define ESPEAK_NG_SPEECH_H

#include <espeak-ng/espeak_ng.h>

#include "mbrola.h"

#ifdef __cplusplus
extern "C"
{
#endif

#if defined(BYTE_ORDER) && BYTE_ORDER == BIG_ENDIAN
#define ARCH_BIG
#endif

#ifdef __QNX__
#define NO_VARIADIC_MACROS
#endif

#if defined(_WIN32) || defined(_WIN64) // Windows
```

```
#define PLATFORM_WINDOWS
#define PATHSEP '\\'
#define N_PATH_HOME  230
#define NO_VARIADIC_MACROS

#else

#define PLATFORM_POSIX
#define PATHSEP  '/'
#define N_PATH_HOME  160
#define USE_NANOSLEEP
#define __cdecl

#endif

// will look for espeak_data directory here, and also in user's
home directory
#ifndef PATH_ESPEAK_DATA
   #define PATH_ESPEAK_DATA  "/usr/share/espeak-ng-data"
#endif

typedef struct {
 const char *mnem;
 int value;
} MNEM_TAB;
int LookupMnem(MNEM_TAB *table, const char *string);
const char *LookupMnemName(MNEM_TAB *table, const int value);

void cancel_audio(void);

extern char path_home[N_PATH_HOME];    // this is the espeak-ng-
data directory

extern ESPEAK_NG_API int GetFileLength(const char *filename);

#ifdef __cplusplus
}
```

```
#endif

#endif // SPEECH_H
```

Chapter 24

./src/libespeak-ng/spect.h

```
#ifndef ESPEAK_NG_SPECT_H
#define ESPEAK_NG_SPECT_H

#include <espeak-ng/espeak_ng.h>

#include "wavegen.h"

#include "synthesize.h"
#include "speech.h"

#ifdef __cplusplus
extern "C"
{
#endif

float polint(float xa[], float ya[], int n, float x);

#define FRAME_WIDTH  1000  // max width for 8000kHz frame
#define MAX_DISPLAY_FREQ 9500
#define FRAME_HEIGHT 240

#define T_ZOOMOUT  301
```

```
#define T_ZOOMIN   302
#define T_USEPITCHENV 303
#define T_SAMPRATE 304
#define T_PITCH1   305
#define T_PITCH2   306
#define T_DURATION 307
#define T_AMPLITUDE 308
#define T_AMPFRAME  309
#define T_TIMEFRAME 310
#define T_TIMESEQ   311

#define T_AV      312
#define T_AVP     313
#define T_FRIC    314
#define T_FRICBP  315
#define T_ASPR    316
#define T_TURB    317
#define T_SKEW    318
#define T_TILT    319
#define T_KOPEN   320
#define T_FNZ     321

#define FILEID1_SPECTSEQ 0x43455053
#define FILEID2_SPECTSEQ 0x51455354  // for eSpeak sequence
#define FILEID2_SPECTSEK 0x4b455354  // for Klatt sequence
#define FILEID2_SPECTSQ2 0x32515354  // with Klatt data

#define FILEID1_SPC2     0x32435053  // an old format for
spectrum files

#define FILEID1_PITCHENV 0x43544950
#define FILEID2_PITCHENV 0x564e4548

#define FILEID1_PRAATSEQ 0x41415250
#define FILEID2_PRAATSEQ 0x51455354

typedef struct {
```

```
 unsigned short pitch1;
 unsigned short pitch2;
 unsigned char env[128];
} PitchEnvelope;

typedef struct {
 short freq;
 short bandw;
} formant_t;

typedef struct {
 short pkfreq;
 short pkheight;
 short pkwidth;
 short pkright;
 short klt_bw;
 short klt_ap;
 short klt_bp;
} peak_t;

typedef struct {
 int keyframe;
 short amp_adjust;
 float length_adjust;
 double rms;

 float time;
 float pitch;
 float length;
 float dx;
 unsigned short nx;
 short markers;
 int max_y;
 unsigned short *spect; // sqrt of harmonic amplitudes,  1-nx at
'pitch'

 short klatt_param[N_KLATTP2];
```

```
 formant_t formants[N_PEAKS]; // this is just the estimate given
by Praat
 peak_t peaks[N_PEAKS];
} SpectFrame;

double GetFrameRms(SpectFrame *frame, int amp);

typedef struct {
 int numframes;
 short amplitude;
 int spare;
 char *name;

 SpectFrame **frames;
 PitchEnvelope pitchenv;
 int pitch1;
 int pitch2;
 int duration;
 int grid;
 int bass_reduction;
 int max_x;
 short max_y;
 int file_format;
} SpectSeq;

SpectSeq *SpectSeqCreate(void);
void SpectSeqDestroy(SpectSeq *spect);
espeak_ng_STATUS LoadSpectSeq(SpectSeq *spect, const char
*filename);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 25

./src/libespeak-ng/mbrola.h

```
// declarations for compilembrola.c and synth_mbrola.c

#ifndef ESPEAK_NG_MBROLA_H
#define ESPEAK_NG_MBROLA_H

#include <stdbool.h>

#include "synthesize.h"

#ifdef __cplusplus
extern "C"
{
#endif

typedef struct {
        int name;
        unsigned int next_phoneme;
        int mbr_name;
        int mbr_name2;
        int percent; // percentage length of first component
        int control;
} MBROLA_TAB;
```

```
extern int mbrola_delay;
extern char mbrola_name[20];

espeak_ng_STATUS LoadMbrolaTable(const char *mbrola_voice,
  const char *phtrans,
  int *srate);

int MbrolaGenerate(PHONEME_LIST *phoneme_list,
  int *n_ph, bool resume);

int MbrolaFill(int length,
  bool resume,
  int amplitude);

void MbrolaReset(void);
int MbrolaTranslate(PHONEME_LIST *plist, int n_phonemes, bool
resume, FILE *f_mbrola);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 26

./src/libespeak-ng/synthdata.h

```
#ifndef ESPEAK_NG_SYNTHDATA_H
#define ESPEAK_NG_SYNTHDATA_H

#include "synthesize.h"
#include "translate.h"

#ifdef __cplusplus
extern "C"
{
#endif

void InterpretPhoneme(Translator *tr,
  int control,
  PHONEME_LIST *plist,
  PHONEME_DATA *phdata,
  WORD_PH_DATA *worddata);

void InterpretPhoneme2(int phcode,
  PHONEME_DATA *phdata);

void FreePhData(void);
unsigned char *GetEnvelope(int index);
```

```
espeak_ng_STATUS LoadPhData(int *srate, espeak_ng_ERROR_CONTEXT
*context);
void LoadConfig(void);
int LookupPhonemeString(const char *string);
int LookupPhonemeTable(const char *name);
frameref_t *LookupSpect(PHONEME_TAB *this_ph,
  int which,
  FMT_PARAMS *fmt_params,
  int *n_frames,
  PHONEME_LIST *plist);

int NumInstnWords(unsigned short *prog);
int PhonemeCode(unsigned int mnem);
void SelectPhonemeTable(int number);
int  SelectPhonemeTableName(const char *name);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 27

./src/libespeak-ng/wavegen.h

```
#ifndef ESPEAK_NG_WAVEGEN_H
#define ESPEAK_NG_WAVEGEN_H

#include "voice.h"

#ifdef __cplusplus
extern "C"
{
#endif

typedef struct {
 int freq;     // Hz<<16
 int height;   // height<<15
 int left;     // Hz<<16
 int right;    // Hz<<16
 double freq1; // floating point versions of the above
 double height1;
 double left1;
 double right1;
 double freq_inc; // increment by this every 64 samples
 double height_inc;
 double left_inc;
```

```
 double right_inc;
} wavegen_peaks_t;

int GetAmplitude(void);
void InitBreath(void);
int PeaksToHarmspect(wavegen_peaks_t *peaks,
  int pitch,
  int *htab,
  int control);

void SetPitch2(voice_t *voice,
  int pitch1,
  int pitch2,
  int *pitch_base,
  int *pitch_range);

void WavegenInit(int rate,
  int wavemult_fact);

int WavegenFill(void);
void WavegenSetVoice(voice_t *v);
int WcmdqFree(void);
void WcmdqStop(void);
int WcmdqUsed(void);
void WcmdqInc(void);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 28

./src/libespeak-ng/sintab.h

```
#ifndef ESPEAK_NG_SINTAB_H
#define ESPEAK_NG_SINTAB_H

#ifdef __cplusplus
extern "C"
{
#endif

short int sin_tab[2048] = {
 0, -25, -50, -75, -100, -125, -150, -175,
 -201, -226, -251, -276, -301, -326, -351, -376,
 -401, -427, -452, -477, -502, -527, -552, -577,
 -602, -627, -652, -677, -702, -727, -752, -777,
 -802, -827, -852, -877, -902, -927, -952, -977,
 -1002, -1027, -1052, -1077, -1102, -1127, -1152, -1177,
 -1201, -1226, -1251, -1276, -1301, -1326, -1350, -1375,
 -1400, -1425, -1449, -1474, -1499, -1523, -1548, -1573,
 -1597, -1622, -1647, -1671, -1696, -1721, -1745, -1770,
 -1794, -1819, -1843, -1868, -1892, -1917, -1941, -1965,
 -1990, -2014, -2038, -2063, -2087, -2111, -2136, -2160,
 -2184, -2208, -2233, -2257, -2281, -2305, -2329, -2353,
 -2377, -2401, -2425, -2449, -2473, -2497, -2521, -2545,
```

```
 // similar lines removed from listning
};

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 29

./src/libespeak-ng/event.h

```
   Manage events (sentence, word, mark, end,...), is responsible
of calling the external
   callback as soon as the relevant audio sample is played.

   The audio stream is composed of samples from synthetised
messages or audio icons.
   Each event is associated to a sample.

   Scenario:

   - event_declare is called for each expected event.

   - A timeout is started for the first pending event.

   - When the timeout happens, the synth_callback is called.

   Note: the timeout is checked against the real progress of the
audio stream, which depends on pauses or underruns. If the real
progress is lower than the expected one, a new timeout starts.

#ifndef ESPEAK_NG_EVENT_H
#define ESPEAK_NG_EVENT_H
```

```c
#include <espeak-ng/espeak_ng.h>

#ifdef __cplusplus
extern "C"
{
#endif

// Initialize the event component.
// First function to be called.
// the callback will be called when the event actually occurs.
// The callback is detailled in speak_lib.h .
void event_init(void);
void event_set_callback(t_espeak_callback *cb);

// Clear any pending event.
espeak_ng_STATUS event_clear_all(void);

// Declare a future event
espeak_ng_STATUS event_declare(espeak_EVENT *event);

// Terminate the event component.
// Last function to be called.
void event_terminate(void);

// general functions
struct timespec;
void clock_gettime2(struct timespec *ts);
void add_time_in_ms(struct timespec *ts, int time_in_ms);

#ifdef __cplusplus
}
#endif

#endif
```

Chapter 30

./src/speak-ng.c

```
#define PROGRAM_NAME "speak-ng"
#define PLAYBACK_MODE (ENOUTPUT_MODE_SYNCHRONOUS |
ENOUTPUT_MODE_SPEAK_AUDIO)

#include "espeak-ng.c"
```

Chapter 31

./src/espeak-ng.c

```
#include "config.h"

#include <ctype.h>
#include <errno.h>
#include <getopt.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>

#ifndef PROGRAM_NAME
#define PROGRAM_NAME "espeak-ng"
#endif

#ifndef PLAYBACK_MODE
```

```
#define PLAYBACK_MODE ENOUTPUT_MODE_SPEAK_AUDIO
#endif

extern ESPEAK_NG_API void strncpy0(char *to, const char *from,
int size);
extern ESPEAK_NG_API int utf8_in(int *c, const char *buf);
extern ESPEAK_NG_API int GetFileLength(const char *filename);

static const char *help_text =
    "\n" PROGRAM_NAME " [options] [\"<words>\"]\n\n"
    "-f <text file>   Text file to speak\n"
    "--stdin    Read text input from stdin instead of a file\n\n"
    "If neither -f nor --stdin, then <words> are spoken, or if
none then text\n"
    "is spoken from stdin, each line separately.\n\n"
    "-a <integer>\n"
    "\t   Amplitude, 0 to 200, default is 100\n"
    "-d <device>\n"
    "\t   Use the specified device to speak the audio on. If not
specified, the\n"
    "\t   default audio device is used.\n"
    "-g <integer>\n"
    "\t   Word gap. Pause between words, units of 10mS at the
default speed\n"
    "-k <integer>\n"
    "\t   Indicate capital letters with: 1=sound, 2=the word
\"capitals\",\n"
    "\t   higher values indicate a pitch increase (try -k20).\n"
    "-l <integer>\n"
    "\t   Line length. If not zero (which is the default),
consider\n"
    "\t   lines less than this length as end-of-clause\n"
    "-p <integer>\n"
    "\t   Pitch adjustment, 0 to 99, default is 50\n"
    "-s <integer>\n"
    "\t   Speed in approximate words per minute. The default is
175\n"
```

```
    "-v <voice name>\n"
    "\t   Use voice file of this name from espeak-ng-
data/voices\n"
    "-w <wave file name>\n"
    "\t   Write speech to this WAV file, rather than speaking it
directly\n"
    "-b\t   Input text encoding, 1=UTF8, 2=8 bit, 4=16 bit \n"
    "-m\t   Interpret SSML markup, and ignore other < > tags\n"
    "-q\t   Quiet, don't produce any speech (may be useful with
-x)\n"
    "-x\t   Write phoneme mnemonics to stdout\n"
    "-X\t   Write phonemes mnemonics and translation trace to
stdout\n"
    "-z\t   No final sentence pause at the end of the text\n"
    "--compile=<voice name>\n"
    "\t   Compile pronunciation rules and dictionary from the
current\n"
    "\t   directory. <voice name> specifies the language\n"
    "--compile-debug=<voice name>\n"
    "\t   Compile pronunciation rules and dictionary from the
current\n"
    "\t   directory, including line numbers for use with -X.\n"
    "\t   <voice name> specifies the language\n"
    "--compile-mbrola=<voice name>\n"
    "\t   Compile an MBROLA voice\n"
    "--compile-intonations\n"
    "\t   Compile the intonation data\n"
    "--compile-phonemes=<phsource-dir>\n"
    "\t   Compile the phoneme data using <phsource-dir> or the
default phsource directory\n"
    "--ipa      Write phonemes to stdout using International
Phonetic Alphabet\n"
    "--path=\"<path>\"\n"
    "\t   Specifies the directory containing the espeak-ng-data
directory\n"
    "--pho      Write mbrola phoneme data (.pho) to stdout or to
the file in --phonout\n"
```

```
    "--phonout=\"<filename>\"\n"
    "\t   Write phoneme output from -x -X --ipa and --pho to this
file\n"
    "--punct=\"<characters>\"\n"
    "\t   Speak the names of punctuation characters during
speaking.  If\n"
    "\t   =<characters> is omitted, all punctuation is spoken.\n"
    "--sep=<character>\n"
    "\t   Separate phonemes (from -x --ipa) with <character>.\n"
    "\t   Default is space, z means ZWJN character.\n"
    "--split=<minutes>\n"
    "\t   Starts a new WAV file every <minutes>.  Used with -w\n"
    "--stdout   Write speech output to stdout\n"
    "--tie=<character>\n"
    "\t   Use a tie character within multi-letter phoneme
names.\n"
    "\t   Default is U+361, z means ZWJ character.\n"
    "--version  Shows version number and date, and location of
espeak-ng-data\n"
    "--voices=<language>\n"
    "\t   List the available voices for the specified
language.\n"
    "\t   If <language> is omitted, then list all voices.\n"
    "--load     Load voice from a file in current directory by
name.\n"
    "-h, --help Show this help.\n";

static int samplerate;
bool quiet = false;
unsigned int samples_total = 0;
unsigned int samples_split = 0;
unsigned int samples_split_seconds = 0;
unsigned int wavefile_count = 0;

FILE *f_wavfile = NULL;
char filetype[5];
char wavefile[200];
```

```
static void DisplayVoices(FILE *f_out, char *language)
{
 int ix;
 const char *p;
 int len;
 int count;
 int c;
 size_t j;
 const espeak_VOICE *v;
 const char *lang_name;
 char age_buf[12];
 char buf[80];
 const espeak_VOICE **voices;
 espeak_VOICE voice_select;

 static char genders[4] = { '-', 'M', 'F', '-' };

 if ((language != NULL) && (language[0] != 0)) {
  // display only voices for the specified language, in order of
priority
  voice_select.languages = language;
  voice_select.age = 0;
  voice_select.gender = 0;
  voice_select.name = NULL;
  voices = espeak_ListVoices(&voice_select);
 } else
  voices = espeak_ListVoices(NULL);

 fprintf(f_out, "Pty Language       Age/Gender VoiceName
File                 Other Languages\n");

 for (ix = 0; (v = voices[ix]) != NULL; ix++) {
  count = 0;
  p = v->languages;
  while (*p != 0) {
   len = strlen(p+1);
```

```
   lang_name = p+1;

   if (v->age == 0)
    strcpy(age_buf, " --");
   else
    sprintf(age_buf, "%3d", v->age);

   if (count == 0) {
    for (j = 0; j < sizeof(buf); j++) {
     // replace spaces in the name
     if ((c = v->name[j]) == ' ')
      c = '_';
     if ((buf[j] = c) == 0)
      break;
    }
    fprintf(f_out, "%2d  %-15s%s/%c      %-18s %-20s ",
            p[0], lang_name, age_buf, genders[v->gender], buf,
v->identifier);
   } else
    fprintf(f_out, "(%s %d)", lang_name, p[0]);
   count++;
   p += len+2;
  }
  fputc('\n', f_out);
 }
}

static void Write4Bytes(FILE *f, int value)
{
 // Write 4 bytes to a file, least significant first
 int ix;

 for (ix = 0; ix < 4; ix++) {
  fputc(value & 0xff, f);
  value = value >> 8;
 }
}
```

```
static int OpenWavFile(char *path, int rate)
{
 static unsigned char wave_hdr[44] = {
  'R', 'I', 'F', 'F', 0x24, 0xf0, 0xff, 0x7f, 'W', 'A', 'V', 'E',
'f', 'm', 't', ' ',
  0x10, 0, 0, 0, 1, 0, 1, 0,  9, 0x3d, 0, 0, 0x12, 0x7a, 0, 0,
  2, 0, 0x10, 0, 'd', 'a', 't', 'a',  0x00, 0xf0, 0xff, 0x7f
 };

 if (path == NULL)
  return 2;

 while (isspace(*path)) path++;

 f_wavfile = NULL;
 if (path[0] != 0) {
  if (strcmp(path, "stdout") == 0) {
#ifdef PLATFORM_WINDOWS
   // prevent Windows adding 0x0d before 0x0a bytes
   _setmode(_fileno(stdout), _O_BINARY);
#endif
   f_wavfile = stdout;
  } else
   f_wavfile = fopen(path, "wb");
 }

 if (f_wavfile == NULL) {
  fprintf(stderr, "Can't write to: '%s'\n", path);
  return 1;
 }

 fwrite(wave_hdr, 1, 24, f_wavfile);
 Write4Bytes(f_wavfile, rate);
 Write4Bytes(f_wavfile, rate * 2);
 fwrite(&wave_hdr[32], 1, 12, f_wavfile);
 return 0;
```

```
}

static void CloseWavFile()
{
 unsigned int pos;

 if ((f_wavfile == NULL) || (f_wavfile == stdout))
  return;

 fflush(f_wavfile);
 pos = ftell(f_wavfile);

 if (fseek(f_wavfile, 4, SEEK_SET) != -1)
  Write4Bytes(f_wavfile, pos - 8);

 if (fseek(f_wavfile, 40, SEEK_SET) != -1)
  Write4Bytes(f_wavfile, pos - 44);

 fclose(f_wavfile);
 f_wavfile = NULL;
}

static int SynthCallback(short *wav, int numsamples, espeak_EVENT
*events)
{
 char fname[210];

 if (quiet || wav == NULL) return 0;

 while (events->type != 0) {
  if (events->type == espeakEVENT_SAMPLERATE) {
   samplerate = events->id.number;
   samples_split = samples_split_seconds * samplerate;
  } else if (events->type == espeakEVENT_SENTENCE) {
   // start a new WAV file when the limit is reached, at this
sentence boundary
   if ((samples_split > 0) && (samples_total > samples_split)) {
```

```
    CloseWavFile();
    samples_total = 0;
    wavefile_count++;
   }
  }
  events++;
 }

 if (f_wavfile == NULL) {
  if (samples_split > 0) {
   sprintf(fname, "%s_%.2d%s", wavefile, wavefile_count+1,
filetype);
   if (OpenWavFile(fname, samplerate) != 0)
    return 1;
  } else if (OpenWavFile(wavefile, samplerate) != 0)
   return 1;
 }

 if (numsamples > 0) {
  samples_total += numsamples;
  fwrite(wav, numsamples*2, 1, f_wavfile);
 }
 return 0;
}

static void PrintVersion()
{
 const char *version;
 const char *path_data;
 espeak_Initialize(AUDIO_OUTPUT_SYNCHRONOUS, 0, NULL,
espeakINITIALIZE_DONT_EXIT);
 version = espeak_Info(&path_data);
 printf("eSpeak NG text-to-speech: %s  Data at: %s\n", version,
path_data);
}

int main(int argc, char **argv)
```

```
{
 static struct option long_options[] = {
  { "help",    no_argument,       0, 'h' },
  { "stdin",   no_argument,       0, 0x100 },
  { "compile-debug", optional_argument, 0, 0x101 },
  { "compile", optional_argument, 0, 0x102 },
  { "punct",   optional_argument, 0, 0x103 },
  { "voices",  optional_argument, 0, 0x104 },
  { "stdout",  no_argument,       0, 0x105 },
  { "split",   optional_argument, 0, 0x106 },
  { "path",    required_argument, 0, 0x107 },
  { "phonout", required_argument, 0, 0x108 },
  { "pho",     no_argument,       0, 0x109 },
  { "ipa",     optional_argument, 0, 0x10a },
  { "version", no_argument,       0, 0x10b },
  { "sep",     optional_argument, 0, 0x10c },
  { "tie",     optional_argument, 0, 0x10d },
  { "compile-mbrola", optional_argument, 0, 0x10e },
  { "compile-intonations", no_argument, 0, 0x10f },
  { "compile-phonemes", optional_argument, 0, 0x110 },
  { "load",    no_argument,       0, 0x111 },
  { 0, 0, 0, 0 }
 };

 FILE *f_text = NULL;
 char *p_text = NULL;
 FILE *f_phonemes_out = stdout;
 char *data_path = NULL; // use default path for espeak-ng-data

 int option_index = 0;
 int c;
 int ix;
 char *optarg2;
 int value;
 int flag_stdin = 0;
 int flag_compile = 0;
 int flag_load = 0;
```

```
 int filesize = 0;
 int synth_flags = espeakCHARS_AUTO | espeakPHONEMES |
espeakENDPAUSE;

 int volume = -1;
 int speed = -1;
 int pitch = -1;
 int wordgap = -1;
 int option_capitals = -1;
 int option_punctuation = -1;
 int phonemes_separator = 0;
 int phoneme_options = 0;
 int option_linelength = 0;
 int option_waveout = 0;

 espeak_VOICE voice_select;
 char filename[200];
 char voicename[40];
 char devicename[200];
 #define N_PUNCTLIST 100
 wchar_t option_punctlist[N_PUNCTLIST];

 voicename[0] = 0;
 wavefile[0] = 0;
 filename[0] = 0;
 devicename[0] = 0;
 option_punctlist[0] = 0;

 while (true) {
  c = getopt_long(argc, argv, "a:b:d:f:g:hk:l:mp:qs:v:w:xXz",
                  long_options, &option_index);

  // Detect the end of the options.
  if (c == -1)
   break;
  optarg2 = optarg;
```

```
switch (c)
{
case 'b':
 // input character encoding, 8bit, 16bit, UTF8
 if ((sscanf(optarg2, "%d", &value) == 1) && (value <= 4))
  synth_flags |= value;
 else
  synth_flags |= espeakCHARS_8BIT;
 break;
case 'd':
 strncpy0(devicename, optarg2, sizeof(devicename));
 break;
case 'h':
 printf("\n");
 PrintVersion();
 printf("%s", help_text);
 return 0;
case 'k':
 option_capitals = atoi(optarg2);
 break;
case 'x':
 phoneme_options |= espeakPHONEMES_SHOW;
 break;
case 'X':
 phoneme_options |= espeakPHONEMES_TRACE;
 break;
case 'm':
 synth_flags |= espeakSSML;
 break;
case 'p':
 pitch = atoi(optarg2);
 break;
case 'q':
 quiet = true;
 break;
case 'f':
 strncpy0(filename, optarg2, sizeof(filename));
```

```
 break;
case 'l':
 option_linelength = atoi(optarg2);
 break;
case 'a':
 volume = atoi(optarg2);
 break;
case 's':
 speed = atoi(optarg2);
 break;
case 'g':
 wordgap = atoi(optarg2);
 break;
case 'v':
 strncpy0(voicename, optarg2, sizeof(voicename));
 break;
case 'w':
 option_waveout = 1;
 strncpy0(wavefile, optarg2, sizeof(filename));
 break;
case 'z': // remove pause from the end of a sentence
 synth_flags &= ~espeakENDPAUSE;
 break;
case 0x100: // --stdin
 flag_stdin = 1;
 break;
case 0x105: // --stdout
 option_waveout = 1;
 strcpy(wavefile, "stdout");
 break;
case 0x101: // --compile-debug
case 0x102: // --compile
 if (optarg2 != NULL && *optarg2) {
  strncpy0(voicename, optarg2, sizeof(voicename));
  flag_compile = c;
  quiet = true;
  break;
```

```
   } else {
    fprintf(stderr, "Voice name to '%s' not specified.\n", c ==
0x101 ? "--compile-debug" : "--compile");
    exit(EXIT_FAILURE);
   }
  case 0x103: // --punct
   option_punctuation = 1;
   if (optarg2 != NULL) {
    ix = 0;
    while ((ix < N_PUNCTLIST) && ((option_punctlist[ix] =
optarg2[ix]) != 0)) ix++;
    option_punctlist[N_PUNCTLIST-1] = 0;
    option_punctuation = 2;
   }
   break;
  case 0x104: // --voices
   espeak_Initialize(AUDIO_OUTPUT_SYNCHRONOUS, 0, data_path, 0);
   DisplayVoices(stdout, optarg2);
   exit(0);
  case 0x106: // -- split
   if (optarg2 == NULL)
    samples_split_seconds = 30 * 60; // default 30 minutes
   else
    samples_split_seconds = atoi(optarg2) * 60;
   break;
  case 0x107: // --path
   data_path = optarg2;
   break;
  case 0x108: // --phonout
   if ((f_phonemes_out = fopen(optarg2, "w")) == NULL)
    fprintf(stderr, "Can't write to: %s\n", optarg2);
   break;
  case 0x109: // --pho
   phoneme_options |= espeakPHONEMES_MBROLA;
   break;
  case 0x10a: // --ipa
   phoneme_options |= espeakPHONEMES_IPA;
```

```
   if (optarg2 != NULL) {
    // deprecated and obsolete
    switch (atoi(optarg2))
    {
    case 1:
     phonemes_separator = '_';
     break;
    case 2:
     phonemes_separator = 0x0361;
     phoneme_options |= espeakPHONEMES_TIE;
     break;
    case 3:
     phonemes_separator = 0x200d; // ZWJ
     phoneme_options |= espeakPHONEMES_TIE;
     break;
    }

   }
   break;
  case 0x10b: // --version
   PrintVersion();
   exit(0);
  case 0x10c: // --sep
   phoneme_options |= espeakPHONEMES_SHOW;
   if (optarg2 == 0)
    phonemes_separator = ' ';
   else
    utf8_in(&phonemes_separator, optarg2);
   if (phonemes_separator == 'z')
    phonemes_separator = 0x200c; // ZWNJ
   break;
  case 0x10d: // --tie
   phoneme_options |= (espeakPHONEMES_SHOW | espeakPHONEMES_TIE);
   if (optarg2 == 0)
    phonemes_separator = 0x0361; // default: combining-double-
inverted-breve
   else
```

```
    utf8_in(&phonemes_separator, optarg2);
   if (phonemes_separator == 'z')
    phonemes_separator = 0x200d; // ZWJ
   break;
  case 0x10e: // --compile-mbrola
  {
   espeak_ng_InitializePath(data_path);
   espeak_ng_ERROR_CONTEXT context = NULL;
   espeak_ng_STATUS result =
espeak_ng_CompileMbrolaVoice(optarg2, stdout, &context);
   if (result != ENS_OK) {
    espeak_ng_PrintStatusCodeMessage(result, stderr, context);
    espeak_ng_ClearErrorContext(&context);
    return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
  }
  case 0x10f: // --compile-intonations
  {
   espeak_ng_InitializePath(data_path);
   espeak_ng_ERROR_CONTEXT context = NULL;
   espeak_ng_STATUS result = espeak_ng_CompileIntonation(stdout,
&context);
   if (result != ENS_OK) {
    espeak_ng_PrintStatusCodeMessage(result, stderr, context);
    espeak_ng_ClearErrorContext(&context);
    return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
  }
  case 0x110: // --compile-phonemes
  {
   espeak_ng_InitializePath(data_path);
   espeak_ng_ERROR_CONTEXT context = NULL;
   espeak_ng_STATUS result;
   if (optarg2) {
    result = espeak_ng_CompilePhonemeDataPath(22050, optarg2,
```

```
NULL, stdout, &context);
   } else {
    result = espeak_ng_CompilePhonemeData(22050, stdout,
&context);
   }
   if (result != ENS_OK) {
    espeak_ng_PrintStatusCodeMessage(result, stderr, context);
    espeak_ng_ClearErrorContext(&context);
    return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
  }
  case 0x111: // --load
   flag_load = 1;
   break;
  default:
   exit(0);
  }
 }

 espeak_ng_InitializePath(data_path);
 espeak_ng_ERROR_CONTEXT context = NULL;
 espeak_ng_STATUS result = espeak_ng_Initialize(&context);
 if (result != ENS_OK) {
  espeak_ng_PrintStatusCodeMessage(result, stderr, context);
  espeak_ng_ClearErrorContext(&context);
  exit(1);
 }

 if (option_waveout || quiet) {
  // writing to a file (or no output), we can use synchronous
mode
  result = espeak_ng_InitializeOutput(ENOUTPUT_MODE_SYNCHRONOUS,
0, devicename[0] ? devicename : NULL);
  samplerate = espeak_ng_GetSampleRate();
  samples_split = samplerate * samples_split_seconds;
```

```
  espeak_SetSynthCallback(SynthCallback);
  if (samples_split) {
   char *extn;
   extn = strrchr(wavefile, '.');
   if ((extn != NULL) && ((wavefile + strlen(wavefile) - extn) <=
4)) {
    strcpy(filetype, extn);
    *extn = 0;
   }
  }
 } else {
  // play the sound output
  result = espeak_ng_InitializeOutput(PLAYBACK_MODE, 0,
devicename[0] ? devicename : NULL);
  samplerate = espeak_ng_GetSampleRate();
 }

 if (result != ENS_OK) {
  espeak_ng_PrintStatusCodeMessage(result, stderr, NULL);
  exit(EXIT_FAILURE);
 }

 if (voicename[0] == 0)
  strcpy(voicename, ESPEAKNG_DEFAULT_VOICE);

 if(flag_load)
  result = espeak_ng_SetVoiceByFile(voicename);
 else
  result = espeak_ng_SetVoiceByName(voicename);
 if (result != ENS_OK) {
  memset(&voice_select, 0, sizeof(voice_select));
  voice_select.languages = voicename;
  result = espeak_ng_SetVoiceByProperties(&voice_select);
  if (result != ENS_OK) {
   espeak_ng_PrintStatusCodeMessage(result, stderr, NULL);
   exit(EXIT_FAILURE);
  }
```

```
 }

 if (flag_compile) {
  // This must be done after the voice is set
  espeak_ng_ERROR_CONTEXT context = NULL;
  espeak_ng_STATUS result = espeak_ng_CompileDictionary("", NULL,
stderr, flag_compile & 0x1, &context);
  if (result != ENS_OK) {
   espeak_ng_PrintStatusCodeMessage(result, stderr, context);
   espeak_ng_ClearErrorContext(&context);
   return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
 }

 // set any non-default values of parameters. This must be done
after espeak_Initialize()
 if (speed > 0)
  espeak_SetParameter(espeakRATE, speed, 0);
 if (volume >= 0)
  espeak_SetParameter(espeakVOLUME, volume, 0);
 if (pitch >= 0)
  espeak_SetParameter(espeakPITCH, pitch, 0);
 if (option_capitals >= 0)
  espeak_SetParameter(espeakCAPITALS, option_capitals, 0);
 if (option_punctuation >= 0)
  espeak_SetParameter(espeakPUNCTUATION, option_punctuation, 0);
 if (wordgap >= 0)
  espeak_SetParameter(espeakWORDGAP, wordgap, 0);
 if (option_linelength > 0)
  espeak_SetParameter(espeakLINELENGTH, option_linelength, 0);
 if (option_punctuation == 2)
  espeak_SetPunctuationList(option_punctlist);

 espeak_SetPhonemeTrace(phoneme_options | (phonemes_separator <<
8), f_phonemes_out);
```

```
 if (filename[0] == 0) {
  if ((optind < argc) && (flag_stdin == 0)) {
   // there's a non-option parameter, and no -f or --stdin
   // use it as text
   p_text = argv[optind];
  } else {
   f_text = stdin;
   if (flag_stdin == 0)
    flag_stdin = 2;
  }
 } else {
  struct stat st;
  if (stat(filename, &st) != 0) {
   fprintf(stderr, "Failed to stat() file '%s'\n", filename);
   exit(EXIT_FAILURE);
  }
  filesize = GetFileLength(filename);
  f_text = fopen(filename, "r");
  if (f_text == NULL) {
   fprintf(stderr, "Failed to read file '%s'\n", filename);
   exit(EXIT_FAILURE);
  }
  if (S_ISFIFO(st.st_mode)) {
   flag_stdin = 2;
  }
 }

 if (p_text != NULL) {
  int size;
  size = strlen(p_text);
  espeak_Synth(p_text, size+1, 0, POS_CHARACTER, 0, synth_flags,
NULL, NULL);
 } else if (flag_stdin) {
  size_t max = 1000;
  if ((p_text = (char *)malloc(max)) == NULL) {
   espeak_ng_PrintStatusCodeMessage(ENOMEM, stderr, NULL);
   exit(EXIT_FAILURE);
```

```
  }

  if (flag_stdin == 2) {
   // line by line input on stdin or from FIFO
   while (fgets(p_text, max, f_text) != NULL) {
    p_text[max-1] = 0;
    espeak_Synth(p_text, max, 0, POS_CHARACTER, 0, synth_flags,
NULL, NULL);
    // Allow subprocesses to use the audio data through pipes.
    fflush(stdout);
   }
   if (f_text != stdin) {
    fclose(f_text);
   }
  } else {
   // bulk input on stdin
   ix = 0;
   while (true) {
    if ((c = fgetc(stdin)) == EOF)
     break;
    p_text[ix++] = (char)c;
    if (ix >= (max-1)) {
     char *new_text = NULL;
     if (max <= SIZE_MAX - 1000) {
      max += 1000;
      new_text = (char *)realloc(p_text, max);
     }
     if (new_text == NULL) {
      free(p_text);
      espeak_ng_PrintStatusCodeMessage(ENOMEM, stderr, NULL);
      exit(EXIT_FAILURE);
     }
     p_text = new_text;
    }
   }
   if (ix > 0) {
    p_text[ix-1] = 0;
```

```
    espeak_Synth(p_text, ix+1, 0, POS_CHARACTER, 0, synth_flags,
NULL, NULL);
   }
  }

  free(p_text);
 } else if (f_text != NULL) {
  if ((p_text = (char *)malloc(filesize+1)) == NULL) {
   espeak_ng_PrintStatusCodeMessage(ENOMEM, stderr, NULL);
   exit(EXIT_FAILURE);
  }

  fread(p_text, 1, filesize, f_text);
  p_text[filesize] = 0;
  espeak_Synth(p_text, filesize+1, 0, POS_CHARACTER, 0,
synth_flags, NULL, NULL);
  fclose(f_text);

  free(p_text);
 }

 result = espeak_ng_Synchronize();
 if (result != ENS_OK) {
  espeak_ng_PrintStatusCodeMessage(result, stderr, NULL);
  exit(EXIT_FAILURE);
 }

 if (f_phonemes_out != stdout)
  fclose(f_phonemes_out);

 CloseWavFile();
 espeak_ng_Terminate();
 return 0;
}
```

Chapter 32

./src/compat/getopt.c

```
#include "config.h"

#include <assert.h>
#include <errno.h>
#include <getopt.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define REPLACE_GETOPT

#define _DIAGASSERT(x) assert(x)
#define __UNCONST(x) (char *)(x)

#ifdef REPLACE_GETOPT
#ifdef __weak_alias
__weak_alias(getopt,_getopt)
#endif
int opterr = 1;  /* if error message should be printed */
int optind = 1;  /* index into parent argv vector */
int optopt = '?';  /* character checked for validity */
int optreset;  /* reset getopt */
```

```
char    *optarg;  /* argument associated with option */
#elif HAVE_NBTOOL_CONFIG_H && !HAVE_DECL_OPTRESET
static int optreset;
#endif

#ifdef __weak_alias
__weak_alias(getopt_long,_getopt_long)
#endif

#define IGNORE_FIRST (*options == '-' || *options == '+')
#define PRINT_ERROR ((opterr) && ((*options != ':') \
          || (IGNORE_FIRST && options[1] != ':')))
#define IS_POSIXLY_CORRECT (getenv("POSIXLY_CORRECT") != NULL)
#define PERMUTE         (!IS_POSIXLY_CORRECT && !IGNORE_FIRST)

#define IN_ORDER        (!IS_POSIXLY_CORRECT && *options == '-')

#define BADCH (int)'?'
#define BADARG  ((IGNORE_FIRST && options[1] == ':') \
    || (*options == ':') ? (int)':' : (int)'?')
#define INORDER (int)1

#define EMSG ""

static int getopt_internal(int, char **, const char *);
static int gcd(int, int);
static void permute_args(int, int, int, char **);

static const char *place = EMSG; /* option letter processing */

static int nonopt_start = -1; /* first non option argument (for
permute) */
static int nonopt_end = -1;   /* first option after non options
(for permute) */

static const char recargchar[] = "option requires an argument --
%c";
```

```
static const char recargstring[] = "option requires an argument
-- %s";
static const char ambig[] = "ambiguous option -- %.*s";
static const char noarg[] = "option doesn't take an argument --
%.*s";
static const char illoptchar[] = "unknown option -- %c";
static const char illoptstring[] = "unknown option -- %s";

static int
gcd(int a, int b)
{
 int c;

 c = a % b;
 while (c != 0) {
  a = b;
  b = c;
  c = a % b;
 }

 return b;
}

static void
permute_args(int panonopt_start, int panonopt_end, int opt_end,
char **nargv)
{
 int cstart, cyclelen, i, j, ncycle, nnonopts, nopts, pos;
 char *swap;

 _DIAGASSERT(nargv != NULL);

 /*
  * compute lengths of blocks and number and size of cycles
  */
 nnonopts = panonopt_end - panonopt_start;
 nopts = opt_end - panonopt_end;
```

```
 ncycle = gcd(nnonopts, nopts);
 cyclelen = (opt_end - panonopt_start) / ncycle;

 for (i = 0; i < ncycle; i++) {
  cstart = panonopt_end+i;
  pos = cstart;
  for (j = 0; j < cyclelen; j++) {
   if (pos >= panonopt_end)
    pos -= nnonopts;
   else
    pos += nopts;
   swap = nargv[pos];
   nargv[pos] = nargv[cstart];
   nargv[cstart] = swap;
  }
 }
}

static int
getopt_internal(int nargc, char **nargv, const char *options)
{
 char *oli;    /* option letter list index */
 int optchar;

 _DIAGASSERT(nargv != NULL);
 _DIAGASSERT(options != NULL);

 optarg = NULL;

 /*
  * XXX Some programs (like rsyncd) expect to be able to
  * XXX re-initialize optind to 0 and have getopt_long(3)
  * XXX properly function again.  Work around this braindamage.
  */
 if (optind == 0)
  optind = 1;
```

```
	if (optreset)
		nonopt_start = nonopt_end = -1;
start:
	if (optreset || !*place) {	/* update scanning pointer */
		optreset = 0;
		if (optind >= nargc) {          /* end of argument vector */
			place = EMSG;
			if (nonopt_end != -1) {
				/* do permutation, if we have to */
				permute_args(nonopt_start, nonopt_end,
				    optind, nargv);
				optind -= nonopt_end - nonopt_start;
			}
			else if (nonopt_start != -1) {
				/*
				 * If we skipped non-options, set optind
				 * to the first of them.
				 */
				optind = nonopt_start;
			}
			nonopt_start = nonopt_end = -1;
			return -1;
		}
		if ((*(place = nargv[optind]) != '-')
		    || (place[1] == '\0')) {    /* found non-option */
			place = EMSG;
			if (IN_ORDER) {
				/*
				 * GNU extension:
				 * return non-option as argument to option 1
				 */
				optarg = nargv[optind++];
				return INORDER;
			}
			if (!PERMUTE) {
				/*
				 * if no permutation wanted, stop parsing
```

```
     * at first non-option
     */
    return -1;
   }
   /* do permutation */
   if (nonopt_start == -1)
    nonopt_start = optind;
   else if (nonopt_end != -1) {
    permute_args(nonopt_start, nonopt_end,
        optind, nargv);
    nonopt_start = optind -
        (nonopt_end - nonopt_start);
    nonopt_end = -1;
   }
   optind++;
   /* process next argument */
   goto start;
  }
  if (nonopt_start != -1 && nonopt_end == -1)
   nonopt_end = optind;
  if (place[1] && *++place == '-') { /* found "--" */
   place++;
   return -2;
  }
 }
 if ((optchar = (int)*place++) == (int)':' ||
     (oli = strchr(options + (IGNORE_FIRST ? 1 : 0), optchar)) ==
NULL) {
  /* option letter unknown or ':' */
  if (!*place)
   ++optind;
  if (PRINT_ERROR)
   fprintf(stderr, illoptchar, optchar);
  optopt = optchar;
  return BADCH;
 }
 if (optchar == 'W' && oli[1] == ';') {  /* -W long-option */
```

```
    /* XXX: what if no long options provided (called by getopt)? */
    if (*place)
     return -2;

    if (++optind >= nargc) { /* no arg */
     place = EMSG;
     if (PRINT_ERROR)
      fprintf(stderr, recargchar, optchar);
     optopt = optchar;
     return BADARG;
    } else    /* white space */
     place = nargv[optind];
    /*
     * Handle -W arg the same as --arg (which causes getopt to
     * stop parsing).
     */
    return -2;
   }
   if (*++oli != ':') {   /* doesn't take argument */
    if (!*place)
     ++optind;
   } else {    /* takes (optional) argument */
    optarg = NULL;
    if (*place)   /* no white space */
     optarg = __UNCONST(place);
    /* XXX: disable test for :: if PC? (GNU doesn't) */
    else if (oli[1] != ':') { /* arg not optional */
     if (++optind >= nargc) { /* no arg */
      place = EMSG;
      if (PRINT_ERROR)
       fprintf(stderr, recargchar, optchar);
      optopt = optchar;
      return BADARG;
     } else
      optarg = nargv[optind];
    }
    place = EMSG;
```

```
  ++optind;
 }
 /* dump back option letter */
 return optchar;
}

#ifdef REPLACE_GETOPT

int
getopt(int nargc, char * const *nargv, const char *options)
{
 int retval;

 _DIAGASSERT(nargv != NULL);
 _DIAGASSERT(options != NULL);

 retval = getopt_internal(nargc, __UNCONST(nargv), options);
 if (retval == -2) {
  ++optind;
  /*
   * We found an option (--), so if we skipped non-options,
   * we have to permute.
   */
  if (nonopt_end != -1) {
   permute_args(nonopt_start, nonopt_end, optind,
           __UNCONST(nargv));
   optind -= nonopt_end - nonopt_start;
  }
  nonopt_start = nonopt_end = -1;
  retval = -1;
 }
 return retval;
}
#endif

int
getopt_long(int nargc, char * const *nargv, const char *options,
```

```
    const struct option *long_options, int *idx)
{
 int retval;

#define IDENTICAL_INTERPRETATION(_x, _y)    \
 (long_options[(_x)].has_arg == long_options[(_y)].has_arg && \
  long_options[(_x)].flag == long_options[(_y)].flag &&  \
  long_options[(_x)].val == long_options[(_y)].val)

 _DIAGASSERT(nargv != NULL);
 _DIAGASSERT(options != NULL);
 _DIAGASSERT(long_options != NULL);
 /* idx may be NULL */

 retval = getopt_internal(nargc, __UNCONST(nargv), options);
 if (retval == -2) {
  char *current_argv, *has_equal;
  size_t current_argv_len;
  int i, ambiguous, match;

  current_argv = __UNCONST(place);
  match = -1;
  ambiguous = 0;

  optind++;
  place = EMSG;

  if (*current_argv == '\0') {  /* found "--" */
   /*
    * We found an option (--), so if we skipped
    * non-options, we have to permute.
    */
   if (nonopt_end != -1) {
    permute_args(nonopt_start, nonopt_end,
        optind, __UNCONST(nargv));
    optind -= nonopt_end - nonopt_start;
   }
```

```
 nonopt_start = nonopt_end = -1;
 return -1;
}
if ((has_equal = strchr(current_argv, '=')) != NULL) {
 /* argument found (--option=arg) */
 current_argv_len = has_equal - current_argv;
 has_equal++;
} else
 current_argv_len = strlen(current_argv);

for (i = 0; long_options[i].name; i++) {
 /* find matching long option */
 if (strncmp(current_argv, long_options[i].name,
     current_argv_len))
  continue;

 if (strlen(long_options[i].name) ==
     (unsigned)current_argv_len) {
  /* exact match */
  match = i;
  ambiguous = 0;
  break;
 }
 if (match == -1)  /* partial match */
  match = i;
 else if (!IDENTICAL_INTERPRETATION(i, match))
  ambiguous = 1;
}
if (ambiguous) {
 /* ambiguous abbreviation */
 if (PRINT_ERROR)
  fprintf(stderr, ambig, (int)current_argv_len,
       current_argv);
 optopt = 0;
 return BADCH;
}
if (match != -1) {   /* option found */
```

```
      if (long_options[match].has_arg == no_argument
    && has_equal) {
 if (PRINT_ERROR)
  fprintf(stderr, noarg, (int)current_argv_len,
      current_argv);
 /*
  * XXX: GNU sets optopt to val regardless of
  * flag
  */
 if (long_options[match].flag == NULL)
  optopt = long_options[match].val;
 else
  optopt = 0;
 return BADARG;
}
if (long_options[match].has_arg == required_argument ||
    long_options[match].has_arg == optional_argument) {
 if (has_equal)
  optarg = has_equal;
 else if (long_options[match].has_arg ==
     required_argument) {
  /*
   * optional argument doesn't use
   * next nargv
   */
  optarg = nargv[optind++];
 }
}
if ((long_options[match].has_arg == required_argument)
    && (optarg == NULL)) {
 /*
  * Missing argument; leading ':'
  * indicates no error should be generated
  */
 if (PRINT_ERROR)
  fprintf(stderr, recargstring, current_argv);
 /*
```

```
     * XXX: GNU sets optopt to val regardless
     * of flag
     */
    if (long_options[match].flag == NULL)
     optopt = long_options[match].val;
    else
     optopt = 0;
    --optind;
    return BADARG;
   }
  } else {   /* unknown option */
   if (PRINT_ERROR)
    fprintf(stderr, illoptstring, current_argv);
   optopt = 0;
   return BADCH;
  }
  if (long_options[match].flag) {
   *long_options[match].flag = long_options[match].val;
   retval = 0;
  } else
   retval = long_options[match].val;
  if (idx)
   *idx = match;
 }
 return retval;
#undef IDENTICAL_INTERPRETATION
}
```

Chapter 33

./src/windows/com/ttsengine.cpp

```
#include "config.h"

#include <windows.h>
#include <sapiddk.h>
#include <sperror.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>

#include <new>
#include <errno.h>

extern "C" ULONG ObjectCount;

static HRESULT espeak_status_to_hresult(espeak_ng_STATUS status)
{
 switch (status)
 {
 case ENS_OK: return S_OK;
 case EACCES: return E_ACCESSDENIED;
 case EINVAL: return E_INVALIDARG;
 case ENOENT: return HRESULT_FROM_WIN32(ERROR_FILE_NOT_FOUND);
```

```
 case ENOMEM: return E_OUTOFMEMORY;
 default:     return E_FAIL;
 }
}

struct TtsEngine
 : public ISpObjectWithToken
 , public ISpTTSEngine
{
 TtsEngine();
 ~TtsEngine();

 // IUnknown

 ULONG __stdcall AddRef();
 ULONG __stdcall Release();

 HRESULT __stdcall QueryInterface(REFIID iid, void **object);

 // ISpObjectWithToken

 HRESULT __stdcall GetObjectToken(ISpObjectToken **token);
 HRESULT __stdcall SetObjectToken(ISpObjectToken *token);

 // ISpTTSEngine

 HRESULT __stdcall
 Speak(DWORD flags,
       REFGUID formatId,
       const WAVEFORMATEX *format,
       const SPVTEXTFRAG *textFragList,
       ISpTTSEngineSite *site);

 HRESULT __stdcall
 GetOutputFormat(const GUID *targetFormatId,
                 const WAVEFORMATEX *targetFormat,
                 GUID *formatId,
```

```
                 WAVEFORMATEX **format);

 int OnEvent(short *data, int samples, espeak_EVENT *events);
private:
 HRESULT GetStringValue(LPCWSTR key, char *&value);

 ULONG refCount;
 ISpObjectToken *objectToken;
 ISpTTSEngineSite *site;
};

static int
espeak_callback(short *data, int samples, espeak_EVENT *events)
{
 TtsEngine *engine = (TtsEngine *)events->user_data;
 return engine->OnEvent(data, samples, events);
}

TtsEngine::TtsEngine()
 : refCount(1)
 , objectToken(NULL)
 , site(NULL)
{
 InterlockedIncrement(&ObjectCount);
}

TtsEngine::~TtsEngine()
{
 InterlockedDecrement(&ObjectCount);
 if (objectToken)
  objectToken->Release();
}

ULONG __stdcall TtsEngine::AddRef()
{
 return InterlockedIncrement(&refCount);
}
```

```
ULONG __stdcall TtsEngine::Release()
{
 ULONG ret = InterlockedDecrement(&refCount);
 if (ret == 0)
  delete this;
 return ret;
}

HRESULT __stdcall TtsEngine::QueryInterface(REFIID iid, void
**object)
{

 if (IsEqualIID(iid, IID_IUnknown) || IsEqualIID(iid,
IID_ISpTTSEngine))
  *object = (ISpTTSEngine *)this;
 else if (IsEqualIID(iid, IID_ISpObjectWithToken))
  *object = (ISpObjectWithToken *)this;
 else
  return E_NOINTERFACE;

 this->AddRef();
 return S_OK;
}

HRESULT __stdcall TtsEngine::GetObjectToken(ISpObjectToken
**token)
{
 if (!token)
  return E_POINTER;

 if (objectToken) {
  objectToken->AddRef();
  return S_OK;
 }
 return S_FALSE;
}
```

```
HRESULT __stdcall TtsEngine::SetObjectToken(ISpObjectToken
*token)
{
 if (!token)
  return E_INVALIDARG;

 if (objectToken)
  return SPERR_ALREADY_INITIALIZED;

 objectToken = token;
 objectToken->AddRef();

 char *path = NULL;
 GetStringValue(L"Path", path);
 espeak_ng_InitializePath(path);
 if (path)
  free(path);

 espeak_ng_STATUS status;
 status = espeak_ng_Initialize(NULL);
 if (status == ENS_OK)
  status = espeak_ng_InitializeOutput(ENOUTPUT_MODE_SYNCHRONOUS,
100, NULL);

 espeak_SetSynthCallback(espeak_callback);

 char *voiceName = NULL;
 if (SUCCEEDED(GetStringValue(L"VoiceName", voiceName))) {
  if (status == ENS_OK)
   status = espeak_ng_SetVoiceByName(voiceName);
  free(voiceName);
 }

 return espeak_status_to_hresult(status);
}
```

```
HRESULT __stdcall
TtsEngine::Speak(DWORD flags,
                 REFGUID formatId,
                 const WAVEFORMATEX *format,
                 const SPVTEXTFRAG *textFragList,
                 ISpTTSEngineSite *site)
{
 if (!site || !textFragList)
  return E_INVALIDARG;

 this->site = site;

 while (textFragList != NULL)
 {
  DWORD actions = site->GetActions();
  if (actions & SPVES_ABORT)
   return S_OK;

  switch (textFragList->State.eAction)
  {
  case SPVA_Speak:
   espeak_ng_Synthesize(textFragList->pTextStart, 0, 0,
POS_CHARACTER, 0, espeakCHARS_WCHAR, NULL, this);
   break;
  }

  textFragList = textFragList->pNext;
 }

 return E_NOTIMPL;
}

HRESULT __stdcall
TtsEngine::GetOutputFormat(const GUID *targetFormatId,
                           const WAVEFORMATEX *targetFormat,
                           GUID *formatId,
                           WAVEFORMATEX **format)
```

```
{

 if (!*format)
  return E_OUTOFMEMORY;
 (*format)->wFormatTag = WAVE_FORMAT_PCM;
 (*format)->nChannels = 1;
 (*format)->nBlockAlign = 2;
 (*format)->nSamplesPerSec = 22050;
 (*format)->wBitsPerSample = 16;
 (*format)->nAvgBytesPerSec = (*format)->nSamplesPerSec *
(*format)->nBlockAlign;
 (*format)->cbSize = 0;

 return S_OK;
}

int
TtsEngine::OnEvent(short *data, int samples, espeak_EVENT
*events)
{
 DWORD actions = site->GetActions();
 if (actions & SPVES_ABORT)
  return 1;

 if (data)
  site->Write(data, samples * 2, NULL);
 return 0;
}

HRESULT TtsEngine::GetStringValue(LPCWSTR key, char *&value)
{
 if (!objectToken)
  return E_FAIL;

 LPWSTR wvalue = NULL;
 HRESULT hr = objectToken->GetStringValue(key, &wvalue);
 if (FAILED(hr))
```

```
  return hr;

 size_t len = wcslen(wvalue);
 value = (char *)malloc(len + 1);
 if (!value) {
  CoTaskMemFree(wvalue);
  return E_OUTOFMEMORY;
 }

 wcstombs(value, wvalue, len + 1);
 CoTaskMemFree(wvalue);

 return S_OK;
}

extern "C" HRESULT __stdcall
TtsEngine_CreateInstance(IClassFactory *iface, IUnknown *outer,
REFIID iid, void **object)
{
 if (outer != NULL)
  return CLASS_E_NOAGGREGATION;

 TtsEngine *engine = new (std::nothrow) TtsEngine();
 if (!engine)
  return E_OUTOFMEMORY;

 HRESULT ret = engine->QueryInterface(iid, object);
 engine->Release();
 return ret;
}
```

Chapter 34

./src/ucd-tools/src/ctype.c

```
#include "ucd/ucd.h"

int ucd_isalnum(codepoint_t c)
{
 ucd_category cat = ucd_lookup_category(c);
 switch (cat)
 {
 case UCD_CATEGORY_Lu:
 case UCD_CATEGORY_Ll:
 case UCD_CATEGORY_Lt:
 case UCD_CATEGORY_Lm:
 case UCD_CATEGORY_Lo:
 case UCD_CATEGORY_Nl:
 case UCD_CATEGORY_Nd:
 case UCD_CATEGORY_No:
  return 1;
 case UCD_CATEGORY_Mn:
 case UCD_CATEGORY_Mc:
 case UCD_CATEGORY_So:
  return (ucd_properties(c, cat) & UCD_PROPERTY_OTHER_ALPHABETIC)
== UCD_PROPERTY_OTHER_ALPHABETIC;
 default:
```

```
  return 0;
 }
}

int ucd_isalpha(codepoint_t c)
{
 ucd_category cat = ucd_lookup_category(c);
 switch (cat)
 {
 case UCD_CATEGORY_Lu:
 case UCD_CATEGORY_Ll:
 case UCD_CATEGORY_Lt:
 case UCD_CATEGORY_Lm:
 case UCD_CATEGORY_Lo:
 case UCD_CATEGORY_Nl:
  return 1;
 case UCD_CATEGORY_Mn:
 case UCD_CATEGORY_Mc:
 case UCD_CATEGORY_So:
  return (ucd_properties(c, cat) & UCD_PROPERTY_OTHER_ALPHABETIC)
== UCD_PROPERTY_OTHER_ALPHABETIC;
 default:
  return 0;
 }
}

int ucd_isblank(codepoint_t c)
{
 switch (ucd_lookup_category(c))
 {
 case UCD_CATEGORY_Zs:
  switch (c) /* Exclude characters with the <noBreak>
DispositionType */
  {
  case 0x00A0: /* U+00A0 : NO-BREAK SPACE */
  case 0x2007: /* U+2007 : FIGURE SPACE */
  case 0x202F: /* U+202F : NARROW NO-BREAK SPACE */
```

```
   return 0;
  }
  return 1;
 case UCD_CATEGORY_Cc:
  return c == 0x09; /* U+0009 : CHARACTER TABULATION */
 default:
  return 0;
 }
}

int ucd_iscntrl(codepoint_t c)
{
 return ucd_lookup_category(c) == UCD_CATEGORY_Cc;
}

int ucd_isdigit(codepoint_t c)
{
 return (c >= 0x30 && c <= 0x39); /* [0-9] */
}

int ucd_isgraph(codepoint_t c)
{
 switch (ucd_lookup_category(c))
 {
 case UCD_CATEGORY_Cc:
 case UCD_CATEGORY_Cf:
 case UCD_CATEGORY_Cn:
 case UCD_CATEGORY_Co:
 case UCD_CATEGORY_Cs:
 case UCD_CATEGORY_Zl:
 case UCD_CATEGORY_Zp:
 case UCD_CATEGORY_Zs:
 case UCD_CATEGORY_Ii:
  return 0;
 default:
  return 1;
 }
```

```
}

int ucd_islower(codepoint_t c)
{
 ucd_category cat = ucd_lookup_category(c);
 switch (cat)
 {
 case UCD_CATEGORY_Ll:
  return 1;
 case UCD_CATEGORY_Lt:
  return ucd_toupper(c) != c;
 case UCD_CATEGORY_Lm:
 case UCD_CATEGORY_Lo:
 case UCD_CATEGORY_Mn:
 case UCD_CATEGORY_Nl:
 case UCD_CATEGORY_So:
  return (ucd_properties(c, cat) & UCD_PROPERTY_OTHER_LOWERCASE)
== UCD_PROPERTY_OTHER_LOWERCASE;
 default:
  return 0;
 }
}

int ucd_isprint(codepoint_t c)
{
 switch (ucd_lookup_category(c))
 {
 case UCD_CATEGORY_Cc:
 case UCD_CATEGORY_Cf:
 case UCD_CATEGORY_Cn:
 case UCD_CATEGORY_Co:
 case UCD_CATEGORY_Cs:
 case UCD_CATEGORY_Ii:
  return 0;
 default:
  return 1;
 }
```

```
}

int ucd_ispunct(codepoint_t c)
{
 return ucd_isgraph(c) && !ucd_isalnum(c);
}

int ucd_isspace(codepoint_t c)
{
 switch (ucd_lookup_category(c))
 {
 case UCD_CATEGORY_Zl:
 case UCD_CATEGORY_Zp:
  return 1;
 case UCD_CATEGORY_Zs:
  switch (c) /* Exclude characters with the <noBreak>
DispositionType */
  {
  case 0x00A0: /* U+00A0 : NO-BREAK SPACE */
  case 0x2007: /* U+2007 : FIGURE SPACE */
  case 0x202F: /* U+202F : NARROW NO-BREAK SPACE */
   return 0;
  }
  return 1;
 case UCD_CATEGORY_Cc:
  switch (c) /* Include control characters marked as White_Space
*/
  {
  case 0x09: /* U+0009 : CHARACTER TABULATION */
  case 0x0A: /* U+000A : LINE FEED */
  case 0x0B: /* U+000B : LINE TABULATION */
  case 0x0C: /* U+000C : FORM FEED */
  case 0x0D: /* U+000D : CARRIAGE RETURN */
  case 0x85: /* U+0085 : NEXT LINE */
   return 1;
  }
 default:
```

```
  return 0;
 }
}

int ucd_isupper(codepoint_t c)
{
 ucd_category cat = ucd_lookup_category(c);
 switch (cat)
 {
 case UCD_CATEGORY_Lu:
  return 1;
 case UCD_CATEGORY_Lt:
  return ucd_tolower(c) != c;
 case UCD_CATEGORY_Nl:
 case UCD_CATEGORY_So:
  return (ucd_properties(c, cat) & UCD_PROPERTY_OTHER_UPPERCASE)
== UCD_PROPERTY_OTHER_UPPERCASE;
 default:
  return 0;
 }
}

int ucd_isxdigit(codepoint_t c)
{
 return (c >= 0x30 && c <= 0x39)  /* [0-9] */
     || (c >= 0x41 && c <= 0x46)  /* [A-Z] */
     || (c >= 0x61 && c <= 0x66); /* [a-z] */
}
```

Chapter 35

./src/ucd-tools/tests/printucddata.c

```
#include "ucd/ucd.h"

#include <string.h>
#include <stdio.h>

static void fput_utf8c(FILE *out, codepoint_t c)
{
 if (c < 0x80)
  fputc((uint8_t)c, out);
 else if (c < 0x800)
 {
  fputc(0xC0 | (c >> 6), out);
  fputc(0x80 + (c & 0x3F), out);
 }
 else if (c < 0x10000)
 {
  fputc(0xE0 | (c >> 12), out);
  fputc(0x80 + ((c >> 6) & 0x3F), out);
  fputc(0x80 + (c & 0x3F), out);
 }
```

```
 else if (c < 0x200000)
 {
  fputc(0xF0 | (c >> 18), out);
  fputc(0x80 + ((c >> 12) & 0x3F), out);
  fputc(0x80 + ((c >>  6) & 0x3F), out);
  fputc(0x80 + (c & 0x3F), out);
 }
}

static int fget_utf8c(FILE *in, codepoint_t *c)
{
 int ch = EOF;
 if ((ch = fgetc(in)) == EOF) return 0;
 if ((uint8_t)ch < 0x80)
  *c = (uint8_t)ch;
 else switch ((uint8_t)ch & 0xF0)
 {
 default:
  *c = (uint8_t)ch & 0x1F;
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
  break;
 case 0xE0:
  *c = (uint8_t)ch & 0x0F;
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
  break;
 case 0xF0:
  *c = (uint8_t)ch & 0x07;
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
```

```
  break;
 }
 return 1;
}

static void uprintf_codepoint(FILE *out, codepoint_t c, char
mode)
{
 switch (mode)
 {
 case 'c': /* character */
  switch (c)
  {
  case '\t': fputs("\\t", out);  break;
  case '\r': fputs("\\r", out);  break;
  case '\n': fputs("\\n", out);  break;
  default:   fput_utf8c(out, c); break;
  }
  break;
 case 'h': /* hexadecimal (lower) */
  fprintf(out, "%06x", c);
  break;
 case 'H': /* hexadecimal (upper) */
  fprintf(out, "%06X", c);
  break;
 }
}

static void uprintf_is(FILE *out, codepoint_t c, char mode)
{
 switch (mode)
 {
 case 'A': /* alpha-numeric */
  fputc(ucd_isalnum(c) ? '1' : '0', out);
  break;
 case 'a': /* alpha */
  fputc(ucd_isalpha(c) ? '1' : '0', out);
```

```
  break;
 case 'b': /* blank */
  fputc(ucd_isblank(c) ? '1' : '0', out);
  break;
 case 'c': /* control */
  fputc(ucd_iscntrl(c) ? '1' : '0', out);
  break;
 case 'd': /* numeric */
  fputc(ucd_isdigit(c) ? '1' : '0', out);
  break;
 case 'g': /* glyph */
  fputc(ucd_isgraph(c) ? '1' : '0', out);
  break;
 case 'l': /* lower case */
  fputc(ucd_islower(c) ? '1' : '0', out);
  break;
 case 'P': /* printable */
  fputc(ucd_isprint(c) ? '1' : '0', out);
  break;
 case 'p': /* punctuation */
  fputc(ucd_ispunct(c) ? '1' : '0', out);
  break;
 case 's': /* whitespace */
  fputc(ucd_isspace(c) ? '1' : '0', out);
  break;
 case 'u': /* upper case */
  fputc(ucd_isupper(c) ? '1' : '0', out);
  break;
 case 'x': /* xdigit */
  fputc(ucd_isxdigit(c) ? '1' : '0', out);
  break;
 }
}

static void uprintf(FILE *out, codepoint_t c, const char *format)
{
 while (*format) switch (*format)
```

```
 {
 case '%':
  switch (*++format)
  {
  case 'c': /* category */
   fputs(ucd_get_category_string(ucd_lookup_category(c)), out);
   break;
  case 'C': /* category group */
   fputs(ucd_get_category_group_string(ucd_lookup_category_group(
c)), out);
   break;
  case 'p': /* codepoint */
   uprintf_codepoint(out, c, *++format);
   break;
  case 'P': /* properties */
   fprintf(out, "%016llx", ucd_properties(c,
ucd_lookup_category(c)));
   break;
  case 'i': /* is* */
   uprintf_is(out, c, *++format);
   break;
  case 'L': /* lowercase */
   uprintf_codepoint(out, ucd_tolower(c), *++format);
   break;
  case 's': /* script */
   fputs(ucd_get_script_string(ucd_lookup_script(c)), out);
   break;
  case 'T': /* titlecase */
   uprintf_codepoint(out, ucd_totitle(c), *++format);
   break;
  case 'U': /* uppercase */
   uprintf_codepoint(out, ucd_toupper(c), *++format);
   break;
  }
  ++format;
  break;
 case '\\':
```

```
  switch (*++format) {
  case 0:
   break;
  case 't':
   fputc('\t', out);
   ++format;
   break;
  case 'r':
   fputc('\r', out);
   ++format;
   break;
  case 'n':
   fputc('\n', out);
   ++format;
   break;
  default:
   fputc(*format, out);
   ++format;
   break;
  }
  break;
 default:
  fputc(*format, out);
  ++format;
  break;
 }
}

static void print_file(FILE *in, const char *format)
{
 codepoint_t c = 0;
 while (fget_utf8c(in, &c))
  uprintf(stdout, c, format ? format :
"%pc\t%pH\t%s\t%c\t%Uc\t%Lc\t%Tc\t%is\n");
}

int main(int argc, char **argv)
```

```
{
 FILE *in = NULL;
 const char *format = NULL;
 int argn;
 for (argn = 1; argn != argc; ++argn)
 {
  const char *arg = argv[argn];
  if (!strcmp(arg, "--stdin") || !strcmp(arg, "-"))
   in = stdin;
  else if (!strncmp(arg, "--format=", 9))
   format = arg + 9;
  else if (in == NULL)
  {
   in = fopen(arg, "r");
   if (!in)
    fprintf(stdout, "cannot open `%s`\n", argv[1]);
  }
 }

 if (in == stdin)
  print_file(stdin, format);
 else if (in != NULL)
 {
  print_file(in, format);
  fclose(in);
 }
 else
 {
  codepoint_t c;
  for (c = 0; c <= 0x10FFFF; ++c)
   uprintf(stdout, c, format ? format :
          "%pH %s %C %c %UH %LH %TH %id %ix %ic %is %ib %ip %iP
%ig %iA %ia %iu %il %P\n");
 }
 return 0;
}
```

Chapter 36

./src/ucd-tools/tests/printucddata_cpp.cpp

```
#include "ucd/ucd.h"

#include <string.h>
#include <stdio.h>

void fput_utf8c(FILE *out, ucd::codepoint_t c)
{
 if (c < 0x80)
  fputc((uint8_t)c, out);
 else if (c < 0x800)
 {
  fputc(0xC0 | (c >> 6), out);
  fputc(0x80 + (c & 0x3F), out);
 }
 else if (c < 0x10000)
 {
  fputc(0xE0 | (c >> 12), out);
  fputc(0x80 + ((c >> 6) & 0x3F), out);
  fputc(0x80 + (c & 0x3F), out);
 }
```

```
 else if (c < 0x200000)
 {
  fputc(0xF0 | (c >> 18), out);
  fputc(0x80 + ((c >> 12) & 0x3F), out);
  fputc(0x80 + ((c >>  6) & 0x3F), out);
  fputc(0x80 + (c & 0x3F), out);
 }
}

bool fget_utf8c(FILE *in, ucd::codepoint_t &c)
{
 int ch = EOF;
 if ((ch = fgetc(in)) == EOF) return false;
 if (uint8_t(ch) < 0x80)
  c = uint8_t(ch);
 else switch (uint8_t(ch) & 0xF0)
 {
 default:
  c = uint8_t(ch) & 0x1F;
  if ((ch = fgetc(in)) == EOF) return false;
  c = (c << 6) + (uint8_t(ch) & 0x3F);
  break;
 case 0xE0:
  c = uint8_t(ch) & 0x0F;
  if ((ch = fgetc(in)) == EOF) return false;
  c = (c << 6) + (uint8_t(ch) & 0x3F);
  if ((ch = fgetc(in)) == EOF) return false;
  c = (c << 6) + (uint8_t(ch) & 0x3F);
  break;
 case 0xF0:
  c = uint8_t(ch) & 0x07;
  if ((ch = fgetc(in)) == EOF) return false;
  c = (c << 6) + (uint8_t(ch) & 0x3F);
  if ((ch = fgetc(in)) == EOF) return false;
  c = (c << 6) + (uint8_t(ch) & 0x3F);
  if ((ch = fgetc(in)) == EOF) return false;
  c = (c << 6) + (uint8_t(ch) & 0x3F);
```

```
  break;
 }
 return true;
}

void uprintf_codepoint(FILE *out, ucd::codepoint_t c, char mode)
{
 switch (mode)
 {
 case 'c': // character
  switch (c)
  {
  case '\t': fputs("\\t", out);  break;
  case '\r': fputs("\\r", out);  break;
  case '\n': fputs("\\n", out);  break;
  default:   fput_utf8c(out, c); break;
  }
  break;
 case 'h': // hexadecimal (lower)
  fprintf(out, "%06x", c);
  break;
 case 'H': // hexadecimal (upper)
  fprintf(out, "%06X", c);
  break;
 }
}

void uprintf_is(FILE *out, ucd::codepoint_t c, char mode)
{
 switch (mode)
 {
 case 'A': // alpha-numeric
  fputc(ucd::isalnum(c) ? '1' : '0', out);
  break;
 case 'a': // alpha
  fputc(ucd::isalpha(c) ? '1' : '0', out);
  break;
```

```
 case 'b': // blank
  fputc(ucd::isblank(c) ? '1' : '0', out);
  break;
 case 'c': // control
  fputc(ucd::iscntrl(c) ? '1' : '0', out);
  break;
 case 'd': // numeric
  fputc(ucd::isdigit(c) ? '1' : '0', out);
  break;
 case 'g': // glyph
  fputc(ucd::isgraph(c) ? '1' : '0', out);
  break;
 case 'l': // lower case
  fputc(ucd::islower(c) ? '1' : '0', out);
  break;
 case 'P': // printable
  fputc(ucd::isprint(c) ? '1' : '0', out);
  break;
 case 'p': // punctuation
  fputc(ucd::ispunct(c) ? '1' : '0', out);
  break;
 case 's': // whitespace
  fputc(ucd::isspace(c) ? '1' : '0', out);
  break;
 case 'u': // upper case
  fputc(ucd::isupper(c) ? '1' : '0', out);
  break;
 case 'x': // xdigit
  fputc(ucd::isxdigit(c) ? '1' : '0', out);
  break;
 }
}

void uprintf(FILE *out, ucd::codepoint_t c, const char *format)
{
 while (*format) switch (*format)
 {
```

```
 case '%':
  switch (*++format)
  {
  case 'c': // category
   fputs(ucd::get_category_string(ucd::lookup_category(c)), out);
   break;
  case 'C': // category group
   fputs(ucd::get_category_group_string(ucd::lookup_category_grou
p(c)), out);
   break;
  case 'p': // codepoint
   uprintf_codepoint(out, c, *++format);
   break;
  case 'P': // properties
   fprintf(out, "%016llx", ucd::properties(c,
ucd::lookup_category(c)));
   break;
  case 'i': // is*
   uprintf_is(out, c, *++format);
   break;
  case 'L': // lowercase
   uprintf_codepoint(out, ucd::tolower(c), *++format);
   break;
  case 's': // script
   fputs(ucd::get_script_string(ucd::lookup_script(c)), out);
   break;
  case 'T': // titlecase
   uprintf_codepoint(out, ucd::totitle(c), *++format);
   break;
  case 'U': // uppercase
   uprintf_codepoint(out, ucd::toupper(c), *++format);
   break;
  }
  ++format;
  break;
 case '\\':
  switch (*++format) {
```

```
  case 0:
   break;
  case 't':
   fputc('\t', out);
   ++format;
   break;
  case 'r':
   fputc('\r', out);
   ++format;
   break;
  case 'n':
   fputc('\n', out);
   ++format;
   break;
  default:
   fputc(*format, out);
   ++format;
   break;
  }
  break;
 default:
  fputc(*format, out);
  ++format;
  break;
 }
}

void print_file(FILE *in, const char *format)
{
 ucd::codepoint_t c = 0;
 while (fget_utf8c(in, c))
  uprintf(stdout, c, format ? format :
"%pc\t%pH\t%s\t%c\t%Uc\t%Lc\t%Tc\t%is\n");
}

int main(int argc, char **argv)
{
```

```
 FILE *in = NULL;
 const char *format = NULL;
 for (int argn = 1; argn != argc; ++argn)
 {
  const char *arg = argv[argn];
  if (!strcmp(arg, "--stdin") || !strcmp(arg, "-"))
   in = stdin;
  else if (!strncmp(arg, "--format=", 9))
   format = arg + 9;
  else if (in == NULL)
  {
   in = fopen(arg, "r");
   if (!in)
    fprintf(stdout, "cannot open `%s`\n", argv[1]);
  }
 }

 if (in == stdin)
  print_file(stdin, format);
 else if (in != NULL)
 {
  print_file(in, format);
  fclose(in);
 }
 else
 {
  for (ucd::codepoint_t c = 0; c <= 0x10FFFF; ++c)
   uprintf(stdout, c, format ? format :
           "%pH %s %C %c %UH %LH %TH %id %ix %ic %is %ib %ip %iP
%ig %iA %ia %iu %il %P\n");
 }
 return 0;
}
```

Chapter 37

./src/ucd-tools/tests/printcdata.c

```
#include "config.h"
#include "ucd/ucd.h"

#include <locale.h>
#include <string.h>
#include <stdio.h>
#include <wchar.h>
#include <wctype.h>

#ifndef HAVE_ISWBLANK
static int iswblank(wint_t c)
{
 return iswspace(c) && !(c >= 0x0A && c <= 0x0D);
}
#endif

static void fput_utf8c(FILE *out, codepoint_t c)
{
 if (c < 0x80)
  fputc((uint8_t)c, out);
 else if (c < 0x800)
 {
```

```
  fputc(0xC0 | (c >> 6), out);
  fputc(0x80 + (c & 0x3F), out);
 }
 else if (c < 0x10000)
 {
  fputc(0xE0 | (c >> 12), out);
  fputc(0x80 + ((c >> 6) & 0x3F), out);
  fputc(0x80 + (c & 0x3F), out);
 }
 else if (c < 0x200000)
 {
  fputc(0xF0 | (c >> 18), out);
  fputc(0x80 + ((c >> 12) & 0x3F), out);
  fputc(0x80 + ((c >>  6) & 0x3F), out);
  fputc(0x80 + (c & 0x3F), out);
 }
}

static int fget_utf8c(FILE *in, codepoint_t *c)
{
 int ch = EOF;
 if ((ch = fgetc(in)) == EOF) return 0;
 if ((uint8_t)ch < 0x80)
  *c = (uint8_t)ch;
 else switch ((uint8_t)ch & 0xF0)
 {
 default:
  *c = (uint8_t)ch & 0x1F;
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
  break;
 case 0xE0:
  *c = (uint8_t)ch & 0x0F;
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
```

```
  break;
 case 0xF0:
  *c = (uint8_t)ch & 0x07;
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
  if ((ch = fgetc(in)) == EOF) return 0;
  *c = (*c << 6) + ((uint8_t)ch & 0x3F);
  break;
 }
 return 1;
}

static void uprintf_codepoint(FILE *out, codepoint_t c, char
mode)
{
 switch (mode)
 {
 case 'c': /* character */
  switch (c)
  {
  case '\t': fputs("\\t", out);  break;
  case '\r': fputs("\\r", out);  break;
  case '\n': fputs("\\n", out);  break;
  default:   fput_utf8c(out, c); break;
  }
  break;
 case 'h': /* hexadecimal (lower) */
  fprintf(out, "%06x", c);
  break;
 case 'H': /* hexadecimal (upper) */
  fprintf(out, "%06X", c);
  break;
 }
}
```

```
static void uprintf_is(FILE *out, codepoint_t c, char mode)
{
 switch (mode)
 {
 case 'A': /* alpha-numeric */
  fputc(iswalnum(c) ? '1' : '0', out);
  break;
 case 'a': /* alpha */
  fputc(iswalpha(c) ? '1' : '0', out);
  break;
 case 'b': /* blank */
  fputc(iswblank(c) ? '1' : '0', out);
  break;
 case 'c': /* control */
  fputc(iswcntrl(c) ? '1' : '0', out);
  break;
 case 'd': /* numeric */
  fputc(iswdigit(c) ? '1' : '0', out);
  break;
 case 'g': /* glyph */
  fputc(iswgraph(c) ? '1' : '0', out);
  break;
 case 'l': /* lower case */
  fputc(iswlower(c) ? '1' : '0', out);
  break;
 case 'P': /* printable */
  fputc(iswprint(c) ? '1' : '0', out);
  break;
 case 'p': /* punctuation */
  fputc(iswpunct(c) ? '1' : '0', out);
  break;
 case 's': /* whitespace */
  fputc(iswspace(c) ? '1' : '0', out);
  break;
 case 'u': /* upper case */
  fputc(iswupper(c) ? '1' : '0', out);
  break;
```

```
 case 'x': /* xdigit */
  fputc(iswxdigit(c) ? '1' : '0', out);
  break;
 }
}

static void uprintf(FILE *out, codepoint_t c, const char *format)
{
 while (*format) switch (*format)
 {
 case '%':
  switch (*++format)
  {
  case 'c': /* category */
   fputs(ucd_get_category_string(ucd_lookup_category(c)), out);
   break;
  case 'C': /* category group */
   fputs(ucd_get_category_group_string(ucd_lookup_category_group(
c)), out);
   break;
  case 'p': /* codepoint */
   uprintf_codepoint(out, c, *++format);
   break;
  case 'P': /* properties */
   fprintf(out, "%016llx", ucd_properties(c,
ucd_lookup_category(c)));
   break;
  case 'i': /* is* */
   uprintf_is(out, c, *++format);
   break;
  case 'L': /* lowercase */
   uprintf_codepoint(out, ucd_tolower(c), *++format);
   break;
  case 's': /* script */
   fputs(ucd_get_script_string(ucd_lookup_script(c)), out);
   break;
  case 'T': /* titlecase */
```

```
   uprintf_codepoint(out, ucd_totitle(c), *++format);
   break;
  case 'U': /* uppercase */
   uprintf_codepoint(out, ucd_toupper(c), *++format);
   break;
  }
  ++format;
  break;
 case '\\':
  switch (*++format) {
  case 0:
   break;
  case 't':
   fputc('\t', out);
   ++format;
   break;
  case 'r':
   fputc('\r', out);
   ++format;
   break;
  case 'n':
   fputc('\n', out);
   ++format;
   break;
  default:
   fputc(*format, out);
   ++format;
   break;
  }
  break;
 default:
  fputc(*format, out);
  ++format;
  break;
 }
}
```

```
static void print_file(FILE *in, const char *format)
{
 codepoint_t c = 0;
 while (fget_utf8c(in, &c))
  uprintf(stdout, c, format ? format :
"%pc\t%pH\t%s\t%c\t%Uc\t%Lc\t%Tc\t%is\n");
}

int main(int argc, char **argv)
{
 FILE *in = NULL;
 const char *format = NULL;
 int argn;
 for (argn = 1; argn != argc; ++argn)
 {
  const char *arg = argv[argn];
  if (!strcmp(arg, "--stdin") || !strcmp(arg, "-"))
   in = stdin;
  else if (!strncmp(arg, "--format=", 9))
   format = arg + 9;
  else if (!strncmp(arg, "--locale=", 9))
   setlocale(LC_CTYPE, arg + 9);
  else if (in == NULL)
  {
   in = fopen(arg, "r");
   if (!in)
    fprintf(stdout, "cannot open `%s`\n", argv[1]);
  }
 }

 if (in == stdin)
  print_file(stdin, format);
 else if (in != NULL)
 {
  print_file(in, format);
  fclose(in);
 }
```

```
 else
 {
  codepoint_t c;
  for (c = 0; c <= 0x10FFFF; ++c)
   uprintf(stdout, c, format ? format :
           "%pH %s %C %c %UH %LH %TH %id %ix %ic %is %ib %ip %iP
%ig %iA %ia %iu %il %P\n");
 }
 return 0;
}
```

Chapter 38

./src/libespeak-ng/translate.c

```
#include "config.h"

#include <ctype.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <wctype.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "dictionary.h"
#include "numbers.h"
#include "phonemelist.h"
#include "readclause.h"
#include "synthdata.h"

#include "speech.h"
```

```
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

Translator *translator = NULL; // the main translator
Translator *translator2 = NULL; // secondary translator for
certain words
static char translator2_language[20] = { 0 };

FILE *f_trans = NULL; // phoneme output text
int option_tone_flags = 0; // bit 8=emphasize allcaps, bit
9=emphasize penultimate stress
int option_phonemes = 0;
int option_phoneme_events = 0;
int option_endpause = 0; // suppress pause after end of text
int option_capitals = 0;
int option_punctuation = 0;
int option_sayas = 0;
static int option_sayas2 = 0; // used in translate_clause()
static int option_emphasis = 0; // 0=normal, 1=normal, 2=weak,
3=moderate, 4=strong
int option_ssml = 0;
int option_phoneme_input = 0; // allow [[phonemes]] in input
int option_wordgap = 0;

static int count_sayas_digits;
int skip_sentences;
int skip_words;
int skip_characters;
char skip_marker[N_MARKER_LENGTH];
bool skipping_text; // waiting until word count, sentence count,
or named marker is reached
int end_character_position;
int count_sentences;
int count_words;
int clause_start_char;
```

```
int clause_start_word;
bool new_sentence;
static int word_emphasis = 0; // set if emphasis level 3 or 4
static int embedded_flag = 0; // there are embedded commands to
be applied to the next phoneme, used in TranslateWord2()

static int prev_clause_pause = 0;
static int max_clause_pause = 0;
static bool any_stressed_words;
int pre_pause;
ALPHABET *current_alphabet;

// these were previously in translator class
char word_phonemes[N_WORD_PHONEMES]; // a word translated into
phoneme codes
int n_ph_list2;
PHONEME_LIST2 ph_list2[N_PHONEME_LIST]; // first stage of
text->phonemes

wchar_t option_punctlist[N_PUNCTLIST] = { 0 };
char ctrl_embedded = '\001'; // to allow an alternative CTRL for
embedded commands

// these are overridden by defaults set in the "speak" file
int option_linelength = 0;

#define N_EMBEDDED_LIST  250
static int embedded_ix;
static int embedded_read;
unsigned int embedded_list[N_EMBEDDED_LIST];

// the source text of a single clause (UTF8 bytes)
static char source[N_TR_SOURCE+40]; // extra space for embedded
command & voice change info at end

int n_replace_phonemes;
REPLACE_PHONEMES replace_phonemes[N_REPLACE_PHONEMES];
```

```
// brackets, also 0x2014 to 0x021f which don't need to be in this
list
static const unsigned short brackets[] = {
 '(', ')', '[', ']', '{', '}', '<', '>', '"', '\'', '`',
 0xab,   0xbb,   // double angle brackets
 0x300a, 0x300b, // double angle brackets (ideograph)
 0xe000+'<',     // private usage area
 0
};

// other characters which break a word, but don't produce a pause
static const unsigned short breaks[] = { '_', 0 };

// Tables of the relative lengths of vowels, depending on the
// type of the two phonemes that follow
// indexes are the "length_mod" value for the following phonemes

// use this table if vowel is not the last in the word
static unsigned char length_mods_en[100] = {
// a    ,    t    s    n    d    z    r    N    <- next
 100, 120, 100, 105, 100, 110, 110, 100,  95, 100, // a  <- next2
 105, 120, 105, 110, 125, 130, 135, 115, 125, 100, // ,
 105, 120,  75, 100,  75, 105, 120,  85,  75, 100, // t
 105, 120,  85, 105,  95, 115, 120, 100,  95, 100, // s
 110, 120,  95, 105, 100, 115, 120, 100, 100, 100, // n
 105, 120, 100, 105,  95, 115, 120, 110,  95, 100, // d
 105, 120, 100, 105, 105, 122, 125, 110, 105, 100, // z
 105, 120, 100, 105, 105, 122, 125, 110, 105, 100, // r
 105, 120,  95, 105, 100, 115, 120, 110, 100, 100, // N
 100, 120, 100, 100, 100, 100, 100, 100, 100, 100
};

// as above, but for the last syllable in a word
static unsigned char length_mods_en0[100] = {
// a    ,    t    s    n    d    z    r    N    <- next
 100, 150, 100, 105, 110, 115, 110, 110, 110, 100, // a  <- next2
```

```
 105, 150, 105, 110, 125, 135, 140, 115, 135, 100, // ,
 105, 150,  90, 105,  90, 122, 135, 100,  90, 100, // t
 105, 150, 100, 105, 100, 122, 135, 100, 100, 100, // s
 105, 150, 100, 105, 105, 115, 135, 110, 105, 100, // n
 105, 150, 100, 105, 105, 122, 130, 120, 125, 100, // d
 105, 150, 100, 105, 110, 122, 125, 115, 110, 100, // z
 105, 150, 100, 105, 105, 122, 135, 120, 105, 100, // r
 105, 150, 100, 105, 105, 115, 135, 110, 105, 100, // N
 100, 100, 100, 100, 100, 100, 100, 100, 100, 100
};

static unsigned char length_mods_equal[100] = {
// a    ,    t    s    n    d    z    r    N    <- next
 110, 120, 100, 110, 110, 110, 110, 110, 110, 110, // a  <- next2
 110, 120, 100, 110, 110, 110, 110, 110, 110, 110, // ,
 110, 120, 100, 110, 100, 110, 110, 110, 100, 110, // t
 110, 120, 100, 110, 110, 110, 110, 110, 110, 110, // s
 110, 120, 100, 110, 110, 110, 110, 110, 110, 110, // n
 110, 120, 100, 110, 110, 110, 110, 110, 110, 110, // d
 110, 120, 100, 110, 110, 110, 110, 110, 110, 110, // z
 110, 120, 100, 110, 110, 110, 110, 110, 110, 110, // r
 110, 120, 100, 110, 110, 110, 110, 110, 110, 110, // N
 110, 120, 100, 110, 110, 110, 110, 110, 110, 110
};

static unsigned char *length_mod_tabs[6] = {
 length_mods_en,
 length_mods_en,    // 1
 length_mods_en0,   // 2
 length_mods_equal, // 3
 length_mods_equal, // 4
 length_mods_equal  // 5
};

void SetLengthMods(Translator *tr, int value)
{
 int value2;
```

```
 tr->langopts.length_mods0 = tr->langopts.length_mods =
length_mod_tabs[value % 100];
 if ((value2 = value / 100) != 0)
  tr->langopts.length_mods0 = length_mod_tabs[value2];
}

int IsAlpha(unsigned int c)
{
 // Replacement for iswalph() which also checks for some in-word
symbols

 static const unsigned short extra_indic_alphas[] = {
  0xa70, 0xa71, // Gurmukhi: tippi, addak
  0
 };

 if (iswalpha(c))
  return 1;

 if (c < 0x300)
  return 0;

 if ((c >= 0x901) && (c <= 0xdf7)) {
  // Indic scripts: Devanagari, Tamil, etc
  if ((c & 0x7f) < 0x64)
   return 1;
  if (lookupwchar(extra_indic_alphas, c) != 0)
   return 1;
  if ((c >= 0xd7a) && (c <= 0xd7f))
   return 1; // malaytalam chillu characters

  return 0;
 }

 if ((c >= 0x5b0) && (c <= 0x5c2))
  return 1; // Hebrew vowel marks
```

```
 if (c == 0x0605)
  return 1;

 if ((c == 0x670) || ((c >= 0x64b) && (c <= 0x65e)))
  return 1; // arabic vowel marks

 if ((c >= 0x300) && (c <= 0x36f))
  return 1; // combining accents

 if ((c >= 0x780) && (c <= 0x7b1))
  return 1; // taani/divehi (maldives)

 if ((c >= 0xf40) && (c <= 0xfbc))
  return 1; // tibetan

 if ((c >= 0x1100) && (c <= 0x11ff))
  return 1; // Korean jamo

 if ((c >= 0x2800) && (c <= 0x28ff))
  return 1; // braille

 if ((c > 0x3040) && (c <= 0xa700))
  return 1; // Chinese/Japanese.  Should never get here, but Mac
OS 10.4's iswalpha seems to be broken, so just make sure

 return 0;
}

int IsDigit09(unsigned int c)
{
 if ((c >= '0') && (c <= '9'))
  return 1;
 return 0;
}

int IsDigit(unsigned int c)
```

```
{
 if (iswdigit(c))
  return 1;

 if ((c >= 0x966) && (c <= 0x96f))
  return 1;

 return 0;
}

static int IsSpace(unsigned int c)
{
 if (c == 0)
  return 0;
 if ((c >= 0x2500) && (c < 0x25a0))
  return 1; // box drawing characters
 if ((c >= 0xfff9) && (c <= 0xffff))
  return 1; // unicode specials
 return iswspace(c);
}

int isspace2(unsigned int c)
{
 // can't use isspace() because on Windows, isspace(0xe1) gives
TRUE !
 int c2;

 if (((c2 = (c & 0xff)) == 0) || (c > ' '))
  return 0;
 return 1;
}

void DeleteTranslator(Translator *tr)
{
 if (!tr) return;

 if (tr->data_dictlist != NULL)
```

```
  free(tr->data_dictlist);
 free(tr);
}

int lookupwchar(const unsigned short *list, int c)
{
 // Is the character c in the list ?
 int ix;

 for (ix = 0; list[ix] != 0; ix++) {
  if (list[ix] == c)
   return ix+1;
 }
 return 0;
}

int lookupwchar2(const unsigned short *list, int c)
{
 // Replace character c by another character.
 // Returns 0 = not found, 1 = delete character

 int ix;

 for (ix = 0; list[ix] != 0; ix += 2) {
  if (list[ix] == c)
   return list[ix+1];
 }
 return 0;
}

int IsBracket(int c)
{
 if ((c >= 0x2014) && (c <= 0x201f))
  return 1;
 return lookupwchar(brackets, c);
}
```

```
int utf8_nbytes(const char *buf)
{
 // Returns the number of bytes for the first UTF-8 character in
buf

 unsigned char c = (unsigned char)buf[0];
 if (c < 0x80)
  return 1;
 if (c < 0xe0)
  return 2;
 if (c < 0xf0)
  return 3;
 return 4;
}

int utf8_in2(int *c, const char *buf, int backwards)
{
 // Reads a unicode characater from a UTF8 string
 // Returns the number of UTF8 bytes used.
 // c: holds integer representation of multibyte character
 // buf: position of buffer is moved, if character is read
 // backwards: set if we are moving backwards through the UTF8
string

 int c1;
 int n_bytes;
 int ix;
 static const unsigned char mask[4] = { 0xff, 0x1f, 0x0f, 0x07 };

 // find the start of the next/previous character
 while ((*buf & 0xc0) == 0x80) {
  // skip over non-initial bytes of a multi-byte utf8 character
  if (backwards)
   buf--;
  else
   buf++;
 }
```

```
 n_bytes = 0;

 if ((c1 = *buf++) & 0x80) {
  if ((c1 & 0xe0) == 0xc0)
   n_bytes = 1;
  else if ((c1 & 0xf0) == 0xe0)
   n_bytes = 2;
  else if ((c1 & 0xf8) == 0xf0)
   n_bytes = 3;

  c1 &= mask[n_bytes];
  for (ix = 0; ix < n_bytes; ix++)
   c1 = (c1 << 6) + (*buf++ & 0x3f);
 }

 return n_bytes+1;
}

#pragma GCC visibility push(default)
int utf8_in(int *c, const char *buf)
{
 /* Read a unicode characater from a UTF8 string
  * Returns the number of UTF8 bytes used.
  * buf: position of buffer is moved, if character is read
  * c: holds UTF-16 representation of multibyte character by
  * skipping UTF-8 header bits of bytes in following way:
  * 2-byte character "ā":
  * hex           binary
  * c481          1100010010000001
  *    |          11000100  000001
  *    V             \    \ |    |
  * 0101          0000000100000001
  * 3-byte character " ":
  * ea9985 111010101001100110000101
  *            1010  011001  000101
  *    |       +  +--.\   \  |    |
```

```
  *    V        `--.  \`.  `.|    |
  *   A645         1010011001000101
  * 4-byte character " ":
  * f0a09c8e 11110000101000001001110010001110
  *    V          000  100000  011100  001110
  *   02070e         000000100000011100001110
  */
 return utf8_in2(c, buf, 0);
}
#pragma GCC visibility pop

int utf8_out(unsigned int c, char *buf)
{
 // write a UTF-16 character into a buffer as UTF-8
 // returns the number of bytes written

 int n_bytes;
 int j;
 int shift;
 static char unsigned code[4] = { 0, 0xc0, 0xe0, 0xf0 };

 if (c < 0x80) {
  buf[0] = c;
  return 1;
 }
 if (c >= 0x110000) {
  buf[0] = ' '; // out of range character code
  return 1;
 }
 if (c < 0x0800)
  n_bytes = 1;
 else if (c < 0x10000)
  n_bytes = 2;
 else
  n_bytes = 3;

 shift = 6*n_bytes;
```

```
 buf[0] = code[n_bytes] | (c >> shift);
 for (j = 0; j < n_bytes; j++) {
  shift -= 6;
  buf[j+1] = 0x80 + ((c >> shift) & 0x3f);
 }
 return n_bytes+1;
}

char *strchr_w(const char *s, int c)
{
 // return NULL for any non-ascii character
 if (c >= 0x80)
  return NULL;
 return strchr((char *)s, c); // (char *) is needed for Borland
compiler
}

static char *SpeakIndividualLetters(Translator *tr, char *word,
char *phonemes, int spell_word)
{
 int posn = 0;
 int capitals = 0;
 bool non_initial = false;

 if (spell_word > 2)
  capitals = 2; // speak 'capital'
 if (spell_word > 1)
  capitals |= 4; // speak charater code for unknown letters

 while ((*word != ' ') && (*word != 0)) {
  word += TranslateLetter(tr, word, phonemes, capitals |
non_initial, current_alphabet);
  posn++;
  non_initial = true;
  if (phonemes[0] == phonSWITCH) {
   // change to another language in order to translate this word
   strcpy(word_phonemes, phonemes);
```

```
   return NULL;
  }
 }
 SetSpellingStress(tr, phonemes, spell_word, posn);
 return word;
}

static int CheckDottedAbbrev(char *word1)
{
 int wc;
 int count = 0;
 int nbytes;
 int ok;
 int ix;
 char *word;
 char *wbuf;
 char word_buf[80];

 word = word1;
 wbuf = word_buf;

 for (;;) {
  ok = 0;
  nbytes = utf8_in(&wc, word);
  if ((word[nbytes] == ' ') && IsAlpha(wc)) {
   if (word[nbytes+1] == '.') {
    if (word[nbytes+2] == ' ')
     ok = 1;
    else if (word[nbytes+2] == '\'') {
     nbytes += 2; // delete the final dot (eg. u.s.a.'s)
     ok = 2;
    }
   } else if ((count > 0) && (word[nbytes] == ' '))
    ok = 2;
  }

  if (ok == 0)
```

```
   break;

  for (ix = 0; ix < nbytes; ix++)
   *wbuf++ = word[ix];

  count++;

  if (ok == 2) {
   word += nbytes;
   break;
  }

  word += (nbytes + 3);
 }

 if (count > 1) {
  ix = wbuf - word_buf;
  memcpy(word1, word_buf, ix);
  while (&word1[ix] < word)
   word1[ix++] = ' ';
  dictionary_skipwords = (count - 1)*2;
 }
 return count;
}

static int TranslateWord3(Translator *tr, char *word_start,
WORD_TAB *wtab, char *word_out)
{
 // word1 is terminated by space (0x20) character

 char *word1;
 int word_length;
 int ix;
 char *p;
 int pfix;
 int n_chars;
 unsigned int dictionary_flags[2];
```

```
 unsigned int dictionary_flags2[2];
 int end_type = 0;
 int end_type1 = 0;
 int prefix_type = 0;
 int prefix_stress;
 char *wordx;
 char phonemes[N_WORD_PHONEMES];
 char phonemes2[N_WORD_PHONEMES];
 char prefix_phonemes[N_WORD_PHONEMES];
 char unpron_phonemes[N_WORD_PHONEMES];
 char end_phonemes[N_WORD_PHONEMES];
 char end_phonemes2[N_WORD_PHONEMES];
 char word_copy[N_WORD_BYTES];
 char word_copy2[N_WORD_BYTES];
 int word_copy_length;
 char prefix_chars[0x3f + 2];
 bool found = false;
 int end_flags;
 int c_temp; // save a character byte while we temporarily
replace it with space
 int first_char;
 int last_char = 0;
 int prefix_flags = 0;
 bool more_suffixes;
 bool confirm_prefix;
 int spell_word;
 int emphasize_allcaps = 0;
 int wflags;
 int was_unpronouncable = 0;
 int loopcount;
 int add_suffix_phonemes = 0;
 WORD_TAB wtab_null[8];

 // translate these to get pronunciations of plural 's' suffix
(different forms depending on
 // the preceding letter
 static char word_zz[4] = { 0, 'z', 'z', 0 };
```

```
 static char word_iz[4] = { 0, 'i', 'z', 0 };
 static char word_ss[4] = { 0, 's', 's', 0 };

 if (wtab == NULL) {
  memset(wtab_null, 0, sizeof(wtab_null));
  wtab = wtab_null;
 }
 wflags = wtab->flags;

 dictionary_flags[0] = 0;
 dictionary_flags[1] = 0;
 dictionary_flags2[0] = 0;
 dictionary_flags2[1] = 0;
 dictionary_skipwords = 0;

 phonemes[0] = 0;
 unpron_phonemes[0] = 0;
 prefix_phonemes[0] = 0;
 end_phonemes[0] = 0;

 if (tr->data_dictlist == NULL) {
  // dictionary is not loaded
  word_phonemes[0] = 0;
  return 0;
 }

 // count the length of the word
 word1 = word_start;
 if (*word1 == ' ') word1++; // possibly a dot was replaced by
space:  $dot
 wordx = word1;

 utf8_in(&first_char, wordx);
 word_length = 0;
 while ((*wordx != 0) && (*wordx != ' ')) {
  wordx += utf8_in(&last_char, wordx);
  word_length++;
```

```
 }

 word_copy_length = wordx - word_start;
 if (word_copy_length >= N_WORD_BYTES)
  word_copy_length = N_WORD_BYTES-1;
 memcpy(word_copy2, word_start, word_copy_length);

 spell_word = 0;

 if ((word_length == 1) && (wflags & FLAG_TRANSLATOR2)) {
  // retranslating a 1-character word using a different language,
say its name
  utf8_in(&c_temp, wordx+1); // the next character
  if (!IsAlpha(c_temp) || (AlphabetFromChar(last_char) !=
AlphabetFromChar(c_temp)))
   spell_word = 1;
 }

 if (option_sayas == SAYAS_KEY) {
  if (word_length == 1)
   spell_word = 4;
  else {
   // is there a translation for this keyname ?
   word1--;
   *word1 = '_'; // prefix keyname with '_'
   found = LookupDictList(tr, &word1, phonemes, dictionary_flags,
0, wtab);
  }
 }

 // try an initial lookup in the dictionary list, we may find a
pronunciation specified, or
 // we may just find some flags
 if (option_sayas & 0x10) {
  // SAYAS_CHAR, SAYAS_GYLPH, or SAYAS_SINGLE_CHAR
  spell_word = option_sayas & 0xf; // 2,3,4
 } else {
```

```
  if (!found)
   found = LookupDictList(tr, &word1, phonemes, dictionary_flags,
FLAG_ALLOW_TEXTMODE, wtab);   // the original word

  if ((dictionary_flags[0] & (FLAG_ALLOW_DOT | FLAG_NEEDS_DOT))
&& (wordx[1] == '.'))
   wordx[1] = ' '; // remove a Dot after this word

  if (dictionary_flags[0] & FLAG_TEXTMODE) {
   if (word_out != NULL)
    strcpy(word_out, word1);

   return dictionary_flags[0];
  } else if ((found == false) && (dictionary_flags[0] &
FLAG_SKIPWORDS) && !(dictionary_flags[0] & FLAG_ABBREV)) {
   // grouped words, but no translation.  Join the words with
hyphens.
   wordx = word1;
   ix = 0;
   while (ix < dictionary_skipwords) {
    if (*wordx == ' ') {
     *wordx = '-';
     ix++;
    }
    wordx++;
   }
  }

  if ((word_length == 1) && (dictionary_skipwords == 0)) {
   // is this a series of single letters separated by dots?
   if (CheckDottedAbbrev(word1)) {
    dictionary_flags[0] = 0;
    dictionary_flags[1] = 0;
    spell_word = 1;
    if (dictionary_skipwords)
     dictionary_flags[0] = FLAG_SKIPWORDS;
   }
```

```
  }

  if (phonemes[0] == phonSWITCH) {
   // change to another language in order to translate this word
   strcpy(word_phonemes, phonemes);
   return 0;
  }

  if (!found && (dictionary_flags[0] & FLAG_ABBREV)) {
   // the word has $abbrev flag, but no pronunciation specified.
Speak as individual letters
   spell_word = 1;
  }

  if (!found && iswdigit(first_char)) {
   Lookup(tr, "_0lang", word_phonemes);
   if (word_phonemes[0] == phonSWITCH)
    return 0;

   if ((tr->langopts.numbers2 & NUM2_ENGLISH_NUMERALS) &&
!(wtab->flags & FLAG_CHAR_REPLACED)) {
    // for this language, speak English numerals (0-9) with the
English voice
    sprintf(word_phonemes, "%c", phonSWITCH);
    return 0;
   }

   found = TranslateNumber(tr, word1, phonemes, dictionary_flags,
wtab, 0);
  }

  if (!found && ((wflags & FLAG_UPPERS) != FLAG_FIRST_UPPER)) {
   // either all upper or all lower case

   if ((tr->langopts.numbers & NUM_ROMAN) ||
((tr->langopts.numbers & NUM_ROMAN_CAPITALS) && (wflags &
FLAG_ALL_UPPER))) {
```

```
    if ((wflags & FLAG_LAST_WORD) || !(wtab[1].flags &
FLAG_NOSPACE)) {
     // don't use Roman number if this word is not separated from
the next word (eg. "XLTest")
     if ((found = TranslateRoman(tr, word1, phonemes, wtab)) !=
0)
      dictionary_flags[0] |= FLAG_ABBREV; // prevent emphasis if
capitals
    }
   }
  }

  if ((wflags & FLAG_ALL_UPPER) && (word_length > 1) &&
iswalpha(first_char)) {
   if ((option_tone_flags & OPTION_EMPHASIZE_ALLCAPS) &&
!(dictionary_flags[0] & FLAG_ABBREV)) {
    // emphasize words which are in capitals
    emphasize_allcaps = FLAG_EMPHASIZED;
   } else if (!found && !(dictionary_flags[0] &  FLAG_SKIPWORDS)
&& (word_length < 4) && (tr->clause_lower_count > 3)
              && (tr->clause_upper_count <=
tr->clause_lower_count)) {
    // An upper case word in a lower case clause. This could be
an abbreviation.
    spell_word = 1;
   }
  }
 }

 if (spell_word > 0) {
  // Speak as individual letters
  phonemes[0] = 0;

  if (SpeakIndividualLetters(tr, word1, phonemes, spell_word) ==
NULL) {
   if (word_length > 1)
    return FLAG_SPELLWORD; // a mixture of languages, retranslate
```

```
as individual letters, separated by spaces
   return 0;
  }
  strcpy(word_phonemes, phonemes);
  if (wflags & FLAG_TRANSLATOR2)
   return 0;
  return dictionary_flags[0] & FLAG_SKIPWORDS; // for "b.c.d"
 } else if (found == false) {
  // word's pronunciation is not given in the dictionary list,
although
  // dictionary_flags may have ben set there

  int posn;
  bool non_initial = false;
  int length;

  posn = 0;
  length = 999;
  wordx = word1;

  while (((length < 3) && (length > 0)) || (word_length > 1 &&
Unpronouncable(tr, wordx, posn))) {
   // This word looks "unpronouncable", so speak letters
individually until we
   // find a remainder that we can pronounce.
   was_unpronouncable = FLAG_WAS_UNPRONOUNCABLE;
   emphasize_allcaps = 0;

   if (wordx[0] == '\'')
    break;

   if (posn > 0)
    non_initial = true;

   wordx += TranslateLetter(tr, wordx, unpron_phonemes,
non_initial, current_alphabet);
   posn++;
```

```
   if (unpron_phonemes[0] == phonSWITCH) {
    // change to another language in order to translate this word
    strcpy(word_phonemes, unpron_phonemes);
    if (strcmp(&unpron_phonemes[1], "en") == 0)
     return FLAG_SPELLWORD; // _^_en must have been set in
TranslateLetter(), not *_rules which uses only _^_
    return 0;
   }

   length = 0;
   while (wordx[length] != ' ') length++;
  }
  SetSpellingStress(tr, unpron_phonemes, 0, posn);

  // anything left ?
  if (*wordx != ' ') {
   if ((unpron_phonemes[0] != 0) && (wordx[0] != '\'')) {
    // letters which have been spoken individually from affecting
the pronunciation of the pronuncable part
    wordx[-1] = ' ';
   }

   // Translate the stem
   end_type = TranslateRules(tr, wordx, phonemes,
N_WORD_PHONEMES, end_phonemes, wflags, dictionary_flags);

   if (phonemes[0] == phonSWITCH) {
    // change to another language in order to translate this word
    strcpy(word_phonemes, phonemes);
    return 0;
   }

   if ((phonemes[0] == 0) && (end_phonemes[0] == 0)) {
    int wc;
    // characters not recognised, speak them individually
    // ?? should we say super/sub-script numbers and letters
here?
```

```
    utf8_in(&wc, wordx);
    if ((word_length == 1) && (IsAlpha(wc) || IsSuperscript(wc)))
{
     if ((wordx = SpeakIndividualLetters(tr, wordx, phonemes,
spell_word)) == NULL)
      return 0;
     strcpy(word_phonemes, phonemes);
     return 0;
    }
   }

   c_temp = wordx[-1];

   found = false;
   confirm_prefix = true;
   for (loopcount = 0; (loopcount < 50) && (end_type & SUFX_P);
loopcount++) {
    // Found a standard prefix, remove it and retranslate
    // loopcount guards against an endless loop
    if (confirm_prefix && !(end_type & SUFX_B)) {
     int end2;
     char end_phonemes2[N_WORD_PHONEMES];

     // remove any standard suffix and confirm that the prefix is
still recognised
     phonemes2[0] = 0;
     end2 = TranslateRules(tr, wordx, phonemes2, N_WORD_PHONEMES,
end_phonemes2, wflags|FLAG_NO_PREFIX|FLAG_NO_TRACE,
dictionary_flags);
     if (end2) {
      RemoveEnding(tr, wordx, end2, word_copy);
      end_type = TranslateRules(tr, wordx, phonemes,
N_WORD_PHONEMES, end_phonemes, wflags|FLAG_NO_TRACE,
dictionary_flags);
      memcpy(wordx, word_copy, strlen(word_copy));
      if ((end_type & SUFX_P) == 0) {
       // after removing the suffix, the prefix is no longer
```

```
recognised.
       // Keep the suffix, but don't use the prefix
       end_type = end2;
       strcpy(phonemes, phonemes2);
       strcpy(end_phonemes, end_phonemes2);
       if (option_phonemes & espeakPHONEMES_TRACE) {
        DecodePhonemes(end_phonemes, end_phonemes2);
        fprintf(f_trans, "  suffix [%s]\n\n", end_phonemes2);
       }
      }
      confirm_prefix = false;
      continue;
     }
    }

    prefix_type = end_type;

    if (prefix_type & SUFX_V)
     tr->expect_verb = 1; // use the verb form of the word

    wordx[-1] = c_temp;

    if ((prefix_type & SUFX_B) == 0) {
     for (ix = (prefix_type & 0xf); ix > 0; ix--) { // num. of
characters to remove
      wordx++;
      while ((*wordx & 0xc0) == 0x80) wordx++; // for multibyte
characters
     }
    } else {
     pfix = 1;
     prefix_chars[0] = 0;
     n_chars = prefix_type & 0x3f;

     for (ix = 0; ix < n_chars; ix++) { // num. of bytes to
remove
      prefix_chars[pfix++] = *wordx++;
```

```
      if ((prefix_type & SUFX_B) && (ix == (n_chars-1)))
       prefix_chars[pfix-1] = 0; // discard the last character of
the prefix, this is the separator character
     }
     prefix_chars[pfix] = 0;
    }
    c_temp = wordx[-1];
    wordx[-1] = ' ';
    confirm_prefix = true;
    wflags |= FLAG_PREFIX_REMOVED;

    if (prefix_type & SUFX_B) {
     // SUFX_B is used for Turkish, tr_rules contains " ' (Pb"
     // examine the prefix part
     char *wordpf;
     char prefix_phonemes2[12];

     strncpy0(prefix_phonemes2, end_phonemes,
sizeof(prefix_phonemes2));
     wordpf = &prefix_chars[1];
     strcpy(prefix_phonemes, phonemes);

     // look for stress marker or $abbrev
     found = LookupDictList(tr, &wordpf, phonemes,
dictionary_flags, 0, wtab);
     if (found)
      strcpy(prefix_phonemes, phonemes);
     if (dictionary_flags[0] & FLAG_ABBREV) {
      prefix_phonemes[0] = 0;
      SpeakIndividualLetters(tr, wordpf, prefix_phonemes, 1);
     }
    } else
     strcat(prefix_phonemes, end_phonemes);
    end_phonemes[0] = 0;

    end_type = 0;
```

```
    found = LookupDictList(tr, &wordx, phonemes,
dictionary_flags2, SUFX_P, wtab); // without prefix
    if (dictionary_flags[0] == 0) {
     dictionary_flags[0] = dictionary_flags2[0];
     dictionary_flags[1] = dictionary_flags2[1];
    } else
     prefix_flags = 1;
    if (found == false) {
     end_type = TranslateRules(tr, wordx, phonemes,
N_WORD_PHONEMES, end_phonemes, wflags & (FLAG_HYPHEN_AFTER |
FLAG_PREFIX_REMOVED), dictionary_flags);

     if (phonemes[0] == phonSWITCH) {
      // change to another language in order to translate this
word
      wordx[-1] = c_temp;
      strcpy(word_phonemes, phonemes);
      return 0;
     }
    }
   }

   if ((end_type != 0) && !(end_type & SUFX_P)) {
    end_type1 = end_type;
    strcpy(phonemes2, phonemes);

    // The word has a standard ending, re-translate without this
ending
    end_flags = RemoveEnding(tr, wordx, end_type, word_copy);
    more_suffixes = true;

    while (more_suffixes) {
     more_suffixes = false;
     phonemes[0] = 0;

     if (prefix_phonemes[0] != 0) {
      // lookup the stem without the prefix removed
```

```
      wordx[-1] = c_temp;
      found = LookupDictList(tr, &word1, phonemes,
dictionary_flags2, end_flags, wtab);  // include prefix, but not
suffix
      wordx[-1] = ' ';
      if (phonemes[0] == phonSWITCH) {
       // change to another language in order to translate this
word
       memcpy(wordx, word_copy, strlen(word_copy));
       strcpy(word_phonemes, phonemes);
       return 0;
      }
      if (dictionary_flags[0] == 0) {
       dictionary_flags[0] = dictionary_flags2[0];
       dictionary_flags[1] = dictionary_flags2[1];
      }
      if (found)
       prefix_phonemes[0] = 0; // matched whole word, don't need
prefix now

      if ((found == false) && (dictionary_flags2[0] != 0))
       prefix_flags = 1;
     }
     if (found == false) {
      found = LookupDictList(tr, &wordx, phonemes,
dictionary_flags2, end_flags, wtab);  // without prefix and
suffix
      if (phonemes[0] == phonSWITCH) {
       // change to another language in order to translate this
word
       memcpy(wordx, word_copy, strlen(word_copy));
       strcpy(word_phonemes, phonemes);
       return 0;
      }

      if (dictionary_flags[0] == 0) {
       dictionary_flags[0] = dictionary_flags2[0];
```

```
        dictionary_flags[1] = dictionary_flags2[1];
       }
      }
      if (found == false) {
       if (end_type & SUFX_Q) {
        // don't retranslate, use the original lookup result
        strcpy(phonemes, phonemes2);
       } else {
        if (end_flags & FLAG_SUFX)
         wflags |= FLAG_SUFFIX_REMOVED;
        if (end_type & SUFX_A)
         wflags |= FLAG_SUFFIX_VOWEL;

        if (end_type & SUFX_M) {
         // allow more suffixes before this suffix
         strcpy(end_phonemes2, end_phonemes);
         end_type = TranslateRules(tr, wordx, phonemes,
N_WORD_PHONEMES, end_phonemes, wflags, dictionary_flags);
         strcat(end_phonemes, end_phonemes2); // add the phonemes
for the previous suffixes after this one

         if ((end_type != 0) && !(end_type & SUFX_P)) {
          // there is another suffix
          end_flags = RemoveEnding(tr, wordx, end_type, NULL);
          more_suffixes = true;
         }
        } else {
         // don't remove any previous suffix
         TranslateRules(tr, wordx, phonemes, N_WORD_PHONEMES,
NULL, wflags, dictionary_flags);
         end_type = 0;
        }

        if (phonemes[0] == phonSWITCH) {
         // change to another language in order to translate this
word
         strcpy(word_phonemes, phonemes);
```

```
        memcpy(wordx, word_copy, strlen(word_copy));
        wordx[-1] = c_temp;
        return 0;
       }
      }
     }
    }

    if ((end_type1 & SUFX_T) == 0) {
     // the default is to add the suffix and then determine the
word's stress pattern
     AppendPhonemes(tr, phonemes, N_WORD_PHONEMES, end_phonemes);
     end_phonemes[0] = 0;
    }
    memcpy(wordx, word_copy, strlen(word_copy));
   }

   wordx[-1] = c_temp;
  }
 }

 if (wflags & FLAG_HAS_PLURAL) {
  // s or 's suffix, append [s], [z] or [Iz] depending on
previous letter
  if (last_char == 'f')
   TranslateRules(tr, &word_ss[1], phonemes, N_WORD_PHONEMES,
NULL, 0, NULL);
  else if ((last_char == 0) || (strchr_w("hsx", last_char) ==
NULL))
   TranslateRules(tr, &word_zz[1], phonemes, N_WORD_PHONEMES,
NULL, 0, NULL);
  else
   TranslateRules(tr, &word_iz[1], phonemes, N_WORD_PHONEMES,
NULL, 0, NULL);
 }

 wflags |= emphasize_allcaps;
```

```
 // determine stress pattern for this word

 add_suffix_phonemes = 0;
 if (end_phonemes[0] != 0)
  add_suffix_phonemes = 2;

 prefix_stress = 0;
 for (p = prefix_phonemes; *p != 0; p++) {
  if ((*p == phonSTRESS_P) || (*p == phonSTRESS_P2))
   prefix_stress = *p;
 }
 if (prefix_flags || (prefix_stress != 0)) {
  if ((tr->langopts.param[LOPT_PREFIXES]) || (prefix_type &
SUFX_T)) {
   char *p;
   // German, keep a secondary stress on the stem
   SetWordStress(tr, phonemes, dictionary_flags, 3, 0);

   // reduce all but the first primary stress
   ix = 0;
   for (p = prefix_phonemes; *p != 0; p++) {
    if (*p == phonSTRESS_P) {
     if (ix == 0)
      ix = 1;
     else
      *p = phonSTRESS_3;
    }
   }
   snprintf(word_phonemes, sizeof(word_phonemes), "%s%s%s",
unpron_phonemes, prefix_phonemes, phonemes);
   word_phonemes[N_WORD_PHONEMES-1] = 0;
   SetWordStress(tr, word_phonemes, dictionary_flags, -1, 0);
  } else {
   // stress position affects the whole word, including prefix
   snprintf(word_phonemes, sizeof(word_phonemes), "%s%s%s",
unpron_phonemes, prefix_phonemes, phonemes);
```

```
   word_phonemes[N_WORD_PHONEMES-1] = 0;
   SetWordStress(tr, word_phonemes, dictionary_flags, -1, 0);
  }
 } else {
  SetWordStress(tr, phonemes, dictionary_flags, -1,
add_suffix_phonemes);
  snprintf(word_phonemes, sizeof(word_phonemes), "%s%s%s",
unpron_phonemes, prefix_phonemes, phonemes);
  word_phonemes[N_WORD_PHONEMES-1] = 0;
 }

 if (end_phonemes[0] != 0) {
  // a suffix had the SUFX_T option set, add the suffix after the
stress pattern has been determined
  ix = strlen(word_phonemes);
  end_phonemes[N_WORD_PHONEMES-1-ix] = 0; // ensure no buffer
overflow
  strcpy(&word_phonemes[ix], end_phonemes);
 }

 if (wflags & FLAG_LAST_WORD) {
  // don't use $brk pause before the last word of a sentence
  // (but allow it for emphasis, see below
  dictionary_flags[0] &= ~FLAG_PAUSE1;
 }

 if ((wflags & FLAG_HYPHEN) && (tr->langopts.stress_flags &
S_HYPEN_UNSTRESS))
  ChangeWordStress(tr, word_phonemes, 3);
 else if (wflags & FLAG_EMPHASIZED2) {
  // A word is indicated in the source text as stressed
  // Give it stress level 6 (for the intonation module)
  ChangeWordStress(tr, word_phonemes, 6);

  if (wflags & FLAG_EMPHASIZED)
   dictionary_flags[0] |= FLAG_PAUSE1; // precede by short pause
 } else if (wtab[dictionary_skipwords].flags & FLAG_LAST_WORD) {
```

```
  // the word has attribute to stress or unstress when at end of
clause
  if (dictionary_flags[0] & (FLAG_STRESS_END | FLAG_STRESS_END2))
   ChangeWordStress(tr, word_phonemes, 4);
  else if ((dictionary_flags[0] & FLAG_UNSTRESS_END) &&
(any_stressed_words))
   ChangeWordStress(tr, word_phonemes, 3);
 }

 // dictionary flags for this word give a clue about which
alternative pronunciations of
 // following words to use.
 if (end_type1 & SUFX_F) {
  // expect a verb form, with or without -s suffix
  tr->expect_verb = 2;
  tr->expect_verb_s = 2;
 }

 if (dictionary_flags[1] & FLAG_PASTF) {
  // expect perfect tense in next two words
  tr->expect_past = 3;
  tr->expect_verb = 0;
  tr->expect_noun = 0;
 } else if (dictionary_flags[1] & FLAG_VERBF) {
  // expect a verb in the next word
  tr->expect_verb = 2;
  tr->expect_verb_s = 0; // verb won't have -s suffix
  tr->expect_noun = 0;
 } else if (dictionary_flags[1] & FLAG_VERBSF) {
  // expect a verb, must have a -s suffix
  tr->expect_verb = 0;
  tr->expect_verb_s = 2;
  tr->expect_past = 0;
  tr->expect_noun = 0;
 } else if (dictionary_flags[1] & FLAG_NOUNF) {
  // not expecting a verb next
  tr->expect_noun = 2;
```

```
  tr->expect_verb = 0;
  tr->expect_verb_s = 0;
  tr->expect_past = 0;
 }

 if ((wordx[0] != 0) && (!(dictionary_flags[1] & FLAG_VERB_EXT)))
{
  if (tr->expect_verb > 0)
   tr->expect_verb--;

  if (tr->expect_verb_s > 0)
   tr->expect_verb_s--;

  if (tr->expect_noun > 0)
   tr->expect_noun--;

  if (tr->expect_past > 0)
   tr->expect_past--;
 }

 if ((word_length == 1) && (tr->translator_name == L('e', 'n'))
&& iswalpha(first_char) && (first_char != 'i')) {
  // English Specific !!!!
  // any single letter before a dot is an abbreviation, except
'I'
  dictionary_flags[0] |= FLAG_ALLOW_DOT;
 }

 if ((tr->langopts.param[LOPT_ALT] & 2) && ((dictionary_flags[0]
& (FLAG_ALT_TRANS | FLAG_ALT2_TRANS)) != 0))
  ApplySpecialAttribute2(tr, word_phonemes, dictionary_flags[0]);

 dictionary_flags[0] |= was_unpronouncable;
 memcpy(word_start, word_copy2, word_copy_length);
 return dictionary_flags[0];
}
```

```
int TranslateWord(Translator *tr, char *word_start, WORD_TAB
*wtab, char *word_out)
{
 char words_phonemes[N_WORD_PHONEMES]; // a word translated into
phoneme codes
 char *phonemes = words_phonemes;
 int available = N_WORD_PHONEMES;
 bool first_word = true;

 int flags = TranslateWord3(tr, word_start, wtab, word_out);
 if (flags & FLAG_TEXTMODE && word_out) {
  // Ensure that start of word rules match with the replaced
text,
  // so that emoji and other characters are pronounced correctly.
  char word[N_WORD_BYTES+1];
  word[0] = 0;
  word[1] = ' ';
  strcpy(word+2, word_out);
  word_out = word+2;

  while (*word_out && available > 1) {
   int c;
   utf8_in(&c, word_out);
   if (iswupper(c)) {
    wtab->flags |= FLAG_FIRST_UPPER;
    utf8_out(tolower(c), word_out);
   } else {
    wtab->flags &= ~FLAG_FIRST_UPPER;
   }

   TranslateWord3(tr, word_out, wtab, NULL);

   int n;
   if (first_word) {
    n = snprintf(phonemes, available, "%s", word_phonemes);
    first_word = false;
   } else {
```

```
    n = snprintf(phonemes, available, "%c%s", phonEND_WORD,
word_phonemes);
   }

   available -= n;
   phonemes += n;

   // skip to the next word in a multi-word replacement. Always
skip at least one word.
   for (dictionary_skipwords++; dictionary_skipwords > 0;
dictionary_skipwords--) {
    while (!isspace(*word_out)) ++word_out;
    while (isspace(*word_out))  ++word_out;
   }
  }

  // If the list file contains a text replacement to another
  // entry in the list file, e.g.:
  //     ripost     riposte $text
  //     riposte    rI#p0st
  // calling it from a prefix or suffix rule such as 'riposted'
  // causes word_out[0] to be NULL, as TranslateWord3 has the
  // information needed to perform the mapping. In this case,
  // no phonemes have been written in this loop and the phonemes
  // have been calculated, so don't override them.
  if (phonemes != words_phonemes) {
   snprintf(word_phonemes, sizeof(word_phonemes), "%s",
words_phonemes);
  }
 }
 return flags;
}

static void SetPlist2(PHONEME_LIST2 *p, unsigned char phcode)
{
 p->phcode = phcode;
 p->stresslevel = 0;
```

```
 p->tone_ph = 0;
 p->synthflags = embedded_flag;
 p->sourceix = 0;
 embedded_flag = 0;
}

static int CountSyllables(unsigned char *phonemes)
{
 int count = 0;
 int phon;
 while ((phon = *phonemes++) != 0) {
  if (phoneme_tab[phon]->type == phVOWEL)
   count++;
 }
 return count;
}

static void Word_EmbeddedCmd()
{
 // Process embedded commands for emphasis, sayas, and break
 int embedded_cmd;
 int value;

 do {
  embedded_cmd = embedded_list[embedded_read++];
  value = embedded_cmd >> 8;

  switch (embedded_cmd & 0x1f)
  {
  case EMBED_Y:
   option_sayas = value;
   break;

  case EMBED_F:
   option_emphasis = value;
   break;
```

```
  case EMBED_B:
   // break command
   if (value == 0)
    pre_pause = 0; // break=none
   else
    pre_pause += value;
   break;
  }
 } while (((embedded_cmd & 0x80) == 0) && (embedded_read <
embedded_ix));
}

int SetTranslator2(const char *new_language)
{
 // Set translator2 to a second language
 int new_phoneme_tab;

 if ((new_phoneme_tab = SelectPhonemeTableName(new_language)) >=
0) {
  if ((translator2 != NULL) && (strcmp(new_language,
translator2_language) != 0)) {
   // we already have an alternative translator, but not for the
required language, delete it
   DeleteTranslator(translator2);
   translator2 = NULL;
  }

  if (translator2 == NULL) {
   translator2 = SelectTranslator(new_language);
   strcpy(translator2_language, new_language);

   if (LoadDictionary(translator2, translator2->dictionary_name,
0) != 0) {
    SelectPhonemeTable(voice->phoneme_tab_ix); // revert to
original phoneme table
    new_phoneme_tab = -1;
    translator2_language[0] = 0;
```

```
   }
   translator2->phoneme_tab_ix = new_phoneme_tab;
  }
 }
 if (translator2 != NULL)
  translator2->phonemes_repeat[0] = 0;
 return new_phoneme_tab;
}

static int TranslateWord2(Translator *tr, char *word, WORD_TAB
*wtab, int pre_pause)
{
 int flags = 0;
 int stress;
 int next_stress;
 int next_tone = 0;
 unsigned char *p;
 int srcix;
 int found_dict_flag;
 unsigned char ph_code;
 PHONEME_LIST2 *plist2;
 PHONEME_TAB *ph;
 int max_stress;
 int max_stress_ix = 0;
 int prev_vowel = -1;
 int pitch_raised = 0;
 int switch_phonemes = -1;
 bool first_phoneme = true;
 int source_ix;
 int len;
 int ix;
 int sylimit; // max. number of syllables in a word to be
combined with a preceding preposition
 const char *new_language;
 int bad_phoneme;
 int word_flags;
 int word_copy_len;
```

```
 char word_copy[N_WORD_BYTES+1];
 char word_replaced[N_WORD_BYTES+1];
 char old_dictionary_name[40];

 len = wtab->length;
 if (len > 31) len = 31;
 source_ix = (wtab->sourceix & 0x7ff) | (len << 11); // bits 0-10
sourceix, bits 11-15 word length

 word_flags = wtab[0].flags;
 if (word_flags & FLAG_EMBEDDED) {
  wtab[0].flags &= ~FLAG_EMBEDDED; // clear it in case we call
TranslateWord2() again for the same word
  embedded_flag = SFLAG_EMBEDDED;

  Word_EmbeddedCmd();
 }

 if ((word[0] == 0) || (word_flags & FLAG_DELETE_WORD)) {
  // nothing to translate.  Add a dummy phoneme to carry any
embedded commands
  if (embedded_flag) {
   ph_list2[n_ph_list2].phcode = phonEND_WORD;
   ph_list2[n_ph_list2].stresslevel = 0;
   ph_list2[n_ph_list2].wordstress = 0;
   ph_list2[n_ph_list2].tone_ph = 0;
   ph_list2[n_ph_list2].synthflags = embedded_flag;
   ph_list2[n_ph_list2].sourceix = 0;
   n_ph_list2++;
   embedded_flag = 0;
  }
  word_phonemes[0] = 0;
  return 0;
 }

 // after a $pause word attribute, ignore a $pause attribute on
the next two words
```

```
 if (tr->prepause_timeout > 0)
  tr->prepause_timeout--;

 if ((option_sayas & 0xf0) == 0x10) {
  if (!(word_flags & FLAG_FIRST_WORD)) {
   // SAYAS_CHARS, SAYAS_GLYPHS, or SAYAS_SINGLECHARS.  Pause
between each word.
   pre_pause += 4;
  }
 }

 if (word_flags & FLAG_FIRST_UPPER) {
  if ((option_capitals > 2) && (embedded_ix < N_EMBEDDED_LIST-6))
{
   // indicate capital letter by raising pitch
   if (embedded_flag)
    embedded_list[embedded_ix-1] &= ~0x80; // already embedded
command before this word, remove terminator
   if ((pitch_raised = option_capitals) == 3)
    pitch_raised = 20; // default pitch raise for capitals
   embedded_list[embedded_ix++] = EMBED_P+0x40+0x80 +
(pitch_raised << 8); // raise pitch
   embedded_flag = SFLAG_EMBEDDED;
  }
 }

 p = (unsigned char *)word_phonemes;
 if (word_flags & FLAG_PHONEMES) {
  // The input is in phoneme mnemonics, not language text
  int c1;
  char lang_name[12];

  if (memcmp(word, "_^_", 3) == 0) {
   // switch languages
   word += 3;
   for (ix = 0;;) {
    c1 = *word++;
```

```
    if ((c1 == ' ') || (c1 == 0))
     break;
    lang_name[ix++] = tolower(c1);
   }
   lang_name[ix] = 0;

   if ((ix = LookupPhonemeTable(lang_name)) > 0) {
    SelectPhonemeTable(ix);
    word_phonemes[0] = phonSWITCH;
    word_phonemes[1] = ix;
    word_phonemes[2] = 0;
   }
  } else
   EncodePhonemes(word, word_phonemes, &bad_phoneme);
  flags = FLAG_FOUND;
 } else {
  int c2;
  ix = 0;
  while (((c2 = word_copy[ix] = word[ix]) != ' ') && (c2 != 0) &&
(ix < N_WORD_BYTES)) ix++;
  word_copy_len = ix;

  word_replaced[2] = 0;
  flags = TranslateWord(translator, word, wtab,
&word_replaced[2]);

  if (flags & FLAG_SPELLWORD) {
   // re-translate the word as individual letters, separated by
spaces
   memcpy(word, word_copy, word_copy_len);
   return flags;
  }

  if ((flags & FLAG_COMBINE) && !(wtab[1].flags & FLAG_PHONEMES))
{
   char *p2;
   bool ok = true;
```

```
   unsigned int flags2[2];
   int c_word2;
   char ph_buf[N_WORD_PHONEMES];

   flags2[0] = 0;
   sylimit = tr->langopts.param[LOPT_COMBINE_WORDS];

   // LANG=cs,sk
   // combine a preposition with the following word
   p2 = word;
   while (*p2 != ' ') p2++;

   utf8_in(&c_word2, p2+1); // first character of the next word;
   if (!iswalpha(c_word2))
    ok = false;

   if (ok == true) {
    strcpy(ph_buf, word_phonemes);

    flags2[0] = TranslateWord(translator, p2+1, wtab+1, NULL);
    if ((flags2[0] & FLAG_WAS_UNPRONOUNCABLE) ||
(word_phonemes[0] == phonSWITCH))
     ok = false;

    if (sylimit & 0x100) {
     // only if the second word has $alt attribute
     if ((flags2[0] & FLAG_ALT_TRANS) == 0)
      ok = false;
    }

    if ((sylimit & 0x200) && ((wtab+1)->flags & FLAG_LAST_WORD))
{
     // not if the next word is end-of-sentence
     ok = false;
    }

    if (ok == false)
```

```
     strcpy(word_phonemes, ph_buf);
   }

   if (ok) {
    *p2 = '-'; // replace next space by hyphen
    wtab[0].flags &= ~FLAG_ALL_UPPER; // prevent it being
considered an abbreviation
    flags = TranslateWord(translator, word, wtab, NULL); //
translate the combined word
    if ((sylimit > 0) && (CountSyllables(p) > (sylimit & 0x1f)))
{
     // revert to separate words
     *p2 = ' ';
     flags = TranslateWord(translator, word, wtab, NULL);
    } else {
     if (flags == 0)
      flags = flags2[0]; // no flags for the combined word, so
use flags from the second word eg. lang-hu "nem december 7-e"
     flags |= FLAG_SKIPWORDS;
     dictionary_skipwords = 1;
    }
   }
  }

  if (p[0] == phonSWITCH) {
   int switch_attempt;
   strcpy(old_dictionary_name, dictionary_name);
   for (switch_attempt = 0; switch_attempt < 2; switch_attempt++)
{
    // this word uses a different language
    memcpy(word, word_copy, word_copy_len);

    new_language = (char *)(&p[1]);
    if (new_language[0] == 0)
     new_language = "en";

    switch_phonemes = SetTranslator2(new_language);
```

```
    if (switch_phonemes >= 0) {
     // re-translate the word using the new translator
     wtab[0].flags |= FLAG_TRANSLATOR2;
     if (word_replaced[2] != 0) {
      word_replaced[0] = 0; // byte before the start of the word
      word_replaced[1] = ' ';
      flags = TranslateWord(translator2, &word_replaced[1], wtab,
NULL);
     } else
      flags = TranslateWord(translator2, word, wtab,
&word_replaced[2]);
    }

    if (p[0] != phonSWITCH)
     break;
   }

   if (p[0] == phonSWITCH)
    return FLAG_SPELLWORD;

   if (switch_phonemes < 0) {
    // language code is not recognised or 2nd translator won't
translate it
    p[0] = phonSCHWA; // just say something
    p[1] = phonSCHWA;
    p[2] = 0;
   }

   if (switch_phonemes == -1) {
    strcpy(dictionary_name, old_dictionary_name);
    SelectPhonemeTable(voice->phoneme_tab_ix);

    // leave switch_phonemes set, but use the original phoneme
table number.
    // This will suppress LOPT_REGRESSIVE_VOICING
    switch_phonemes = voice->phoneme_tab_ix; // original phoneme
```

```
table
   }
  }

  if (!(word_flags & FLAG_HYPHEN)) {
   if (flags & FLAG_PAUSE1) {
    if (pre_pause < 1)
     pre_pause = 1;
   }
   if ((flags & FLAG_PREPAUSE) && !(word_flags & (FLAG_LAST_WORD
| FLAG_FIRST_WORD)) && !(wtab[-1].flags & FLAG_FIRST_WORD) &&
(tr->prepause_timeout == 0)) {
    // the word is marked in the dictionary list with $pause
    if (pre_pause < 4) pre_pause = 4;
    tr->prepause_timeout = 3;
   }
  }

  if ((option_emphasis >= 3) && (pre_pause < 1))
   pre_pause = 1;
 }

 stress = 0;
 next_stress = 1;
 srcix = 0;
 max_stress = -1;

 found_dict_flag = 0;
 if ((flags & FLAG_FOUND) && !(flags & FLAG_TEXTMODE))
  found_dict_flag = SFLAG_DICTIONARY;

 while ((pre_pause > 0) && (n_ph_list2 < N_PHONEME_LIST-4)) {
  // add pause phonemes here. Either because of punctuation
(brackets or quotes) in the
  // text, or because the word is marked in the dictionary lookup
as a conjunction
  if (pre_pause > 1) {
```

```
   SetPlist2(&ph_list2[n_ph_list2++], phonPAUSE);
   pre_pause -= 2;
  } else {
   SetPlist2(&ph_list2[n_ph_list2++], phonPAUSE_NOLINK);
   pre_pause--;
  }
  tr->end_stressed_vowel = 0; // forget about the previous word
  tr->prev_dict_flags[0] = 0;
  tr->prev_dict_flags[1] = 0;
 }
 plist2 = &ph_list2[n_ph_list2];

 if ((option_capitals == 1) && (word_flags & FLAG_FIRST_UPPER)) {
  SetPlist2(&ph_list2[n_ph_list2++], phonPAUSE_SHORT);
  SetPlist2(&ph_list2[n_ph_list2++], phonCAPITAL);
  if ((word_flags & FLAG_ALL_UPPER) && IsAlpha(word[1])) {
   // word > 1 letter and all capitals
   SetPlist2(&ph_list2[n_ph_list2++], phonPAUSE_SHORT);
   SetPlist2(&ph_list2[n_ph_list2++], phonCAPITAL);
  }
 }

 if (switch_phonemes >= 0) {
  if ((p[0] == phonPAUSE) && (p[1] == phonSWITCH)) {
   // the new word starts with a phoneme table switch, so there's
no need to switch before it.
   if (ph_list2[n_ph_list2-1].phcode == phonSWITCH) {
    // previous phoneme is also a phonSWITCH, delete it
    n_ph_list2--;
   }
  } else {
   // this word uses a different phoneme table
   if (ph_list2[n_ph_list2-1].phcode == phonSWITCH) {
    // previous phoneme is also a phonSWITCH, just change its
phoneme table number
    n_ph_list2--;
   } else
```

```
    SetPlist2(&ph_list2[n_ph_list2], phonSWITCH);
   ph_list2[n_ph_list2++].tone_ph = switch_phonemes; // temporary
phoneme table number
  }
 }

 // remove initial pause from a word if it follows a hyphen
 if ((word_flags & FLAG_HYPHEN) && (phoneme_tab[*p]->type ==
phPAUSE))
  p++;

 if ((p[0] == 0) && (embedded_flag)) {
  // no phonemes.  Insert a very short pause to carry an embedded
command
  p[0] = phonPAUSE_VSHORT;
  p[1] = 0;
 }

 while (((ph_code = *p++) != 0) && (n_ph_list2 <
N_PHONEME_LIST-4)) {
  if (ph_code == 255)
   continue; // unknown phoneme

  // Add the phonemes to the first stage phoneme list (ph_list2)
  ph = phoneme_tab[ph_code];
  if (ph == NULL) {
   printf("Invalid phoneme code %d\n", ph_code);
   continue;
  }

  if (ph_code == phonSWITCH) {
   ph_list2[n_ph_list2].phcode = ph_code;
   ph_list2[n_ph_list2].sourceix = 0;
   ph_list2[n_ph_list2].synthflags = 0;
   ph_list2[n_ph_list2++].tone_ph = *p;
   SelectPhonemeTable(*p);
   p++;
```

```
  } else if (ph->type == phSTRESS) {
   // don't add stress phonemes codes to the list, but give their
stress
   // value to the next vowel phoneme
   // std_length is used to hold stress number or (if >10) a tone
number for a tone language
   if (ph->program == 0)
    next_stress = ph->std_length;
   else {
    // for tone languages, the tone number for a syllable follows
the vowel
    if (prev_vowel >= 0)
     ph_list2[prev_vowel].tone_ph = ph_code;
    else
     next_tone = ph_code; // no previous vowel, apply to the next
vowel
   }
  } else if (ph_code == phonSYLLABIC) {
   // mark the previous phoneme as a syllabic consonant
   prev_vowel = n_ph_list2-1;
   ph_list2[prev_vowel].synthflags |= SFLAG_SYLLABLE;
   ph_list2[prev_vowel].stresslevel = next_stress;
  } else if (ph_code == phonLENGTHEN)
   ph_list2[n_ph_list2-1].synthflags |= SFLAG_LENGTHEN;
  else if (ph_code == phonEND_WORD) {
   // a || symbol in a phoneme string was used to indicate a word
boundary
   // Don't add this phoneme to the list, but make sure the next
phoneme has
   // a newword indication
   srcix = source_ix+1;
  } else if (ph_code == phonX1) {
   // a language specific action
   if (tr->langopts.param[LOPT_IT_DOUBLING])
    flags |= FLAG_DOUBLING;
  } else {
   ph_list2[n_ph_list2].phcode = ph_code;
```

```
   ph_list2[n_ph_list2].tone_ph = 0;
   ph_list2[n_ph_list2].synthflags = embedded_flag |
found_dict_flag;
   embedded_flag = 0;
   ph_list2[n_ph_list2].sourceix = srcix;
   srcix = 0;

   if (ph->type == phVOWEL) {
    stress = next_stress;
    next_stress = 1; // default is 'unstressed'

    if (stress >= 4)
     any_stressed_words = true;

    if ((prev_vowel >= 0) && (n_ph_list2-1) != prev_vowel)
     ph_list2[n_ph_list2-1].stresslevel = stress; // set stress
for previous consonant

    ph_list2[n_ph_list2].synthflags |= SFLAG_SYLLABLE;
    prev_vowel = n_ph_list2;

    if (stress > max_stress) {
     max_stress = stress;
     max_stress_ix = n_ph_list2;
    }
    if (next_tone != 0) {
     ph_list2[n_ph_list2].tone_ph = next_tone;
     next_tone = 0;
    }
   } else {
    if (first_phoneme && tr->langopts.param[LOPT_IT_DOUBLING]) {
     if (((tr->prev_dict_flags[0] & FLAG_DOUBLING) &&
(tr->langopts.param[LOPT_IT_DOUBLING] & 1)) ||
         (tr->end_stressed_vowel &&
(tr->langopts.param[LOPT_IT_DOUBLING] & 2))) {
      // italian, double the initial consonant if the previous
word ends with a
```

```
      // stressed vowel, or is marked with a flag
      ph_list2[n_ph_list2].synthflags |= SFLAG_LENGTHEN;
     }
    }
   }

   ph_list2[n_ph_list2].stresslevel = stress;
   n_ph_list2++;
   first_phoneme = false;
  }
 }

 if (word_flags & FLAG_COMMA_AFTER)
  SetPlist2(&ph_list2[n_ph_list2++], phonPAUSE_CLAUSE);

 // don't set new-word if there is a hyphen before it
 if ((word_flags & FLAG_HYPHEN) == 0)
  plist2->sourceix = source_ix;

 tr->end_stressed_vowel = 0;
 if ((stress >= 4) &&
(phoneme_tab[ph_list2[n_ph_list2-1].phcode]->type == phVOWEL))
  tr->end_stressed_vowel = 1; // word ends with a stressed vowel

 if (switch_phonemes >= 0) {
  // this word uses a different phoneme table, now switch back
  strcpy(dictionary_name, old_dictionary_name);
  SelectPhonemeTable(voice->phoneme_tab_ix);
  SetPlist2(&ph_list2[n_ph_list2], phonSWITCH);
  ph_list2[n_ph_list2++].tone_ph = voice->phoneme_tab_ix; //
original phoneme table number
 }

 if (pitch_raised > 0) {
  embedded_list[embedded_ix++] = EMBED_P+0x60+0x80 +
(pitch_raised << 8); // lower pitch
  SetPlist2(&ph_list2[n_ph_list2], phonPAUSE_SHORT);
```

```
  ph_list2[n_ph_list2++].synthflags = SFLAG_EMBEDDED;
 }

 if (flags & FLAG_STRESS_END2) {
  // this's word's stress could be increased later
  ph_list2[max_stress_ix].synthflags |= SFLAG_PROMOTE_STRESS;
 }

 tr->prev_dict_flags[0] = flags;
 return flags;
}

static int EmbeddedCommand(unsigned int *source_index_out)
{
 // An embedded command to change the pitch, volume, etc.
 // returns number of commands added to embedded_list

 // pitch,speed,amplitude,expression,reverb,tone,voice,sayas
 const char *commands = "PSARHTIVYMUBF";
 int value = -1;
 int sign = 0;
 unsigned char c;
 char *p;
 int cmd;
 int source_index = *source_index_out;

 c = source[source_index];
 if (c == '+') {
  sign = 0x40;
  source_index++;
 } else if (c == '-') {
  sign = 0x60;
  source_index++;
 }

 if (IsDigit09(source[source_index])) {
  value = atoi(&source[source_index]);
```

```
  while (IsDigit09(source[source_index]))
   source_index++;
 }

 c = source[source_index++];
 if (embedded_ix >= (N_EMBEDDED_LIST - 2))
  return 0; // list is full

 if ((p = strchr_w(commands, c)) == NULL)
  return 0;
 cmd = (p - commands)+1;
 if (value == -1) {
  value = embedded_default[cmd];
  sign = 0;
 }

 if (cmd == EMBED_Y) {
  option_sayas2 = value;
  count_sayas_digits = 0;
 }
 if (cmd == EMBED_F) {
  if (value >= 3)
   word_emphasis = FLAG_EMPHASIZED;
  else
   word_emphasis = 0;
 }

 embedded_list[embedded_ix++] = cmd + sign + (value << 8);

 return 1;
}

static const char *FindReplacementChars(Translator *tr, const
char **pfrom, unsigned int c, const char *next, int
*ignore_next_n)
{
 const char *from = *pfrom;
```

```
while ( !is_str_totally_null(from, 4) ) {
 unsigned int fc = 0; // from character
 unsigned int nc = c; // next character
 const char *match_next = next;

 *pfrom = from;

 from += utf8_in((int *)&fc, from);
 if (nc == fc) {
  if (*from == 0) return from + 1;

  bool matched = true;
  int nmatched = 0;
  while (*from != 0) {
   from += utf8_in((int *)&fc, from);

   match_next += utf8_in((int *)&nc, match_next);
   nc = towlower2(nc, tr);

   if (nc != fc)
    matched = false;
   else
    nmatched++;
  }

  if (*from == 0 && matched) {
   *ignore_next_n = nmatched;
   return from + 1;
  }
 }

 // replacement 'from' string (skip the remaining part, if any)
 while (*from != '\0') from++;
 from++;

 // replacement 'to' string
 while (*from != '\0') from++;
```

```
  from++;
 }
 return NULL;
}

// handle .replace rule in xx_rules file
static int SubstituteChar(Translator *tr, unsigned int c,
unsigned int next_in, const char *next, int *insert, int
*wordflags)
{
 unsigned int new_c, c2 = ' ', c_lower;
 int upper_case = 0;

 static int ignore_next_n = 0;
 if (ignore_next_n > 0) {
  ignore_next_n--;
  return 8;
 }

 if (c == 0) return 0;

 const char *from = (const char *)tr->langopts.replace_chars;
 if (from == NULL)
  return c;

 // there is a list of character codes to be substituted with
alternative codes

 if (iswupper(c_lower = c)) {
  c_lower = towlower2(c, tr);
  upper_case = 1;
 }

 const char *to = FindReplacementChars(tr, &from, c_lower, next,
&ignore_next_n);
 if (to == NULL)
  return c; // no substitution
```

```
 if (option_phonemes & espeakPHONEMES_TRACE)
  fprintf(f_trans, "Replace: %s > %s\n", from, to);

 to += utf8_in((int *)&new_c, to);
 if (*to != 0) {
  // there is a second character to be inserted
  // don't convert the case of the second character unless the
next letter is also upper case
  to += utf8_in((int *)&c2, to);
  if (upper_case && iswupper(next_in))
   c2 = ucd_toupper(c2);
  *insert = c2;
 }

 if (upper_case)
  new_c = ucd_toupper(new_c);

 return new_c;
}

static int TranslateChar(Translator *tr, char *ptr, int prev_in,
unsigned int c, unsigned int next_in, int *insert, int
*wordflags)
{
 // To allow language specific examination and replacement of
characters

 int code;
 int initial;
 int medial;
 int final;
 int next2;

 static const unsigned char hangul_compatibility[0x34] = {
  0,  0x00, 0x01, 0xaa, 0x02, 0xac, 0xad, 0x03,
  0x04, 0x05, 0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb4,
```

```
  0xb6, 0x06, 0x07, 0x08, 0xb9, 0x09, 0x0a, 0xbc,
  0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x61,
  0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69,
  0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71,
  0x72, 0x73, 0x74, 0x75
 };

 // check for Korean Hangul letters
 if (((code = c - 0xac00) >= 0) && (c <= 0xd7af)) {
  // break a syllable hangul into 2 or 3 individual jamo
  initial = (code/28)/21;
  medial = (code/28) % 21;
  final = code % 28;

  if (initial == 11) {
   // null initial
   c = medial + 0x1161;
   if (final > 0)
    *insert = final + 0x11a7;
  } else {
   // extact the initial and insert the remainder with a null
initial
   c = initial + 0x1100;
   *insert = (11*28*21) + (medial*28) + final + 0xac00;
  }
  return c;
 } else if (((code = c - 0x3130) >= 0) && (code < 0x34)) {
  // Hangul compatibility jamo
  return hangul_compatibility[code] + 0x1100;
 }

 switch (tr->translator_name)
 {
 case L('a', 'f'):
 case L('n', 'l'):
  // look for 'n  and replace by a special character (unicode:
schwa)
```

```
  if (!iswalpha(prev_in)) {
   utf8_in(&next2, &ptr[1]);

   if ((c == '\'') && IsSpace(next2)) {
    if ((next_in == 'n') && (tr->translator_name == L('a', 'f')))
{
     // n preceded by either apostrophe or U2019 "right single
quotation mark"
     ptr[0] = ' '; // delete the n
     return 0x0259; // replace  '  by  unicode schwa character
    }
    if ((next_in == 'n') || (next_in == 't')) {
     // Dutch, [@n] and [@t]
     return 0x0259; // replace  '  by  unicode schwa character
    }
   }
  }
  break;
 }
 // handle .replace rule in xx_rules file
 return SubstituteChar(tr, c, next_in, ptr, insert, wordflags);
}

static const char *UCase_ga[] = { "bp", "bhf", "dt", "gc", "hA",
"mb", "nd", "ng", "ts", "tA", "nA", NULL };

static int UpperCaseInWord(Translator *tr, char *word, int c)
{
 int ix;
 int len;
 const char *p;

 if (tr->translator_name == L('g', 'a')) {
  // Irish
  for (ix = 0;; ix++) {
   if ((p = UCase_ga[ix]) == NULL)
```

```
    break;

   len = strlen(p);
   if ((word[-len] == ' ') && (memcmp(&word[-len+1], p, len-1) ==
0)) {
    if ((c == p[len-1]) || ((p[len-1] == 'A') && IsVowel(tr, c)))
     return 1;
   }
  }
 }
 return 0;
}

void TranslateClause(Translator *tr, int *tone_out, char
**voice_change)
{
 int ix;
 int c;
 int cc = 0;
 unsigned int source_index = 0;
 unsigned int prev_source_index = 0;
 int source_index_word = 0;
 int prev_in;
 int prev_out = ' ';
 int prev_out2;
 int prev_in_save = 0;
 int next_in;
 int next_in_nbytes;
 int char_inserted = 0;
 int clause_pause;
 int pre_pause_add = 0;
 int all_upper_case = FLAG_ALL_UPPER;
 int alpha_count = 0;
 bool finished = false;
 bool single_quoted = false;
 bool phoneme_mode = false;
 int dict_flags = 0; // returned from dictionary lookup
```

```
int word_flags; // set here
int next_word_flags;
bool new_sentence2;
int embedded_count = 0;
int letter_count = 0;
bool space_inserted = false;
bool syllable_marked = false;
bool decimal_sep_count = false;
char *word;
char *p;
int j, k;
int n_digits;
int charix_top = 0;

short charix[N_TR_SOURCE+4];
WORD_TAB words[N_CLAUSE_WORDS];
static char voice_change_name[40];
int word_count = 0; // index into words

char sbuf[N_TR_SOURCE];

int terminator;
int tone;

if (tr == NULL)
 return;

embedded_ix = 0;
embedded_read = 0;
pre_pause = 0;
any_stressed_words = false;

if ((clause_start_char = count_characters) < 0)
 clause_start_char = 0;
clause_start_word = count_words + 1;

for (ix = 0; ix < N_TR_SOURCE; ix++)
```

```
  charix[ix] = 0;
 terminator = ReadClause(tr, source, charix, &charix_top,
N_TR_SOURCE, &tone, voice_change_name);

 if (tone_out != NULL) {
  if (tone == 0)
   *tone_out = (terminator & CLAUSE_INTONATION_TYPE) >> 12; //
tone type not overridden in ReadClause, use default
  else
   *tone_out = tone; // override tone type
 }

 charix[charix_top+1] = 0;
 charix[charix_top+2] = 0x7fff;
 charix[charix_top+3] = 0;

 clause_pause = (terminator & CLAUSE_PAUSE) * 10; // mS
 if (terminator & CLAUSE_PAUSE_LONG)
  clause_pause = clause_pause * 32; // pause value is *320mS not
*10mS

 for (p = source; *p != 0; p++) {
  if (!isspace2(*p))
   break;
 }
 if (*p == 0) {
  // No characters except spaces. This is not a sentence.
  // Don't add this pause, just make up the previous pause to
this value;
  clause_pause -= max_clause_pause;
  if (clause_pause < 0)
   clause_pause = 0;

  if (new_sentence)
   terminator |= CLAUSE_TYPE_SENTENCE; // carry forward an end-
of-sentence indicator
  max_clause_pause += clause_pause;
```

```
 new_sentence2 = false;
} else {
 max_clause_pause = clause_pause;
 new_sentence2 = new_sentence;
}
tr->clause_terminator = terminator;

if (new_sentence2) {
 count_sentences++;
 if (skip_sentences > 0) {
  skip_sentences--;
  if (skip_sentences == 0)
   skipping_text = false;
 }
}

memset(&ph_list2[0], 0, sizeof(ph_list2[0]));
ph_list2[0].phcode = phonPAUSE_SHORT;

n_ph_list2 = 1;
tr->prev_last_stress = 0;
tr->prepause_timeout = 0;
tr->expect_verb = 0;
tr->expect_noun = 0;
tr->expect_past = 0;
tr->expect_verb_s = 0;
tr->phonemes_repeat_count = 0;
tr->end_stressed_vowel = 0;
tr->prev_dict_flags[0] = 0;
tr->prev_dict_flags[1] = 0;

word_count = 0;
word_flags = 0;
next_word_flags = 0;

sbuf[0] = 0;
sbuf[1] = ' ';
```

```
 sbuf[2] = ' ';
 ix = 3;
 prev_in = ' ';

 words[0].start = ix;
 words[0].flags = 0;

 for (j = 0; charix[j] <= 0; j++) ;
 words[0].sourceix = charix[j];
 k = 0;
 while (charix[j] != 0) {
  // count the number of characters (excluding multibyte
continuation bytes)
  if (charix[j++] != -1)
   k++;
 }
 words[0].length = k;

 while (!finished && (ix < (int)sizeof(sbuf) - 1) && (n_ph_list2
< N_PHONEME_LIST-4)) {
  prev_out2 = prev_out;
  utf8_in2(&prev_out, &sbuf[ix-1], 1);

  if (tr->langopts.tone_numbers && IsDigit09(prev_out) &&
IsAlpha(prev_out2)) {
   // tone numbers can be part of a word, consider them as
alphabetic
   prev_out = 'a';
  }

  if (prev_in_save != 0) {
   prev_in = prev_in_save;
   prev_in_save = 0;
  } else if (source_index > 0)
   utf8_in2(&prev_in, &source[source_index-1], 1);

  prev_source_index = source_index;
```

```
  if (char_inserted) {
   c = char_inserted;
   char_inserted = 0;
  } else {
   source_index += utf8_in(&cc, &source[source_index]);
   c = cc;
  }
  next_in_nbytes = utf8_in(&next_in, &source[source_index]);

  if (c == 0) {
   finished = true;
   c = ' ';
  }

  if ((c == CTRL_EMBEDDED) || (c == ctrl_embedded)) {
   // start of embedded command in the text
   int srcix = source_index-1;

   if (prev_in != ' ') {
    c = ' ';
    prev_in_save = c;
    source_index--;
   } else {
    embedded_count += EmbeddedCommand(&source_index);
    prev_in_save = prev_in;
    // replace the embedded command by spaces
    memset(&source[srcix], ' ', source_index-srcix);
    source_index = srcix;
    continue;
   }
  }

  if ((option_sayas2 == SAYAS_KEY) && (c != ' ')) {
   if ((prev_in == ' ') && (next_in == ' '))
    option_sayas2 = SAYAS_SINGLE_CHARS; // single character,
speak its name
```

```
   c = towlower2(c, tr);
  }

  if (phoneme_mode) {
   all_upper_case = FLAG_PHONEMES;

   if ((c == ']') && (next_in == ']')) {
    phoneme_mode = false;
    source_index++;
    c = ' ';
   }
  } else if ((option_sayas2 & 0xf0) == SAYAS_DIGITS) {
   if (iswdigit(c)) {
    count_sayas_digits++;
    if (count_sayas_digits > (option_sayas2 & 0xf)) {
     // break after the specified number of digits
     c = ' ';
     space_inserted = true;
     count_sayas_digits = 0;
    }
   } else {
    count_sayas_digits = 0;
    if (iswdigit(prev_out)) {
     c = ' ';
     space_inserted = true;
    }
   }
  } else if ((option_sayas2 & 0x10) == 0) {
   // speak as words

   if ((c == 0x92) || (c == 0xb4) || (c == 0x2019) || (c ==
0x2032))
    c = '\''; // 'microsoft' quote or sexed closing single quote,
or prime - possibly used as apostrophe

   if (((c == 0x2018) || (c == '?')) && IsAlpha(prev_out) &&
IsAlpha(next_in)) {
```

```
    // ? between two letters may be a smart-quote replaced by ?
    c = '\'';
   }

   if (c == CHAR_EMPHASIS) {
    // this character is a marker that the previous word is the
focus of the clause
    c = ' ';
    word_flags |= FLAG_FOCUS;
   }

   if (c == CHAR_COMMA_BREAK) {
    c = ' ';
    word_flags |= FLAG_COMMA_AFTER;
   }
   // language specific character translations
   c = TranslateChar(tr, &source[source_index], prev_in, c,
next_in, &char_inserted, &word_flags);
   if (c == 8)
    continue; // ignore this character

   if (char_inserted)
    next_in = char_inserted;

   // allow certain punctuation within a word (usually only
apostrophe)
   if (!IsAlpha(c) && !IsSpace(c) &&
(wcschr(tr->punct_within_word, c) == 0)) {
    if (IsAlpha(prev_out)) {
     if (tr->langopts.tone_numbers && IsDigit09(c) &&
!IsDigit09(next_in)) {
      // allow a tone number as part of the word
     } else {
      c = ' '; // ensure we have an end-of-word terminator
      space_inserted = true;
     }
    }
```

```
   }

   if (iswdigit(prev_out)) {
    if (!iswdigit(c) && (c != '.') && (c != ',') && (c != ' ')) {
     c = ' '; // terminate digit string with a space
     space_inserted = true;
    }
   } else { // Prev output is not digit
    if (prev_in == ',') {
     // Workaround for several consecutive commas -
     // replace current character with space
     if (c == ',')
      c = ' ';
    } else {
     decimal_sep_count = false;
    }
   }

   if (c == '[') {
    if ((next_in == '\002') || ((next_in == '[') &&
option_phoneme_input)) {
     //  "[\002" is used internally to start phoneme mode
     phoneme_mode = true;
     source_index++;
     continue;
    }
   }

   if (IsAlpha(c)) {
    alpha_count++;
    if (!IsAlpha(prev_out) || (tr->langopts.ideographs && ((c >
0x3040) || (prev_out > 0x3040)))) {
     if (wcschr(tr->punct_within_word, prev_out) == 0)
      letter_count = 0; // don't reset count for an apostrophy
within a word

     if ((prev_out != ' ') && (wcschr(tr->punct_within_word,
```

```
prev_out) == 0)) {
      // start of word, insert space if not one there already
      c = ' ';
      space_inserted = true;

      if (!IsBracket(prev_out)) // ?? perhaps only set
FLAG_NOSPACE for . - /  (hyphenated words, URLs, etc)
       next_word_flags |= FLAG_NOSPACE;
     } else {
      if (iswupper(c))
       word_flags |= FLAG_FIRST_UPPER;

      if ((prev_out == ' ') && iswdigit(sbuf[ix-2]) &&
!iswdigit(prev_in)) {
       // word, following a number, but with a space between
       // Add an extra space, to distinguish "2 a" from "2a"
       sbuf[ix++] = ' ';
       words[word_count].start++;
      }
     }
    }

    if (c != ' ') {
     letter_count++;

     if (tr->letter_bits_offset > 0) {
      if (((c < 0x250) && (prev_out >= tr->letter_bits_offset))
||
          ((c >= tr->letter_bits_offset) && (letter_count > 1) &&
(prev_out < 0x250))) {
       // Don't mix native and Latin characters in the same word
       // Break into separate words
       if (IsAlpha(prev_out)) {
        c = ' ';
        space_inserted = true;
        word_flags |= FLAG_HYPHEN_AFTER;
        next_word_flags |= FLAG_HYPHEN;
```

```
      }
     }
    }
   }

   if (iswupper(c)) {
    c = towlower2(c, tr);

    if ((j = tr->langopts.param[LOPT_CAPS_IN_WORD]) > 0) {
     if ((j == 2) && (syllable_marked == false)) {
      char_inserted = c;
      c = 0x2c8; // stress marker
      syllable_marked = true;
     }
    } else {
     if (iswlower(prev_in)) {
      // lower case followed by upper case, possibly CamelCase
      if (UpperCaseInWord(tr, &sbuf[ix], c) == 0) { // start a
new word
       c = ' ';
       space_inserted = true;
       prev_in_save = c;
      }
     } else if ((c != ' ') && iswupper(prev_in) &&
iswlower(next_in)) {
      int next2_in;
      utf8_in(&next2_in, &source[source_index +
next_in_nbytes]);

      if ((tr->translator_name == L('n', 'l')) && (letter_count
== 2) && (c == 'j') && (prev_in == 'I')) {
       // Dutch words may capitalise initial IJ, don't split
      } else if (IsAlpha(next2_in)) {
       // changing from upper to lower case, start new word at
the last uppercase, if 3 or more letters
       c = ' ';
       space_inserted = true;
```

```
        prev_in_save = c;
        next_word_flags |= FLAG_NOSPACE;
       }
      }
     }
    } else {
     if ((all_upper_case) && (letter_count > 2)) {
      if ((c == 's') && (next_in == ' ')) {
       c = ' ';
       all_upper_case |= FLAG_HAS_PLURAL;

       if (sbuf[ix-1] == '\'')
        sbuf[ix-1] = ' ';
      } else
       all_upper_case = 0; // current word contains lower case
letters, not "'s"
     } else
      all_upper_case = 0;
    }
   } else if (c == '-') {
    if (!IsSpace(prev_in) && IsAlpha(next_in)) {
     if (prev_out != ' ') {
      // previous 'word' not yet ended (not alpha or numeric),
start new word now.
      c = ' ';
      space_inserted = true;
     } else {
      // '-' between two letters is a hyphen, treat as a space
      word_flags |= FLAG_HYPHEN;
      if (word_count > 0)
       words[word_count-1].flags |= FLAG_HYPHEN_AFTER;
      c = ' ';
     }
    } else if ((prev_in == ' ') && (next_in == ' ')) {
     // ' - ' dash between two spaces, treat as pause
     c = ' ';
     pre_pause_add = 4;
```

```
    } else if (next_in == '-') {
     // double hyphen, treat as pause
     source_index++;
     c = ' ';
     pre_pause_add = 4;
    } else if ((prev_out == ' ') && IsAlpha(prev_out2) &&
!IsAlpha(prev_in)) {
     // insert extra space between a word + space + hyphen, to
distinguish 'a -2' from 'a-2'
     sbuf[ix++] = ' ';
     words[word_count].start++;
    }
   } else if (c == '.') {
    if (prev_out == '.') {
     // multiple dots, separate by spaces. Note >3 dots has been
replaced by elipsis
     c = ' ';
     space_inserted = true;
    } else if ((word_count > 0) && !(words[word_count-1].flags &
FLAG_NOSPACE) && IsAlpha(prev_in)) {
     // dot after a word, with space following, probably an
abbreviation
     words[word_count-1].flags |= FLAG_HAS_DOT;

     if (IsSpace(next_in) || (next_in == '-'))
      c = ' '; // remove the dot if it's followed by a space or
hyphen, so that it's not pronounced
    }
   } else if (c == '\'') {
    if (((prev_in == '.') || iswalnum(prev_in)) &&
IsAlpha(next_in)) {
     // between two letters, or in an abbreviation (eg.
u.s.a.'s). Consider the apostrophe as part of the word
     single_quoted = false;
    } else if ((tr->langopts.param[LOPT_APOSTROPHE] & 1) &&
IsAlpha(next_in))
     single_quoted = false; // apostrophe at start of word is
```

```
part of the word
    else if ((tr->langopts.param[LOPT_APOSTROPHE] & 2) &&
IsAlpha(prev_in))
     single_quoted = false; // apostrophe at end of word is part
of the word
    else if ((wcschr(tr->char_plus_apostrophe, prev_in) != 0) &&
(prev_out2 == ' ')) {
     // consider single character plus apostrophe as a word
     single_quoted = false;
     if (next_in == ' ')
      source_index++; // skip following space
    } else {
     if ((prev_out == 's') && (single_quoted == false)) {
      // looks like apostrophe after an 's'
      c = ' ';
     } else {
      if (IsSpace(prev_out))
       single_quoted = true;
      else
       single_quoted = false;

      pre_pause_add = 4; // single quote
      c = ' ';
     }
    }
   } else if (lookupwchar(breaks, c) != 0)
    c = ' '; // various characters to treat as space
   else if (iswdigit(c)) {
    if (tr->langopts.tone_numbers && IsAlpha(prev_out) &&
!IsDigit(next_in)) {
    } else if ((prev_out != ' ') && !iswdigit(prev_out)) {
     if ((prev_out != tr->langopts.decimal_sep) ||
((decimal_sep_count == true) && (tr->langopts.decimal_sep ==
','))) {
      c = ' ';
      space_inserted = true;
     } else
```

```
      decimal_sep_count = true;
    } else if ((prev_out == ' ') && IsAlpha(prev_out2) &&
!IsAlpha(prev_in)) {
     // insert extra space between a word and a number, to
distinguish 'a 2' from 'a2'
     sbuf[ix++] = ' ';
     words[word_count].start++;
    }
   }
  }

  if (IsSpace(c)) {
   if (prev_out == ' ') {
    word_flags |= FLAG_MULTIPLE_SPACES;
    continue; // multiple spaces
   }

   if ((cc == 0x09) || (cc == 0x0a))
    next_word_flags |= FLAG_MULTIPLE_SPACES; // tab or newline,
not a simple space

   if (space_inserted) {
    // count the number of characters since the start of the word
    j = 0;
    k = source_index - 1;
    while ((k >= source_index_word) && (charix[k] != 0)) {
     if (charix[k] > 0) // don't count initial bytes of multi-
byte character
      j++;
     k--;
    }
    words[word_count].length = j;
   }

   source_index_word = source_index;

   // end of 'word'
```

```
   sbuf[ix++] = ' ';

   if ((word_count < N_CLAUSE_WORDS-1) && (ix >
words[word_count].start)) {
    if (embedded_count > 0) {
     // there are embedded commands before this word
     embedded_list[embedded_ix-1] |= 0x80; // terminate list of
commands for this word
     words[word_count].flags |= FLAG_EMBEDDED;
     embedded_count = 0;
    }
    if (alpha_count == 0) {
     all_upper_case &= ~FLAG_ALL_UPPER;
    }
    words[word_count].pre_pause = pre_pause;
    words[word_count].flags |= (all_upper_case | word_flags |
word_emphasis);

    if (pre_pause > 0) {
     // insert an extra space before the word, to prevent
influence from previous word across the pause
     for (j = ix; j > words[word_count].start; j--)
      sbuf[j] = sbuf[j-1];
     sbuf[j] = ' ';
     words[word_count].start++;
     ix++;
    }

    word_count++;
    words[word_count].start = ix;
    words[word_count].flags = 0;

    for (j = source_index; j < charix_top && charix[j] <= 0; j++)
// skip blanks
     ;
    words[word_count].sourceix = charix[j];
    k = 0;
```

```
    while (charix[j] != 0) {
     // count the number of characters (excluding multibyte
continuation bytes)
     if (charix[j++] != -1)
      k++;
    }
    words[word_count].length = k;

    word_flags = next_word_flags;
    next_word_flags = 0;
    pre_pause = 0;
    all_upper_case = FLAG_ALL_UPPER;
    alpha_count = 0;
    syllable_marked = false;
   }

   if (space_inserted) {
    source_index = prev_source_index; // rewind to the previous
character
    char_inserted = 0;
    space_inserted = false;
   }
  } else {
   if ((ix < (N_TR_SOURCE - 4)))
    ix += utf8_out(c, &sbuf[ix]);
  }
  if (pre_pause_add > pre_pause)
   pre_pause = pre_pause_add;
  pre_pause_add = 0;
 }

 if ((word_count == 0) && (embedded_count > 0)) {
  // add a null 'word' to carry the embedded command flag
  embedded_list[embedded_ix-1] |= 0x80;
  words[word_count].flags |= FLAG_EMBEDDED;
  word_count = 1;
 }
```

```
 tr->clause_end = &sbuf[ix-1];
 sbuf[ix] = 0;
 words[0].pre_pause = 0; // don't add extra pause at beginning of
clause
 words[word_count].pre_pause = 8;
 if (word_count > 0) {
  ix = word_count-1;
  while ((ix > 0) && (IsBracket(sbuf[words[ix].start])))
   ix--; // the last word is a bracket, mark the previous word as
last
  words[ix].flags |= FLAG_LAST_WORD;

  // FLAG_NOSPACE check to avoid recognizing  .mr  -mr
  if ((terminator & CLAUSE_DOT_AFTER_LAST_WORD) &&
!(words[word_count-1].flags & FLAG_NOSPACE))
   words[word_count-1].flags |= FLAG_HAS_DOT;
 }
 words[0].flags |= FLAG_FIRST_WORD;

 for (ix = 0; ix < word_count; ix++) {
  int nx;
  int c_temp;
  char *pn;
  char *pw;
  int nw;
  char number_buf[150];
  WORD_TAB num_wtab[50]; // copy of 'words', when splitting
numbers into parts

  // start speaking at a specified word position in the text?
  count_words++;
  if (skip_words > 0) {
   skip_words--;
   if (skip_words == 0)
    skipping_text = false;
  }
```

```
  if (skipping_text)
   continue;

  current_alphabet = NULL;

  // digits should have been converted to Latin alphabet ('0' to
'9')
  word = pw = &sbuf[words[ix].start];

  if (iswdigit(word[0]) && (tr->langopts.break_numbers !=
BREAK_THOUSANDS)) {
   // Languages with 100000 numbers.  Remove thousands separators
so that we can insert them again later
   pn = number_buf;
   while (pn < &number_buf[sizeof(number_buf)-20]) {
    if (iswdigit(*pw))
     *pn++ = *pw++;
    else if ((*pw == tr->langopts.thousands_sep) && (pw[1] == '
')
               && iswdigit(pw[2]) && (pw[3] != ' ') && (pw[4] !=
' ')) { // don't allow only 1 or 2 digits in the final part
     pw += 2;
     ix++; // skip "word"
    } else {
     nx = pw - word;
     memset(word, ' ', nx);
     nx = pn - number_buf;
     memcpy(word, number_buf, nx);
     break;
    }
   }
   pw = word;
  }

  for (n_digits = 0; iswdigit(word[n_digits]); n_digits++) //
count consecutive digits
   ;
```

```
  if (n_digits > 4) {
   // word is entirely digits, insert commas and break into 3
digit "words"
   number_buf[0] = ' ';
   pn = &number_buf[1];
   nx = n_digits;
   nw = 0;

   if ((n_digits > tr->langopts.max_digits) || (word[0] == '0'))
    words[ix].flags |= FLAG_INDIVIDUAL_DIGITS;

   while (pn < &number_buf[sizeof(number_buf)-20]) {
    if (!IsDigit09(c = *pw++) && (c != tr->langopts.decimal_sep))
     break;

    *pn++ = c;
    nx--;
    if ((nx > 0) && (tr->langopts.break_numbers & (1 << nx))) {
     memcpy(&num_wtab[nw++], &words[ix], sizeof(WORD_TAB)); //
copy the 'words' entry for each word of numbers

     if (tr->langopts.thousands_sep != ' ')
      *pn++ = tr->langopts.thousands_sep;
     *pn++ = ' ';

     if ((words[ix].flags & FLAG_INDIVIDUAL_DIGITS) == 0) {
      if (tr->langopts.break_numbers & (1 << (nx-1))) {
       // the next group only has 1 digits, make it three
       *pn++ = '0';
      }
      if (tr->langopts.break_numbers & (1 << (nx-2))) {
       // the next group only has 2 digits (eg. Indian
languages), make it three
       *pn++ = '0';
      }
     }
```

```
     }
    }
    pw--;
    memcpy(&num_wtab[nw], &words[ix], sizeof(WORD_TAB)*2); // the
original number word, and the word after it

    for (j = 1; j <= nw; j++)
     num_wtab[j].flags &= ~(FLAG_MULTIPLE_SPACES | FLAG_EMBEDDED);
// don't use these flags for subsequent parts when splitting a
number

    // include the next few characters, in case there are an
ordinal indicator or other suffix
    memcpy(pn, pw, 16);
    pn[16] = 0;
    nw = 0;

    for (pw = &number_buf[1]; pw < pn;) {
     // keep wflags for each part, for FLAG_HYPHEN_AFTER
     dict_flags = TranslateWord2(tr, pw, &num_wtab[nw++],
words[ix].pre_pause);
     while (*pw++ != ' ')
      ;
     words[ix].pre_pause = 0;
    }
   } else {
    pre_pause = 0;

    dict_flags = TranslateWord2(tr, word, &words[ix],
words[ix].pre_pause);

    if (pre_pause > words[ix+1].pre_pause) {
     words[ix+1].pre_pause = pre_pause;
     pre_pause = 0;
    }

    if (dict_flags & FLAG_SPELLWORD) {
```

```
    // redo the word, speaking single letters
    for (pw = word; *pw != ' ';) {
     memset(number_buf, ' ', 9);
     nx = utf8_in(&c_temp, pw);
     memcpy(&number_buf[2], pw, nx);
     TranslateWord2(tr, &number_buf[2], &words[ix], 0);
     pw += nx;
    }
   }

   if ((dict_flags & (FLAG_ALLOW_DOT | FLAG_NEEDS_DOT)) && (ix ==
word_count - 1 - dictionary_skipwords) && (terminator &
CLAUSE_DOT_AFTER_LAST_WORD)) {
    // probably an abbreviation such as Mr. or B. rather than end
of sentence
    clause_pause = 10;
    if (tone_out != NULL)
     *tone_out = 4;
   }
  }

  if (dict_flags & FLAG_SKIPWORDS) {
   // dictionary indicates skip next word(s)
   while (dictionary_skipwords > 0) {
    words[ix+dictionary_skipwords].flags |= FLAG_DELETE_WORD;
    dictionary_skipwords--;
   }
  }
 }

 if (embedded_read < embedded_ix) {
  // any embedded commands not yet processed?
  Word_EmbeddedCmd();
 }

 for (ix = 0; ix < 2; ix++) {
  // terminate the clause with 2 PAUSE phonemes
```

```
  PHONEME_LIST2 *p2;
  p2 = &ph_list2[n_ph_list2 + ix];
  p2->phcode = phonPAUSE;
  p2->stresslevel = 0;
  p2->sourceix = source_index;
  p2->synthflags = 0;
 }
 n_ph_list2 += 2;

 if (count_words == 0)
  clause_pause = 0;
 if (Eof() && ((word_count == 0) || (option_endpause == 0)))
  clause_pause = 10;

 MakePhonemeList(tr, clause_pause, new_sentence2, &n_ph_list2,
ph_list2);
 phoneme_list[N_PHONEME_LIST].ph = NULL; // recognize end of
phoneme_list array, in Generate()
 phoneme_list[N_PHONEME_LIST].sourceix = 1;

 if (embedded_count) { // ???? is this needed
  phoneme_list[n_phoneme_list-2].synthflags = SFLAG_EMBEDDED;
  embedded_list[embedded_ix-1] |= 0x80;
  embedded_list[embedded_ix] = 0x80;
 }

 prev_clause_pause = clause_pause;

 new_sentence = false;
 if (terminator & CLAUSE_TYPE_SENTENCE)
  new_sentence = true; // next clause is a new sentence

 if (voice_change != NULL) {
  // return new voice name if an embedded voice change command
terminated the clause
  if (terminator & CLAUSE_TYPE_VOICE_CHANGE)
   *voice_change = voice_change_name;
```

```
  else
   *voice_change = NULL;
 }
}

void InitText(int control)
{
 count_sentences = 0;
 count_words = 0;
 end_character_position = 0;
 skip_sentences = 0;
 skip_marker[0] = 0;
 skip_words = 0;
 skip_characters = 0;
 skipping_text = false;
 new_sentence = true;

 prev_clause_pause = 0;

 option_sayas = 0;
 option_sayas2 = 0;
 option_emphasis = 0;
 word_emphasis = 0;
 embedded_flag = 0;

 InitText2();

 if ((control & espeakKEEP_NAMEDATA) == 0)
  InitNamedata();
}
```

Chapter 39

./src/libespeak-ng/compiledata.c

```
#include "config.h"

#include <ctype.h>
#include <errno.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "readclause.h"
#include "synthdata.h"
#include "wavegen.h"
```

```
#include "error.h"
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "spect.h"
#include "translate.h"
#include "dictionary.h"

#define N_ITEM_STRING 256

typedef struct {
 unsigned int value;
 char *name;
} NAMETAB;

NAMETAB *manifest = NULL;
int n_manifest;
char phsrc[sizeof(path_home)+40]; // Source: path to the
'phonemes' source file.

typedef struct {
 const char *mnem;
 int type;
 int data;
} keywtab_t;

#define k_AND     1
#define k_OR      2
#define k_THEN    3
#define k_NOT     4

#define kTHISSTRESS 0x800

// keyword types
enum {
 tPHONEME_TYPE = 1,
 tPHONEME_FLAG,
```

```
 tTRANSITION,
 tSTATEMENT,
 tINSTRN1,
 tWHICH_PHONEME,
 tTEST,
};

static keywtab_t k_conditions[] = {
 { "AND",  0, k_AND },
 { "OR",   0, k_OR },
 { "THEN", 0, k_THEN },
 { "NOT",  0, k_NOT },

 { "prevPh",    tWHICH_PHONEME,  0 },
 { "thisPh",    tWHICH_PHONEME,  1 },
 { "nextPh",    tWHICH_PHONEME,  2 },
 { "next2Ph",   tWHICH_PHONEME,  3 },
 { "nextPhW",   tWHICH_PHONEME,  4 },
 { "prevPhW",   tWHICH_PHONEME,  5 },
 { "next2PhW",  tWHICH_PHONEME,  6 },
 { "nextVowel", tWHICH_PHONEME,  7 },
 { "prevVowel", tWHICH_PHONEME,  8 },
 { "next3PhW",  tWHICH_PHONEME,  9 },
 { "prev2PhW",  tWHICH_PHONEME, 10 },

 { "PreVoicing",  tTEST, 0xf01 },
 { "KlattSynth",  tTEST, 0xf02 },
 { "MbrolaSynth", tTEST, 0xf03 },

 { NULL, 0, 0 }
};

static keywtab_t k_properties[] = {
 { "isPause",      0, CONDITION_IS_PHONEME_TYPE | phPAUSE },
 { "isVowel",      0, CONDITION_IS_PHONEME_TYPE | phVOWEL },
 { "isNasal",      0, CONDITION_IS_PHONEME_TYPE | phNASAL },
 { "isLiquid",     0, CONDITION_IS_PHONEME_TYPE | phLIQUID },
```

```
 { "isUStop",      0, CONDITION_IS_PHONEME_TYPE | phSTOP },
 { "isVStop",      0, CONDITION_IS_PHONEME_TYPE | phVSTOP },
 { "isVFricative", 0, CONDITION_IS_PHONEME_TYPE | phVFRICATIVE },

 { "isPalatal",  0, CONDITION_IS_PHFLAG_SET | phFLAGBIT_PALATAL
},
 { "isLong",     0, CONDITION_IS_PHFLAG_SET | phFLAGBIT_LONG },
 { "isRhotic",   0, CONDITION_IS_PHFLAG_SET | phFLAGBIT_RHOTIC },
 { "isSibilant", 0, CONDITION_IS_PHFLAG_SET | phFLAGBIT_SIBILANT
},
 { "isFlag1",    0, CONDITION_IS_PHFLAG_SET | phFLAGBIT_FLAG1 },
 { "isFlag2",    0, CONDITION_IS_PHFLAG_SET | phFLAGBIT_FLAG2 },

 { "isVelar", 0, CONDITION_IS_PLACE_OF_ARTICULATION |
phPLACE_VELAR },

 { "isDiminished",  0, CONDITION_IS_OTHER | STRESS_IS_DIMINISHED
},
 { "isUnstressed",  0, CONDITION_IS_OTHER | STRESS_IS_UNSTRESSED
},
 { "isNotStressed", 0, CONDITION_IS_OTHER |
STRESS_IS_NOT_STRESSED },
 { "isStressed",    0, CONDITION_IS_OTHER | STRESS_IS_SECONDARY
},
 { "isMaxStress",   0, CONDITION_IS_OTHER | STRESS_IS_PRIMARY },

 { "isPause2",           0, CONDITION_IS_OTHER | isBreak },
 { "isWordStart",        0, CONDITION_IS_OTHER | isWordStart },
 { "isWordEnd",          0, CONDITION_IS_OTHER | isWordEnd },
 { "isAfterStress",      0, CONDITION_IS_OTHER | isAfterStress },
 { "isNotVowel",         0, CONDITION_IS_OTHER | isNotVowel },
 { "isFinalVowel",       0, CONDITION_IS_OTHER | isFinalVowel },
 { "isVoiced",           0, CONDITION_IS_OTHER | isVoiced },
 { "isFirstVowel",       0, CONDITION_IS_OTHER | isFirstVowel },
 { "isSecondVowel",      0, CONDITION_IS_OTHER | isSecondVowel },
 { "isTranslationGiven", 0, CONDITION_IS_OTHER |
isTranslationGiven },
```

```
 { NULL, 0, 0 }
};

enum {
 kPHONEMESTART = 1,
 kUTF8_BOM,
 kPROCEDURE,
 kENDPHONEME,
 kENDPROCEDURE,
 kPHONEMETABLE,
 kINCLUDE,
 kIMPORT_PH,

 kSTARTTYPE,
 kENDTYPE,
 kSTRESSTYPE,
 kVOICINGSWITCH,

 kIF,
 kELSE,
 kELIF,
 kENDIF,
 kCALLPH,

 kSWITCH_PREVVOWEL,
 kSWITCH_NEXTVOWEL,
 kENDSWITCH,

 kFMT,
 kWAV,
 kVOWELSTART,
 kVOWELENDING,
 kANDWAV,

 kVOWELIN,
 kVOWELOUT,
```

```
 kTONESPEC,

 kRETURN,
 kCONTINUE,
};

enum {
 kTUNE = 1,
 kENDTUNE,
 kTUNE_PREHEAD,
 kTUNE_ONSET,
 kTUNE_HEAD,
 kTUNE_HEADENV,
 kTUNE_HEADEXTEND,
 kTUNE_HEADLAST,
 kTUNE_NUCLEUS0,
 kTUNE_NUCLEUS1,
 kTUNE_SPLIT,
};

static unsigned const char utf8_bom[] = { 0xef, 0xbb, 0xbf, 0 };

static keywtab_t k_intonation[] = {
 { "tune",       0, kTUNE },
 { "endtune",    0, kENDTUNE },
 { "prehead",    0, kTUNE_PREHEAD },
 { "onset",      0, kTUNE_ONSET },
 { "head",       0, kTUNE_HEAD },
 { "headenv",    0, kTUNE_HEADENV },
 { "headextend", 0, kTUNE_HEADEXTEND },
 { "headlast",   0, kTUNE_HEADLAST },
 { "nucleus0",   0, kTUNE_NUCLEUS0 },
 { "nucleus",    0, kTUNE_NUCLEUS1 },
 { "split",      0, kTUNE_SPLIT },

 { NULL, 0, -1 }
};
```

```
static keywtab_t keywords[] = {
 { "liquid",  tPHONEME_TYPE, phLIQUID },
 { "pause",   tPHONEME_TYPE, phPAUSE },
 { "stress",  tPHONEME_TYPE, phSTRESS },
 { "virtual", tPHONEME_TYPE, phVIRTUAL },
 { "delete_phoneme", tPHONEME_TYPE, phDELETED },

 // keywords
 { "phonemetable",         tSTATEMENT, kPHONEMETABLE },
 { "include",              tSTATEMENT, kINCLUDE },
 { (const char *)utf8_bom, tSTATEMENT, kUTF8_BOM },

 { "phoneme",        tSTATEMENT, kPHONEMESTART },
 { "procedure",      tSTATEMENT, kPROCEDURE },
 { "endphoneme",     tSTATEMENT, kENDPHONEME },
 { "endprocedure",   tSTATEMENT, kENDPROCEDURE },
 { "import_phoneme", tSTATEMENT, kIMPORT_PH },
 { "stress_type",    tSTATEMENT, kSTRESSTYPE },
 { "starttype",      tSTATEMENT, kSTARTTYPE },
 { "endtype",        tSTATEMENT, kENDTYPE },
 { "voicingswitch",  tSTATEMENT, kVOICINGSWITCH },

 { "IF",     tSTATEMENT, kIF },
 { "ELSE",   tSTATEMENT, kELSE },
 { "ELIF",   tSTATEMENT, kELIF },
 { "ELSEIF", tSTATEMENT, kELIF },  // same as ELIF
 { "ENDIF",  tSTATEMENT, kENDIF },
 { "CALL",   tSTATEMENT, kCALLPH },
 { "RETURN", tSTATEMENT, kRETURN },

 { "PrevVowelEndings", tSTATEMENT, kSWITCH_PREVVOWEL },
 { "NextVowelStarts",  tSTATEMENT, kSWITCH_NEXTVOWEL },
 { "EndSwitch",        tSTATEMENT, kENDSWITCH },

 { "Tone",        tSTATEMENT, kTONESPEC },
 { "FMT",         tSTATEMENT, kFMT },
```

```
 { "WAV",         tSTATEMENT, kWAV },
 { "VowelStart",  tSTATEMENT, kVOWELSTART },
 { "VowelEnding", tSTATEMENT, kVOWELENDING },
 { "addWav",      tSTATEMENT, kANDWAV },

 { "Vowelin",  tSTATEMENT, kVOWELIN },
 { "Vowelout", tSTATEMENT, kVOWELOUT },
 { "Continue", tSTATEMENT, kCONTINUE },

 { "ChangePhoneme",       tINSTRN1, i_CHANGE_PHONEME },
 { "ChangeNextPhoneme",   tINSTRN1, i_REPLACE_NEXT_PHONEME },
 { "InsertPhoneme",       tINSTRN1, i_INSERT_PHONEME },
 { "AppendPhoneme",       tINSTRN1, i_APPEND_PHONEME },
 { "IfNextVowelAppend",   tINSTRN1, i_APPEND_IFNEXTVOWEL },
 { "ChangeIfDiminished",  tINSTRN1, i_CHANGE_IF |
STRESS_IS_DIMINISHED },
 { "ChangeIfUnstressed",  tINSTRN1, i_CHANGE_IF |
STRESS_IS_UNSTRESSED },
 { "ChangeIfNotStressed", tINSTRN1, i_CHANGE_IF |
STRESS_IS_NOT_STRESSED },
 { "ChangeIfStressed",    tINSTRN1, i_CHANGE_IF |
STRESS_IS_SECONDARY },
 { "ChangeIfStressed",    tINSTRN1, i_CHANGE_IF |
STRESS_IS_PRIMARY },

 { "PauseBefore", tINSTRN1, i_PAUSE_BEFORE },
 { "PauseAfter",  tINSTRN1, i_PAUSE_AFTER },
 { "length",      tINSTRN1, i_SET_LENGTH },
 { "LongLength",  tINSTRN1, i_LONG_LENGTH },
 { "LengthAdd",   tINSTRN1, i_ADD_LENGTH },
 { "lengthmod",   tINSTRN1, i_LENGTH_MOD },
 { "ipa",         tINSTRN1, i_IPA_NAME },

 // flags
 { "unstressed",   tPHONEME_FLAG, phUNSTRESSED },
 { "nolink",       tPHONEME_FLAG, phNOLINK },
 { "brkafter",     tPHONEME_FLAG, phBRKAFTER },
```

```
 { "rhotic",       tPHONEME_FLAG, phRHOTIC },
 { "lengthenstop", tPHONEME_FLAG, phLENGTHENSTOP },
 { "nopause",      tPHONEME_FLAG, phNOPAUSE },
 { "prevoice",     tPHONEME_FLAG, phPREVOICE },

 { "flag1", tPHONEME_FLAG, phFLAG1 },
 { "flag2", tPHONEME_FLAG, phFLAG2 },

 // vowel transition attributes
 { "len=",   tTRANSITION,  1 },
 { "rms=",   tTRANSITION,  2 },
 { "f1=",    tTRANSITION,  3 },
 { "f2=",    tTRANSITION,  4 },
 { "f3=",    tTRANSITION,  5 },
 { "brk",    tTRANSITION,  6 },
 { "rate",   tTRANSITION,  7 },
 { "glstop", tTRANSITION,  8 },
 { "lenadd", tTRANSITION,  9 },
 { "f4",     tTRANSITION, 10 },
 { "gpaus",  tTRANSITION, 11 },
 { "colr=",  tTRANSITION, 12 },
 { "amp=",   tTRANSITION, 13 },  // set rms of 1st frame as
fraction of rms of 2nd frame  (1/30ths)

 { NULL, 0, -1 }
};

static keywtab_t *keyword_tabs[] = {
 keywords, k_conditions, k_properties, k_intonation
};

static PHONEME_TAB *phoneme_out;

static int n_phcodes_list[N_PHONEME_TABS];
static PHONEME_TAB_LIST phoneme_tab_list2[N_PHONEME_TABS];
static PHONEME_TAB *phoneme_tab2;
static int phoneme_flags;
```

```
#define N_PROCS 50
int n_procs;
int proc_addr[N_PROCS];
char proc_names[N_ITEM_STRING+1][N_PROCS];

#define MAX_PROG_BUF 2000
unsigned short *prog_out;
unsigned short *prog_out_max;
unsigned short prog_buf[MAX_PROG_BUF+20];

static espeak_ng_STATUS
ReadPhondataManifest(espeak_ng_ERROR_CONTEXT *context)
{
 // Read the phondata-manifest file
 FILE *f;
 int n_lines = 0;
 int ix;
 char *p;
 unsigned int value;
 char buf[sizeof(path_home)+40];
 char name[120];

 sprintf(buf, "%s%c%s", path_home, PATHSEP, "phondata-manifest");
 if ((f = fopen(buf, "r")) == NULL)
  return create_file_error_context(context, errno, buf);

 while (fgets(buf, sizeof(buf), f) != NULL)
  n_lines++;

 rewind(f);

 if (manifest != NULL) {
  for (ix = 0; ix < n_manifest; ix++)
   free(manifest[ix].name);
 }
```

```
 if (n_lines == 0) {
  fclose(f);
  return ENS_EMPTY_PHONEME_MANIFEST;
 }

 NAMETAB *new_manifest = (NAMETAB *)realloc(manifest, n_lines *
sizeof(NAMETAB));
 if (new_manifest == NULL) {
  fclose(f);
  free(manifest);
  return ENOMEM;
 } else
  manifest = new_manifest;

 n_manifest = 0;
 while (fgets(buf, sizeof(buf), f) != NULL) {
  if (!isalpha(buf[0]))
   continue;

  if (sscanf(&buf[2], "%x %s", &value, name) == 2) {
   if ((p = (char *)malloc(strlen(name)+1)) != NULL) {
    strcpy(p, name);
    manifest[n_manifest].value = value;
    manifest[n_manifest].name = p;
    n_manifest++;
   }
  }
 }
 fclose(f);

 return ENS_OK;
}

static int n_phoneme_tabs;
static int n_phcodes;

// outout files
```

```
static FILE *f_phdata;
static FILE *f_phindex;
static FILE *f_phtab;
static FILE *f_phcontents;
static FILE *f_errors = NULL;
static FILE *f_prog_log = NULL;
static FILE *f_report;

static FILE *f_in;
static int f_in_linenum;
static int f_in_displ;

static int linenum;
static int count_references = 0;
static int duplicate_references = 0;
static int count_frames = 0;
static int error_count = 0;
static int resample_count = 0;
static int resample_fails = 0;
static int then_count = 0;
static bool after_if = false;

static char current_fname[80];

static int markers_used[8];

typedef struct {
 void *link;
 int value;
 int ph_mnemonic;
 short ph_table;
 char string[1];
} REF_HASH_TAB;

static REF_HASH_TAB *ref_hash_tab[256];

#define N_ENVELOPES  30
```

```
int n_envelopes = 0;
char envelope_paths[N_ENVELOPES][80];
unsigned char envelope_dat[N_ENVELOPES][ENV_LEN];

typedef struct {
 FILE *file;
 int linenum;
 char fname[80];
} STACK;

#define N_STACK  12
int stack_ix;
STACK stack[N_STACK];

#define N_IF_STACK 12
int if_level;
typedef struct {
 unsigned short *p_then;
 unsigned short *p_else;
 bool returned;
} IF_STACK;
IF_STACK if_stack[N_IF_STACK];

enum {
 tENDFILE = 1,
 tSTRING,
 tNUMBER,
 tSIGNEDNUMBER,
 tPHONEMEMNEM,
 tOPENBRACKET,
 tKEYWORD,
 tCONDITION,
 tPROPERTIES,
 tINTONATION,
};

int item_type;
```

```
int item_terminator;
char item_string[N_ITEM_STRING];

static int ref_sorter(char **a, char **b)
{
 int ix;

 REF_HASH_TAB *p1 = (REF_HASH_TAB *)(*a);
 REF_HASH_TAB *p2 = (REF_HASH_TAB *)(*b);

 ix = strcoll(p1->string, p2->string);
 if (ix != 0)
  return ix;

 ix = p1->ph_table - p2->ph_table;
 if (ix != 0)
  return ix;

 return p1->ph_mnemonic - p2->ph_mnemonic;
}

static void CompileReport(void)
{
 int ix;
 int hash;
 int n;
 REF_HASH_TAB *p;
 REF_HASH_TAB **list;
 const char *data_path;
 int prev_table;
 int procedure_num;
 int prev_mnemonic;

 if (f_report == NULL || count_references == 0)
  return;

 // make a list of all the references and sort it
```

```
 list = (REF_HASH_TAB **)malloc((count_references)*
sizeof(REF_HASH_TAB *));
 if (list == NULL)
  return;

 fprintf(f_report, "\n%d phoneme tables\n", n_phoneme_tabs);
 fprintf(f_report, "          new total\n");
 for (ix = 0; ix < n_phoneme_tabs; ix++)
  fprintf(f_report, "%8s %3d %4d\n", phoneme_tab_list2[ix].name,
phoneme_tab_list2[ix].n_phonemes, n_phcodes_list[ix]+1);
 fputc('\n', f_report);

 fprintf(f_report, "Data file      Used by\n");
 ix = 0;
 for (hash = 0; (hash < 256) && (ix < count_references); hash++)
{
  p = ref_hash_tab[hash];
  while (p != NULL) {
   list[ix++] = p;
   p = (REF_HASH_TAB *)(p->link);
  }
 }
 n = ix;
 qsort((void *)list, n, sizeof(REF_HASH_TAB *), (int (*)(const
void *, const void *))ref_sorter);

 data_path = "";
 prev_mnemonic = 0;
 prev_table = 0;
 for (ix = 0; ix < n; ix++) {
  int j = 0;

  if (strcmp(list[ix]->string, data_path) != 0) {
   data_path = list[ix]->string;
   j = strlen(data_path);
   fprintf(f_report, "%s", data_path);
  } else if ((list[ix]->ph_table == prev_table) &&
```

```
(list[ix]->ph_mnemonic == prev_mnemonic))
   continue; // same phoneme, don't list twice

  while (j < 14) {
   fputc(' ', f_report); // pad filename with spaces
   j++;
  }

  prev_mnemonic = list[ix]->ph_mnemonic;
  if ((prev_mnemonic >> 24) == 'P') {
   // a procedure, not a phoneme
   procedure_num = atoi(WordToString(prev_mnemonic));
   fprintf(f_report, "  %s  %s", phoneme_tab_list2[prev_table =
list[ix]->ph_table].name, proc_names[procedure_num]);
  } else
   fprintf(f_report, "  [%s] %s", WordToString(prev_mnemonic),
phoneme_tab_list2[prev_table = list[ix]->ph_table].name);
  fputc('\n', f_report);
 }

 for (ix = 0; ix < n; ix++) {
  free(list[ix]);
  list[ix] = NULL;
 }

 free(list);
 list = NULL;
}

static void error(const char *format, ...)
{
 va_list args;
 va_start(args, format);

 fprintf(f_errors, "%s(%d): ", current_fname, linenum-1);
 vfprintf(f_errors, format, args);
 fprintf(f_errors, "\n");
```

```
 error_count++;

 va_end(args);
}

static void error_from_status(espeak_ng_STATUS status, const char
*context)
{
 char message[512];
 espeak_ng_GetStatusCodeMessage(status, message,
sizeof(message));
 if (context)
  error("%s: '%s'.", message, context);
 else
  error("%s.", message);
}

static unsigned int StringToWord(const char *string)
{
 // Pack 4 characters into a word
 int ix;
 unsigned char c;
 unsigned int word;

 if (string == NULL)
  return 0;

 word = 0;
 for (ix = 0; ix < 4; ix++) {
  if (string[ix] == 0) break;
  c = string[ix];
  word |= (c << (ix*8));
 }
 return word;
}

static MNEM_TAB reserved_phonemes[] = {
```

```
{ "_\001",  phonCONTROL },       // NOT USED
{ "%",      phonSTRESS_U },
{ "%%",     phonSTRESS_D },
{ ",",      phonSTRESS_2 },
{ ",,",     phonSTRESS_3 },
{ "'",      phonSTRESS_P },
{ "''",     phonSTRESS_P2 },
{ "=",      phonSTRESS_PREV },  //  stress previous syllable
{ "_:",     phonPAUSE },        //  pause
{ "_",      phonPAUSE_SHORT },  //  short pause
{ "_!",     phonPAUSE_NOLINK }, //  short pause, no link
{ ":",      phonLENGTHEN },
{ "@",      phonSCHWA },
{ "@-",     phonSCHWA_SHORT },
{ "||",     phonEND_WORD },
{ "1",      phonDEFAULTTONE },  // (numeral 1)  default tone
(for tone language)
{ "#X1",    phonCAPITAL },      // capital letter indication
{ "?",      phonGLOTTALSTOP },  // glottal stop
{ "-",      phonSYLLABIC },     // syllabic consonant
{ "_^_",    phonSWITCH },       //  Change language
{ "_X1",    phonX1 },           // a language specific action
{ "_|",     phonPAUSE_VSHORT }, // very short pause
{ "_::",    phonPAUSE_LONG },   // long pause
{ "t#",     phonT_REDUCED },    // reduced version of [t]
{ "'!",     phonSTRESS_TONIC }, // stress - emphasized
{ "_;_",    phonPAUSE_CLAUSE }, // clause pause

{ "#@",     phonVOWELTYPES },   // vowel type groups, these must
be consecutive
{ "#a",     phonVOWELTYPES+1 },
{ "#e",     phonVOWELTYPES+2 },
{ "#i",     phonVOWELTYPES+3 },
{ "#o",     phonVOWELTYPES+4 },
{ "#u",     phonVOWELTYPES+5 },

{ NULL, 0 }
```

```
};

static void ReservePhCodes()
{
 // Reserve phoneme codes which have fixed numbers so that they
can be
 // referred to from the program code.
 unsigned int word;
 MNEM_TAB *p;

 p = reserved_phonemes;
 while (p->mnem != NULL) {
  word = StringToWord(p->mnem);
  phoneme_tab2[p->value].mnemonic = word;
  phoneme_tab2[p->value].code = p->value;
  if (n_phcodes <= p->value)
   n_phcodes = p->value+1;
  p++;
 }
}

static int LookupPhoneme(const char *string, int control)
{
 // control = 0   explicit declaration
 // control = 1   declare phoneme if not found
 // control = 2   start looking after control & stress phonemes

 int ix;
 int start;
 int use;
 unsigned int word;

 if (strcmp(string, "NULL") == 0)
  return 1;

 ix = strlen(string);
 if ((ix == 0) || (ix > 4))
```

```
  error("Bad phoneme name '%s'", string);
 word = StringToWord(string);

 // don't use phoneme number 0, reserved for string terminator
 start = 1;

 if (control == 2) {
  // don't look for control and stress phonemes (allows these
characters to be
  // used for other purposes)
  start = 8;
 }

 use = 0;
 for (ix = start; ix < n_phcodes; ix++) {
  if (phoneme_tab2[ix].mnemonic == word)
   return ix;

  if ((use == 0) && (phoneme_tab2[ix].mnemonic == 0))
   use = ix;
 }

 if (use == 0) {
  if (control == 0)
   return -1;
  if (n_phcodes >= N_PHONEME_TAB-1)
   return -1; // phoneme table is full
  use = n_phcodes++;
 }

 // add this phoneme to the phoneme table
 phoneme_tab2[use].mnemonic = word;
 phoneme_tab2[use].type = phINVALID;
 phoneme_tab2[use].program = linenum; // for error report if the
phoneme remains undeclared
 return use;
}
```

```
static unsigned int get_char()
{
 unsigned int c;
 c = fgetc(f_in);
 if (c == '\n')
  linenum++;
 return c;
}

static void unget_char(unsigned int c)
{
 ungetc(c, f_in);
 if (c == '\n')
  linenum--;
}

static int CheckNextChar()
{
 int c;
 while (((c = get_char()) == ' ') || (c == '\t'))
  ;
 unget_char(c);
 return c;
}

static int NextItem(int type)
{
 int acc;
 unsigned char c = 0;
 unsigned char c2;
 int ix;
 int sign;
 char *p;
 keywtab_t *pk;

 item_type = -1;
```

```
f_in_displ = ftell(f_in);
f_in_linenum = linenum;

while (!feof(f_in)) {
 c = get_char();
 if (c == '/') {
  if ((c2 = get_char()) == '/') {
   // comment, ignore to end of line
   while (!feof(f_in) && ((c = get_char()) != '\n'))
    ;
  } else
   unget_char(c2);
 }
 if (!isspace(c))
  break;
}
if (feof(f_in))
 return -2;

if (c == '(') {
 if (type == tOPENBRACKET)
  return 1;
 return -1;
}

ix = 0;
while (!feof(f_in) && !isspace(c) && (c != '(') && (c != ')') &&
(c != ',')) {
 if (c == '\\')
  c = get_char();
 item_string[ix++] = c;
 c = get_char();
 if (feof(f_in))
  break;
 if (item_string[ix-1] == '=')
  break;
```

```
}
item_string[ix] = 0;

while (isspace(c))
 c = get_char();

item_terminator = ' ';
if ((c == ')') || (c == '(') || (c == ','))
 item_terminator = c;

if ((c == ')') || (c == ','))
 c = ' ';
else if (!feof(f_in))
 unget_char(c);

if (type == tSTRING)
 return 0;

if ((type == tNUMBER) || (type == tSIGNEDNUMBER)) {
 acc = 0;
 sign = 1;
 p = item_string;

 if ((*p == '-') && (type == tSIGNEDNUMBER)) {
  sign = -1;
  p++;
 }
 if (!isdigit(*p)) {
  if ((type == tNUMBER) && (*p == '-'))
   error("Expected an unsigned number");
  else
   error("Expected a number");
 }
 while (isdigit(*p)) {
  acc *= 10;
  acc += (*p - '0');
  p++;
```

```
  }
  return acc * sign;
 }

 if ((type >= tKEYWORD) && (type <= tINTONATION)) {
  pk = keyword_tabs[type-tKEYWORD];
  while (pk->mnem != NULL) {
   if (strcmp(item_string, pk->mnem) == 0) {
    item_type = pk->type;
    return pk->data;
   }
   pk++;
  }
  item_type = -1;
  return -1; // keyword not found
 }
 if (type == tPHONEMEMNEM)
  return LookupPhoneme(item_string, 2);
 return -1;
}

static int NextItemMax(int max)
{
 // Get a number, but restrict value to max
 int value;

 value = NextItem(tNUMBER);
 if (value > max) {
  error("Value %d is greater than maximum %d", value, max);
  value = max;
 }
 return value;
}

static int NextItemBrackets(int type, int control)
{
 // Expect a parameter inside parantheses
```

```
 // control: bit 0  0= need (
 //          bit 1  1= allow comma

 int value;

 if ((control & 1) == 0) {
  if (!NextItem(tOPENBRACKET))
   error("Expected '('");
 }

 value = NextItem(type);
 if ((control & 2) && (item_terminator == ','))
  return value;

 if (item_terminator != ')')
  error("Expected ')'");
 return value;
}

static void UngetItem()
{
 fseek(f_in, f_in_displ, SEEK_SET);
 linenum = f_in_linenum;
}

static int Range(int value, int divide, int min, int max)
{
 if (value < 0)
  value -= divide/2;
 else
  value += divide/2;
 value = value / divide;

 if (value > max)
  value = max;
 if (value < min)
  value = min;
```

```
 return value - min;
}

static int CompileVowelTransition(int which)
{
 // Compile a vowel transition
 int key;
 int len = 0;
 int rms = 0;
 int f1 = 0;
 int f2 = 0;
 int f2_min = 0;
 int f2_max = 0;
 int f3_adj = 0;
 int f3_amp = 0;
 int flags = 0;
 int vcolour = 0;
 int x;
 int instn = i_VOWELIN;
 int word1;
 int word2;

 if (which == 1) {
  len = 50 / 2; // defaults for transition into vowel
  rms = 25 / 2;

  if (phoneme_out->type == phSTOP) {
   len = 42 / 2; // defaults for transition into vowel
   rms = 30 / 2;
  }
 } else if (which == 2) {
  instn = i_VOWELOUT;
  len = 36 / 2; // defaults for transition out of vowel
  rms = 16 / 2;
 }

 for (;;) {
```

```
key = NextItem(tKEYWORD);
if (item_type != tTRANSITION) {
 UngetItem();
 break;
}

switch (key & 0xf)
{
case 1:
 len = Range(NextItem(tNUMBER), 2, 0, 63) & 0x3f;
 flags |= 1;
 break;
case 2:
 rms = Range(NextItem(tNUMBER), 2, 0, 31) & 0x1f;
 flags |= 1;
 break;
case 3:
 f1 = NextItem(tNUMBER);
 break;
case 4:
 f2 = Range(NextItem(tNUMBER), 50, 0, 63) & 0x3f;
 f2_min = Range(NextItem(tSIGNEDNUMBER), 50, -15, 15) & 0x1f;
 f2_max = Range(NextItem(tSIGNEDNUMBER), 50, -15, 15) & 0x1f;
 if (f2_min > f2_max) {
  x = f2_min;
  f2_min = f2_max;
  f2_max = x;
 }
 break;
case 5:
 f3_adj = Range(NextItem(tSIGNEDNUMBER), 50, -15, 15) & 0x1f;
 f3_amp = Range(NextItem(tNUMBER), 8, 0, 15) & 0x1f;
 break;
case 6:
 flags |= 2; // break
 break;
case 7:
```

```
   flags |= 4; // rate
   break;
  case 8:
   flags |= 8; // glstop
   break;
  case 9:
   flags |= 16; // lenadd
   break;
  case 10:
   flags |= 32;  // f4
   break;
  case 11:
   flags |= 64;  // pause
   break;
  case 12:
   vcolour = NextItem(tNUMBER);
   break;
  case 13:
   // set rms of 1st frame as fraction of rms of 2nd frame
(1/30ths)
   rms = (Range(NextItem(tNUMBER), 1, 0, 31) & 0x1f) | 0x20;
   flags |= 1;
   break;
  }
 }
 word1 = len + (rms << 6) + (flags << 12);
 word2 =  f2 + (f2_min << 6) + (f2_max << 11) + (f3_adj << 16) +
(f3_amp << 21) + (f1 << 26) + (vcolour << 29);
 prog_out[0] = instn + ((word1 >> 16) & 0xff);
 prog_out[1] = word1;
 prog_out[2] = word2 >> 16;
 prog_out[3] = word2;
 prog_out += 4;

 return 0;
}
```

```
static espeak_ng_STATUS LoadSpect(const char *path, int control,
int *addr)
{
 SpectSeq *spectseq;
 int peak;
 int frame;
 int n_frames;
 int ix;
 int x, x2;
 int rms;
 float total;
 float pkheight;
 int marker1_set = 0;
 int frame_vowelbreak = 0;
 int klatt_flag = 0;
 SpectFrame *fr;
 frame_t *fr_out;
 char filename[sizeof(path_home)+20];

 SPECT_SEQ seq_out;
 SPECT_SEQK seqk_out;

 // create SpectSeq and import data
 spectseq = SpectSeqCreate();
 if (spectseq == NULL)
  return ENOMEM;

 snprintf(filename, sizeof(filename), "%s/%s", phsrc, path);
 espeak_ng_STATUS status = LoadSpectSeq(spectseq, filename);
 if (status != ENS_OK) {
  error("Bad vowel file: '%s'", path);
  SpectSeqDestroy(spectseq);
  return status;
 }

 // do we need additional klatt data ?
 for (frame = 0; frame < spectseq->numframes; frame++) {
```

```
  for (ix = 5; ix < N_KLATTP2; ix++) {
   if (spectseq->frames[frame]->klatt_param[ix] != 0)
    klatt_flag = FRFLAG_KLATT;
  }
 }

 seq_out.n_frames = 0;
 seq_out.sqflags = 0;
 seq_out.length_total = 0;

 total = 0;
 for (frame = 0; frame < spectseq->numframes; frame++) {
  if (spectseq->frames[frame]->keyframe) {
   if (seq_out.n_frames == 1)
    frame_vowelbreak = frame;
   if (spectseq->frames[frame]->markers & 0x2) {
    // marker 1 is set
    marker1_set = 1;
   }

   seq_out.n_frames++;
   if (frame > 0)
    total += spectseq->frames[frame-1]->length;
  }
 }
 seq_out.length_total = (int)total;

 if ((control & 1) && (marker1_set == 0)) {
  // This is a vowel, but no Vowel Break marker is set
  // set a marker flag for the second frame of a vowel
  spectseq->frames[frame_vowelbreak]->markers |=
FRFLAG_VOWEL_CENTRE;
 }

 n_frames = 0;
 for (frame = 0; frame < spectseq->numframes; frame++) {
  fr = spectseq->frames[frame];
```

```
  if (fr->keyframe) {
   if (klatt_flag)
    fr_out = &seqk_out.frame[n_frames];
   else
    fr_out = (frame_t *)&seq_out.frame[n_frames];

   x = (int)(fr->length + 0.5); // round to nearest mS
   if (x > 255) x = 255;
   fr_out->length = x;

   fr_out->frflags = fr->markers | klatt_flag;

   rms = (int)GetFrameRms(fr, spectseq->amplitude);
   if (rms > 255) rms = 255;
   fr_out->rms = rms;

   if (n_frames == (seq_out.n_frames-1))
    fr_out->length = 0; // give last frame zero length

   // write: peak data
   count_frames++;
   for (peak = 0; peak < 8; peak++) {
    if (peak < 7)
     fr_out->ffreq[peak] = fr->peaks[peak].pkfreq;

    pkheight = spectseq->amplitude * fr->amp_adjust *
fr->peaks[peak].pkheight;
    pkheight = pkheight/640000;
    if (pkheight > 255) pkheight = 255;
    fr_out->fheight[peak] = (int)pkheight;

    if (peak < 6) {
     x =  fr->peaks[peak].pkwidth/4;
     if (x > 255) x = 255;
     fr_out->fwidth[peak] = x;
```

```
   if (peak < 3) {
    x2 =  fr->peaks[peak].pkright/4;
    if (x2 > 255) x2 = 255;
    fr_out->fright[peak] = x2;
   }
  }

  if (peak < 4) {
   x = fr->peaks[peak].klt_bw / 2;
   if (x > 255) x = 255;
   fr_out->bw[peak] = x;
  }
 }

 for (ix = 0; ix < 5; ix++) {
  fr_out->klattp[ix] = fr->klatt_param[ix];

  fr_out->klattp[KLATT_FNZ] = fr->klatt_param[KLATT_FNZ] / 2;
 }

 if (klatt_flag) {
  // additional klatt parameters
  for (ix = 0; ix < 5; ix++)
   fr_out->klattp2[ix] = fr->klatt_param[ix+5];

  for (peak = 0; peak < 7; peak++) {
   fr_out->klatt_ap[peak] = fr->peaks[peak].klt_ap;

   x = fr->peaks[peak].klt_bp / 2;
   if (x > 255) x = 255;
   fr_out->klatt_bp[peak] = x;
  }
  fr_out->spare = 0;
 }

 if (fr_out->bw[1] == 0) {
  fr_out->bw[0] = 89 / 2;
```

```
    fr_out->bw[1] = 90 / 2;
    fr_out->bw[2] = 140 / 2;
    fr_out->bw[3] = 260 / 2;
   }

   n_frames++;
  }
 }

 if (klatt_flag) {
  seqk_out.n_frames = seq_out.n_frames;
  seqk_out.sqflags = seq_out.sqflags;
  seqk_out.length_total = seq_out.length_total;

  ix = (char *)(&seqk_out.frame[seqk_out.n_frames]) - (char
*)(&seqk_out);
  fwrite(&seqk_out, ix, 1, f_phdata);
  while (ix & 3)
  {
   // round up to multiple of 4 bytes
   fputc(0, f_phdata);
   ix++;
  }
 } else {
  ix = (char *)(&seq_out.frame[seq_out.n_frames]) - (char
*)(&seq_out);
  fwrite(&seq_out, ix, 1, f_phdata);
  while (ix & 3)
  {
   // round up to multiple of 4 bytes
   fputc(0, f_phdata);
   ix++;
  }
 }

 SpectSeqDestroy(spectseq);
 return ENS_OK;
```

```
}

static int LoadWavefile(FILE *f, const char *fname)
{
 int displ;
 unsigned char c1;
 unsigned char c3;
 int c2;
 int sample;
 int sample2;
 float x;
 int max = 0;
 int length;
 int sr1, sr2;
 bool failed;
 int len;
 bool resample_wav = false;
 const char *fname2;
 char fname_temp[100];
 char msg[120];
 int scale_factor = 0;

 fseek(f, 24, SEEK_SET);
 sr1 = Read4Bytes(f);
 sr2 = Read4Bytes(f);
 fseek(f, 40, SEEK_SET);

 if ((sr1 != samplerate_native) || (sr2 != sr1*2)) {
  int fd_temp;
  char command[sizeof(path_home)+250];

  failed = false;

#ifdef HAVE_MKSTEMP
  strcpy(fname_temp, "/tmp/espeakXXXXXX");
  if ((fd_temp = mkstemp(fname_temp)) >= 0)
   close(fd_temp);
```

```
#else
  strcpy(fname_temp, tmpnam(NULL));
#endif

  fname2 = fname;
  len = strlen(fname);
  if (strcmp(&fname[len-4], ".wav") == 0) {
   strcpy(msg, fname);
   msg[len-4] = 0;
   fname2 = msg;
  }

  sprintf(command, "sox \"%s/%s.wav\" -r %d -c1 -t wav %s\n",
phsrc, fname2, samplerate_native, fname_temp);
  if (system(command) != 0)
   failed = true;

  if (failed || (GetFileLength(fname_temp) <= 0)) {
   if (resample_fails < 2)
    error("Resample command failed: %s", command);
   resample_fails++;

   if (sr1 != samplerate_native)
    error("Can't resample (%d to %d): %s", sr1,
samplerate_native, fname);
   else
    error("WAV file is not mono: %s", fname);
   remove(fname_temp);
   return 0;
  }

  f = fopen(fname_temp, "rb");
  if (f == NULL) {
   error("Can't read temp file: %s", fname_temp);
   return 0;
  }
  if (f_report != NULL)
```

```
   fprintf(f_report, "resampled  %s\n", fname);
  resample_count++;
  resample_wav = true;
  fseek(f, 40, SEEK_SET); // skip past the WAV header, up to
before "data length"
 }

 displ = ftell(f_phdata);

 // data contains:  4 bytes of length (n_samples * 2), followed
by 2-byte samples (lsb byte first)
 length = Read4Bytes(f);

 while (true) {
  int c;

  if ((c = fgetc(f)) == EOF)
   break;
  c1 = (unsigned char)c;

  if ((c = fgetc(f)) == EOF)
   break;
  c3 = (unsigned char)c;

  c2 = c3 << 24;
  c2 = c2 >> 16; // sign extend

  sample = (c1 & 0xff) + c2;

  if (sample > max)
   max = sample;
  else if (sample < -max)
   max = -sample;
 }

 scale_factor = (max / 127) + 1;
```

```
#define MIN_FACTOR   -1 // was 6, disable use of 16 bit samples
if (scale_factor > MIN_FACTOR) {
 length = length/2 + (scale_factor << 16);
}

Write4Bytes(f_phdata, length);
fseek(f, 44, SEEK_SET);

while (!feof(f)) {
 c1 = fgetc(f);
 c3 = fgetc(f);
 c2 = c3 << 24;
 c2 = c2 >> 16; // sign extend

 sample = (c1 & 0xff) + c2;

 if (feof(f)) break;

 if (scale_factor <= MIN_FACTOR) {
  fputc(sample & 0xff, f_phdata);
  fputc(sample >> 8, f_phdata);
 } else {
  x = ((float)sample / scale_factor) + 0.5;
  sample2 = (int)x;
  if (sample2 > 127)
   sample2 = 127;
  if (sample2 < -128)
   sample2 = -128;
  fputc(sample2, f_phdata);
 }
}

length = ftell(f_phdata);
while ((length & 3) != 0) {
 // pad to a multiple of 4 bytes
 fputc(0, f_phdata);
 length++;
```

```
 }

 if (resample_wav == true) {
  fclose(f);
  remove(fname_temp);
 }
 return displ | 0x800000; // set bit 23 to indicate a wave file
rather than a spectrum
}

static espeak_ng_STATUS LoadEnvelope(FILE *f, const char *fname,
int *displ)
{
 char buf[128];

 if (displ)
  *displ = ftell(f_phdata);

 if (fseek(f, 12, SEEK_SET) == -1)
  return errno;

 if (fread(buf, 128, 1, f) != 128)
  return errno;
 fwrite(buf, 128, 1, f_phdata);

 if (n_envelopes < N_ENVELOPES) {
  strncpy0(envelope_paths[n_envelopes], fname,
sizeof(envelope_paths[0]));
  memcpy(envelope_dat[n_envelopes], buf,
sizeof(envelope_dat[0]));
  n_envelopes++;
 }

 return ENS_OK;
}

// Generate a hash code from the specified string
```

```
static int Hash8(const char *string)
{
 int c;
 int chars = 0;
 int hash = 0;

 while ((c = *string++) != 0) {
  c = tolower(c) - 'a';
  hash = hash * 8 + c;
  hash = (hash & 0x1ff) ^ (hash >> 8); // exclusive or
  chars++;
 }

 return (hash+chars) & 0xff;
}

static int LoadEnvelope2(FILE *f, const char *fname)
{
 int ix, ix2;
 int n;
 int x, y;
 int displ;
 int n_points;
 double yy;
 char line_buf[128];
 float env_x[20];
 float env_y[20];
 int env_lin[20];
 unsigned char env[ENV_LEN];

 n_points = 0;
 if (fgets(line_buf, sizeof(line_buf), f) != NULL) { ; // skip
first line, then loop
  while (!feof(f)) {
   if (fgets(line_buf, sizeof(line_buf), f) == NULL)
    break;
```

```
   env_lin[n_points] = 0;
   n = sscanf(line_buf, "%f %f %d", &env_x[n_points],
&env_y[n_points], &env_lin[n_points]);
   if (n >= 2) {
    env_x[n_points] *= (float)1.28; // convert range 0-100 to
0-128
    n_points++;
   }
  }
 }
 if (n_points > 0) {
  env_x[n_points] = env_x[n_points-1];
  env_y[n_points] = env_y[n_points-1];
 }

 ix = -1;
 ix2 = 0;
 if (n_points > 0) for (x = 0; x < ENV_LEN; x++) {
  if (n_points > 3 && x > env_x[ix+4])
   ix++;
  if (n_points > 2 && x >= env_x[ix2+1])
   ix2++;

  if (env_lin[ix2] > 0) {
   yy = env_y[ix2] + (env_y[ix2+1] - env_y[ix2]) * ((float)x -
env_x[ix2]) / (env_x[ix2+1] - env_x[ix2]);
   y = (int)(yy * 2.55);
  } else if (n_points > 3)
   y = (int)(polint(&env_x[ix], &env_y[ix], 4, x) * 2.55); //
convert to range 0-255
  else
   y = (int)(polint(&env_x[ix], &env_y[ix], 3, x) * 2.55);
  if (y < 0) y = 0;
  if (y > 255) y = 255;
  env[x] = y;
 }
```

```
 if (n_envelopes < N_ENVELOPES) {
  strncpy0(envelope_paths[n_envelopes], fname,
sizeof(envelope_paths[0]));
  memcpy(envelope_dat[n_envelopes], env, ENV_LEN);
  n_envelopes++;
 }

 displ = ftell(f_phdata);
 fwrite(env, 1, ENV_LEN, f_phdata);

 return displ;
}

static espeak_ng_STATUS LoadDataFile(const char *path, int
control, int *addr)
{
 // load spectrum sequence or sample data from a file.
 // return index into spect or sample data area. bit 23=1 if a
sample

 FILE *f;
 int id;
 int hash;
 int type_code = ' ';
 REF_HASH_TAB *p, *p2;
 char buf[sizeof(path_home)+150];

 if (strcmp(path, "NULL") == 0)
  return ENS_OK;
 if (strcmp(path, "DFT") == 0) {
  *addr = 1;
  return ENS_OK;
 }

 count_references++;

 hash = Hash8(path);
```

```
p = ref_hash_tab[hash];
while (p != NULL) {
 if (strcmp(path, p->string) == 0) {
  duplicate_references++;
  *addr = p->value; // already loaded this data
  break;
 }
 p = (REF_HASH_TAB *)p->link;
}

if (*addr == 0) {
 sprintf(buf, "%s/%s", phsrc, path);

 if ((f = fopen(buf, "rb")) == NULL) {
  sprintf(buf, "%s/%s.wav", phsrc, path);
  if ((f = fopen(buf, "rb")) == NULL) {
   error("Can't read file: %s", path);
   return errno;
  }
 }

 id = Read4Bytes(f);
 rewind(f);

 espeak_ng_STATUS status = ENS_OK;
 if (id == 0x43455053) {
  status = LoadSpect(path, control, addr);
  type_code = 'S';
 } else if (id == 0x46464952) {
  *addr = LoadWavefile(f, path);
  type_code = 'W';
 } else if (id == 0x43544950) {
  status = LoadEnvelope(f, path, addr);
  type_code = 'E';
 } else if (id == 0x45564E45) {
  *addr = LoadEnvelope2(f, path);
  type_code = 'E';
```

```
  } else {
   error("File not SPEC or RIFF: %s", path);
   *addr = -1;
   status = ENS_UNSUPPORTED_PHON_FORMAT;
  }
  fclose(f);

  if (status != ENS_OK)
   return status;

  if (*addr > 0)
   fprintf(f_phcontents, "%c  0x%.5x  %s\n", type_code, *addr &
0x7fffff, path);
 }

 // add this item to the hash table
 if (*addr > 0) {
  p = ref_hash_tab[hash];
  p2 = (REF_HASH_TAB
*)malloc(sizeof(REF_HASH_TAB)+strlen(path)+1);
  if (p2 == NULL)
   return ENOMEM;
  p2->value = *addr;
  p2->ph_mnemonic = phoneme_out->mnemonic; // phoneme which uses
this file
  p2->ph_table = n_phoneme_tabs-1;
  strcpy(p2->string, path);
  p2->link = (char *)p;
  ref_hash_tab[hash] = p2;
 }

 return ENS_OK;
}

static void CompileToneSpec(void)
{
 int pitch1 = 0;
```

```
 int pitch2 = 0;
 int pitch_env = 0;
 int amp_env = 0;

 pitch1 = NextItemBrackets(tNUMBER, 2);
 pitch2 = NextItemBrackets(tNUMBER, 3);

 if (item_terminator == ',') {
  NextItemBrackets(tSTRING, 3);
  LoadDataFile(item_string, 0, &pitch_env);
 }

 if (item_terminator == ',') {
  NextItemBrackets(tSTRING, 1);
  LoadDataFile(item_string, 0, &amp_env);
 }

 if (pitch1 < pitch2) {
  phoneme_out->start_type = pitch1;
  phoneme_out->end_type = pitch2;
 } else {
  phoneme_out->start_type = pitch2;
  phoneme_out->end_type = pitch1;
 }

 if (pitch_env != 0) {
  *prog_out++ = i_PITCHENV + ((pitch_env >> 16) & 0xf);
  *prog_out++ = pitch_env;
 }
 if (amp_env != 0) {
  *prog_out++ = i_AMPENV + ((amp_env >> 16) & 0xf);
  *prog_out++ = amp_env;
 }
}

static void CompileSound(int keyword, int isvowel)
{
```

```
 int addr = 0;
 int value = 0;
 char path[N_ITEM_STRING];
 static int sound_instns[] = { i_FMT, i_WAV, i_VWLSTART,
i_VWLENDING, i_WAVADD };

 NextItemBrackets(tSTRING, 2);
 strcpy(path, item_string);
 if (item_terminator == ',') {
  if ((keyword == kVOWELSTART) || (keyword == kVOWELENDING)) {
   value = NextItemBrackets(tSIGNEDNUMBER, 1);
   if (value > 127) {
    value = 127;
    error("Parameter > 127");
   }
   if (value < -128) {
    value = -128;
    error("Parameter < -128");
   }
  } else {
   value = NextItemBrackets(tNUMBER, 1);
   if (value > 255) {
    value = 255;
    error("Parameter > 255");
   }
  }
 }
 LoadDataFile(path, isvowel, &addr);
 addr = addr / 4; // addr is words not bytes

}

   Condition
   bits 14,15   1
   bit 13       1 = AND, 0 = OR
   bit 12       spare
   bit 8-11
```

```
   =0-3       p,t,n,n2   data=phoneme code
   =4-7       p,t,n,n2   data=(bits5-7: phtype, place, property,
special) (bits0-4: data)
   =8         data = stress bitmap
   =9         special tests

static int CompileIf(int elif)
{
 int key;
 bool finish = false;
 int word = 0;
 int word2;
 int data;
 int bitmap;
 int brackets;
 bool not_flag;
 unsigned short *prog_last_if = NULL;

 then_count = 2;
 after_if = true;

 while (!finish) {
  not_flag = false;
  word2 = 0;
  if (prog_out >= prog_out_max) {
   error("Phoneme program too large");
   return 0;
  }

  if ((key = NextItem(tCONDITION)) < 0)
   error("Expected a condition, not '%s'", item_string);

  if ((item_type == 0) && (key == k_NOT)) {
   not_flag = true;
   if ((key = NextItem(tCONDITION)) < 0)
    error("Expected a condition, not '%s'", item_string);
  }
```

```
  if (item_type == tWHICH_PHONEME) {
   // prevPh(), thisPh(), nextPh(), next2Ph() etc
   if (key >= 6) {
    // put the 'which' code in the next instruction
    word2 = key;
    key = 6;
   }
   key = key << 8;

   data = NextItemBrackets(tPROPERTIES, 0);
   if (data >= 0)
    word = key + data + 0x700;
   else {
    data = LookupPhoneme(item_string, 2);
    word = key + data;
   }
  } else if (item_type == tTEST) {
   if (key == kTHISSTRESS) {
    bitmap = 0;
    brackets = 2;
    do {
     data = NextItemBrackets(tNUMBER, brackets);
     if (data > 7)
      error("Expected list of stress levels");
     bitmap |= (1 << data);

     brackets = 3;
    } while (item_terminator == ',');
    word = i_StressLevel | bitmap;
   } else
    word = key;
  } else {
   error("Unexpected keyword '%s'", item_string);

   if ((strcmp(item_string, "phoneme") == 0) ||
(strcmp(item_string, "endphoneme") == 0))
```

```
    return -1;
  }

  // output the word
  prog_last_if = prog_out;
  *prog_out++ = word | i_CONDITION;

  if (word2 != 0)
   *prog_out++ = word2;
  if (not_flag)
   *prog_out++ = i_NOT;

  // expect AND, OR, THEN
  switch (NextItem(tCONDITION))
  {
  case k_AND:
   break;
  case k_OR:
   if (prog_last_if != NULL)
    *prog_last_if |=  i_OR;
   break;
  case k_THEN:
   finish = true;
   break;
  default:
   error("Expected AND, OR, THEN");
   break;
  }
 }

 if (elif == 0) {
  if_level++;
  if_stack[if_level].p_else = NULL;
 }

 if_stack[if_level].returned = false;
 if_stack[if_level].p_then = prog_out;
```

```
 return 0;
}

static void FillThen(int add)
{
 unsigned short *p;
 int offset;

 p = if_stack[if_level].p_then;
 if (p != NULL) {
  offset = prog_out - p + add;

  if ((then_count == 1) && (if_level == 1)) {
   // The THEN part only contains one statement, we can remove
the THEN jump
   // and the interpreter will implicitly skip the statement.
   while (p < prog_out) {
    p[0] = p[1];
    p++;
   }
   prog_out--;
  } else {
   if (offset > MAX_JUMP)
    error("IF block is too long");
   *p = i_JUMP_FALSE + offset;
  }
  if_stack[if_level].p_then = NULL;
 }

 then_count = 0;
}

static int CompileElse(void)
{
 unsigned short *ref;
 unsigned short *p;
```

```
 if (if_level < 1) {
  error("ELSE not expected");
  return 0;
 }

 if (if_stack[if_level].returned == false)
  FillThen(1);
 else
  FillThen(0);

 if (if_stack[if_level].returned == false) {
  ref = prog_out;
  *prog_out++ = 0;

  if ((p = if_stack[if_level].p_else) != NULL)
   *ref = ref - p; // backwards offset to the previous else
  if_stack[if_level].p_else = ref;
 }

 return 0;
}

static int CompileElif(void)
{
 if (if_level < 1) {
  error("ELIF not expected");
  return 0;
 }

 CompileElse();
 CompileIf(1);
 return 0;
}

static int CompileEndif(void)
{
```

```
 unsigned short *p;
 int chain;
 int offset;

 if (if_level < 1) {
  error("ENDIF not expected");
  return 0;
 }

 FillThen(0);

 if ((p = if_stack[if_level].p_else) != NULL) {
  do {
   chain = *p; // a chain of previous else links

   offset = prog_out - p;
   if (offset > MAX_JUMP)
    error("IF block is too long");
   *p = i_JUMP + offset;

   p -= chain;
  } while (chain > 0);
 }

 if_level--;
 return 0;
}

static int CompileSwitch(int type)
{
 // Type 0:  EndSwitch
 //      1:  SwitchPrevVowelType
 //      2:  SwitchNextVowelType

 if (type == 0) {
  // check the instructions in the Switch
  return 0;
```

```
 }

 if (type == 1)
  *prog_out++ = i_SWITCH_PREVVOWEL+6;
 if (type == 2)
  *prog_out++ = i_SWITCH_NEXTVOWEL+6;
 return 0;
}

static PHONEME_TAB_LIST *FindPhonemeTable(const char *string)
{
 int ix;

 for (ix = 0; ix < n_phoneme_tabs; ix++) {
  if (strcmp(phoneme_tab_list2[ix].name, string) == 0)
   return &phoneme_tab_list2[ix];
 }
 error("Unknown phoneme table: '%s'", string);
 return NULL;
}

static PHONEME_TAB *FindPhoneme(const char *string)
{
 PHONEME_TAB_LIST *phtab = NULL;
 int ix;
 unsigned int mnem;
 char *phname;
 char buf[200];

 // is this the name of a phoneme which is in scope
 if ((strlen(string) <= 4) && ((ix = LookupPhoneme(string, 0)) !=
-1))
  return &phoneme_tab2[ix];

 // no, treat the name as phonemetable/phoneme
 strcpy(buf, string);
 if ((phname = strchr(buf, '/')) != 0)
```

```
  *phname++ = 0;

 phtab = FindPhonemeTable(buf);
 if (phtab == NULL)
  return NULL; // phoneme table not found

 mnem = StringToWord(phname);
 for (ix = 1; ix < 256; ix++) {
  if (mnem == phtab->phoneme_tab_ptr[ix].mnemonic)
   return &phtab->phoneme_tab_ptr[ix];
 }

 error("Phoneme reference not found: '%s'", string);
 return NULL;
}

static void ImportPhoneme(void)
{
 unsigned int ph_mnem;
 unsigned int ph_code;
 PHONEME_TAB *ph;

 NextItem(tSTRING);

 if ((ph = FindPhoneme(item_string)) == NULL) {
  error("Cannot find phoneme '%s' to import.", item_string);
  return;
 }

 if (phoneme_out->phflags != 0 ||
     phoneme_out->type != phINVALID ||
     phoneme_out->start_type != 0 ||
     phoneme_out->end_type != 0 ||
     phoneme_out->std_length != 0 ||
     phoneme_out->length_mod != 0) {
  error("Phoneme import will override set properties.");
 }
```

```
 ph_mnem = phoneme_out->mnemonic;
 ph_code = phoneme_out->code;
 memcpy(phoneme_out, ph, sizeof(PHONEME_TAB));
 phoneme_out->mnemonic = ph_mnem;
 phoneme_out->code = ph_code;
 if (phoneme_out->type != phVOWEL)
  phoneme_out->end_type = 0;  // voicingswitch, this must be set
later to refer to a local phoneme
}

static void CallPhoneme(void)
{
 PHONEME_TAB *ph;
 int ix;
 int addr = 0;

 NextItem(tSTRING);

 // first look for a procedure name
 for (ix = 0; ix < n_procs; ix++) {
  if (strcmp(proc_names[ix], item_string) == 0) {
   addr = proc_addr[ix];
   break;
  }
 }
 if (ix == n_procs) {
  // procedure not found, try a phoneme name
  if ((ph = FindPhoneme(item_string)) == NULL)
   return;
  addr = ph->program;

  if (phoneme_out->type == phINVALID) {
   // Phoneme type has not been set. Copy it from the called
phoneme
   phoneme_out->type = ph->type;
   phoneme_out->start_type = ph->start_type;
```

```
   phoneme_out->end_type = ph->end_type;
   phoneme_out->std_length = ph->std_length;
   phoneme_out->length_mod = ph->length_mod;

   phoneme_flags = ph->phflags & ~phARTICULATION;
  }
 }

}

static void DecThenCount()
{
 if (then_count > 0)
  then_count--;
}

static int CompilePhoneme(int compile_phoneme)
{
 int endphoneme = 0;
 int keyword;
 int value;
 int phcode = 0;
 int flags;
 int ix;
 int start;
 int count;
 int c;
 char *p;
 int vowel_length_factor = 100; // for testing
 char number_buf[12];
 char ipa_buf[N_ITEM_STRING+1];
 PHONEME_TAB phoneme_out2;
 PHONEME_PROG_LOG phoneme_prog_log;

 prog_out = prog_buf;
 prog_out_max = &prog_buf[MAX_PROG_BUF-1];
 if_level = 0;
```

```
 if_stack[0].returned = false;
 after_if = false;
 phoneme_flags = 0;

 NextItem(tSTRING);
 if (compile_phoneme) {
  phcode = LookupPhoneme(item_string, 1); // declare phoneme if
not already there
  if (phcode == -1) return 0;
  phoneme_out = &phoneme_tab2[phcode];
 } else {
  // declare a procedure
  if (n_procs >= N_PROCS) {
   error("Too many procedures");
   return 0;
  }
  strcpy(proc_names[n_procs], item_string);
  phoneme_out = &phoneme_out2;
  sprintf(number_buf, "%.3dP", n_procs);
  phoneme_out->mnemonic = StringToWord(number_buf);
 }

 phoneme_out->code = phcode;
 phoneme_out->program = 0;
 phoneme_out->type = phINVALID;
 phoneme_out->std_length = 0;
 phoneme_out->start_type = 0;
 phoneme_out->end_type = 0;
 phoneme_out->length_mod = 0;
 phoneme_out->phflags = 0;

 while (!endphoneme && !feof(f_in)) {
  if ((keyword = NextItem(tKEYWORD)) < 0) {
   if (keyword == -2) {
    error("Missing 'endphoneme' before end-of-file"); // end of
file
    break;
```

```
    }

    phoneme_feature_t feature =
phoneme_feature_from_string(item_string);
    espeak_ng_STATUS status = phoneme_add_feature(phoneme_out,
feature);
    if (status == ENS_OK)
     continue;
    error_from_status(status, item_string);
    continue;
   }

   switch (item_type)
   {
   case tPHONEME_TYPE:
    if (phoneme_out->type != phINVALID) {
     if (phoneme_out->type == phFRICATIVE && keyword == phLIQUID)
      ; // apr liquid => ok
     else
      error("More than one phoneme type: %s", item_string);
    }
    phoneme_out->type = keyword;
    break;
   case tPHONEME_FLAG:
    phoneme_flags |= keyword;
    break;
   case tINSTRN1:
    // instruction group 0, with 8 bit operands which set data in
PHONEME_DATA
    switch (keyword)
    {
    case i_CHANGE_PHONEME:
    case i_APPEND_PHONEME:
    case i_APPEND_IFNEXTVOWEL:
    case i_INSERT_PHONEME:
    case i_REPLACE_NEXT_PHONEME:
    case i_VOICING_SWITCH:
```

```
case i_CHANGE_IF | STRESS_IS_DIMINISHED:
case i_CHANGE_IF | STRESS_IS_UNSTRESSED:
case i_CHANGE_IF | STRESS_IS_NOT_STRESSED:
case i_CHANGE_IF | STRESS_IS_SECONDARY:
case i_CHANGE_IF | STRESS_IS_PRIMARY:
 value = NextItemBrackets(tPHONEMEMNEM, 0);
 *prog_out++ = (keyword << 8) + value;
 DecThenCount();
 break;
case i_PAUSE_BEFORE:
 value = NextItemMax(255);
 *prog_out++ = (i_PAUSE_BEFORE << 8) + value;
 DecThenCount();
 break;
case i_PAUSE_AFTER:
 value = NextItemMax(255);
 *prog_out++ = (i_PAUSE_AFTER << 8) + value;
 DecThenCount();
 break;
case i_SET_LENGTH:
 value = NextItemMax(511);
 if (phoneme_out->type == phVOWEL)
  value = (value * vowel_length_factor)/100;

 if (after_if == false)
  phoneme_out->std_length = value/2;
 else {
  *prog_out++ = (i_SET_LENGTH << 8) + value/2;
  DecThenCount();
 }
 break;
case i_ADD_LENGTH:
 value = NextItem(tSIGNEDNUMBER) / 2;
 *prog_out++ = (i_ADD_LENGTH << 8) + (value & 0xff);
 DecThenCount();
 break;
case i_LENGTH_MOD:
```

```
    value = NextItem(tNUMBER);
    phoneme_out->length_mod = value;
    break;
   case i_IPA_NAME:
    NextItem(tSTRING);

    if (strcmp(item_string, "NULL") == 0)
     strcpy(item_string, " ");

    // copy the string, recognize characters in the form U+9999
    flags = 0;
    count = 0;
    ix = 1;

    for (p = item_string; *p != 0;) {
     p += utf8_in(&c, p);

     if ((c == '|') && (count > 0)) {
      // '|' means don't allow a tie or joiner before this letter
      flags |= (1 << (count -1));
     } else if ((c == 'U') && (p[0] == '+')) {
      int j;
      // U+9999
      p++;
      memcpy(number_buf, p, 4); // U+ should be followed by 4 hex
digits
      number_buf[4] = 0;
      c = '#';
      sscanf(number_buf, "%x", (unsigned int *)&c);

      // move past the 4 hexdecimal digits
      for (j = 0; j < 4; j++) {
       if (!isalnum(*p))
        break;
       p++;
      }
      ix += utf8_out(c, &ipa_buf[ix]);
```

```
      count++;
     } else {
      ix += utf8_out(c, &ipa_buf[ix]);
      count++;
     }
    }
    ipa_buf[0] = flags;
    ipa_buf[ix] = 0;

    start = 1;
    if (flags != 0)
     start = 0; // only include the flags byte if bits are set
    value = strlen(&ipa_buf[start]); // number of UTF-8 bytes

    *prog_out++ = (i_IPA_NAME << 8) + value;
    for (ix = 0; ix < value; ix += 2)
     *prog_out++ = (ipa_buf[ix+start] << 8) +
(ipa_buf[ix+start+1] & 0xff);
    DecThenCount();
    break;
   }
   break;
  case tSTATEMENT:
   switch (keyword)
   {
   case kIMPORT_PH:
    ImportPhoneme();
    phoneme_flags = phoneme_out->phflags;
    break;
   case kSTARTTYPE:
    phcode = NextItem(tPHONEMEMNEM);
    if (phcode == -1)
     phcode = LookupPhoneme(item_string, 1);
    phoneme_out->start_type = phcode;
    if (phoneme_out->type == phINVALID)
     error("a phoneme type or manner of articulation must be
specified before starttype");
```

```
    break;
   case kENDTYPE:
    phcode = NextItem(tPHONEMEMNEM);
    if (phcode == -1)
     phcode = LookupPhoneme(item_string, 1);
    if (phoneme_out->type == phINVALID)
     error("a phoneme type or manner of articulation must be
specified before endtype");
    else if (phoneme_out->type == phVOWEL)
     phoneme_out->end_type = phcode;
    else if (phcode != phoneme_out->start_type)
     error("endtype must equal starttype for consonants");
    break;
   case kVOICINGSWITCH:
    phcode = NextItem(tPHONEMEMNEM);
    if (phcode == -1)
     phcode = LookupPhoneme(item_string, 1);
    if (phoneme_out->type == phVOWEL)
     error("voicingswitch cannot be used on vowels");
    else
     phoneme_out->end_type = phcode; // use end_type field for
consonants as voicing_switch
    break;
   case kSTRESSTYPE:
    value = NextItem(tNUMBER);
    phoneme_out->std_length = value;
    if (prog_out > prog_buf) {
     error("stress phonemes can't contain program instructions");
     prog_out = prog_buf;
    }
    break;
   case kIF:
    endphoneme = CompileIf(0);
    break;
   case kELSE:
    endphoneme = CompileElse();
    break;
```

```
case kELIF:
 endphoneme = CompileElif();
 break;
case kENDIF:
 endphoneme = CompileEndif();
 break;
case kENDSWITCH:
 break;
case kSWITCH_PREVVOWEL:
 endphoneme = CompileSwitch(1);
 break;
case kSWITCH_NEXTVOWEL:
 endphoneme = CompileSwitch(2);
 break;
case kCALLPH:
 CallPhoneme();
 DecThenCount();
 break;
case kFMT:
 if_stack[if_level].returned = true;
 DecThenCount();
 if (phoneme_out->type == phVOWEL)
  CompileSound(keyword, 1);
 else
  CompileSound(keyword, 0);
 break;
case kWAV:
 if_stack[if_level].returned = true;
 // fallthrough:
case kVOWELSTART:
case kVOWELENDING:
case kANDWAV:
 DecThenCount();
 CompileSound(keyword, 0);
 break;
case kVOWELIN:
 DecThenCount();
```

```
    endphoneme = CompileVowelTransition(1);
    break;
   case kVOWELOUT:
    DecThenCount();
    endphoneme = CompileVowelTransition(2);
    break;
   case kTONESPEC:
    DecThenCount();
    CompileToneSpec();
    break;
   case kCONTINUE:
    *prog_out++ = INSTN_CONTINUE;
    DecThenCount();
    break;
   case kRETURN:
    *prog_out++ = INSTN_RETURN;
    DecThenCount();
    break;
   case kINCLUDE:
   case kPHONEMETABLE:
    error("Missing 'endphoneme' before '%s'", item_string);  //
drop through to endphoneme
    // fallthrough:
   case kENDPHONEME:
   case kENDPROCEDURE:
    endphoneme = 1;
    if (if_level > 0)
     error("Missing ENDIF");
    if ((prog_out > prog_buf) && (if_stack[0].returned == false))
     *prog_out++ = INSTN_RETURN;
    break;
   }
   break;
  }
 }

 if (endphoneme != 1)
```

```
  error("'endphoneme' not expected here");

 if (compile_phoneme) {
  if (phoneme_out->type == phINVALID) {
   error("Phoneme type is missing");
   phoneme_out->type = 0;
  }
  phoneme_out->phflags |= phoneme_flags;

  if (phoneme_out->phflags & phVOICED) {
   if (phoneme_out->type == phSTOP)
    phoneme_out->type = phVSTOP;
   else if (phoneme_out->type == phFRICATIVE)
    phoneme_out->type = phVFRICATIVE;
  }

  if (phoneme_out->std_length == 0) {
   if (phoneme_out->type == phVOWEL)
    phoneme_out->std_length = 180/2; // default length for vowel
  }

  phoneme_out->phflags |= phLOCAL; // declared in this phoneme
table

  if (phoneme_out->type == phDELETED)
   phoneme_out->mnemonic = 0x01; // will not be recognised
 }

 if (prog_out > prog_buf) {
  // write out the program for this phoneme
  fflush(f_phindex);
  phoneme_out->program = ftell(f_phindex) / sizeof(unsigned
short);

  if (f_prog_log != NULL) {
   phoneme_prog_log.addr = phoneme_out->program;
   phoneme_prog_log.length = prog_out - prog_buf;
```

```
   fwrite(&phoneme_prog_log, 1, sizeof(phoneme_prog_log),
f_prog_log);
  }

  if (compile_phoneme == 0)
   proc_addr[n_procs++] =  ftell(f_phindex) / sizeof(unsigned
short);
  fwrite(prog_buf, sizeof(unsigned short), prog_out - prog_buf,
f_phindex);
 }

 return 0;
}

static void WritePhonemeTables()
{
 int ix;
 int j;
 int n;
 int value;
 int count;
 PHONEME_TAB *p;

 value = n_phoneme_tabs;
 fputc(value, f_phtab);
 fputc(0, f_phtab);

 for (ix = 0; ix < n_phoneme_tabs; ix++) {
  p = phoneme_tab_list2[ix].phoneme_tab_ptr;
  n = n_phcodes_list[ix];
  memset(&p[n], 0, sizeof(p[n]));
  p[n].mnemonic = 0; // terminate the phoneme table

  // count number of locally declared phonemes
  count = 0;
  for (j = 0; j < n; j++) {
   if (ix == 0)
```

```
    p[j].phflags |= phLOCAL; // write all phonemes in the base
phoneme table

   if (p[j].phflags & phLOCAL)
    count++;
  }
  phoneme_tab_list2[ix].n_phonemes = count+1;

  fputc(count+1, f_phtab);
  fputc(phoneme_tab_list2[ix].includes, f_phtab);
  fputc(0, f_phtab);

  fwrite(phoneme_tab_list2[ix].name, 1, N_PHONEME_TAB_NAME,
f_phtab);

  for (j = 0; j < n; j++) {
   if (p[j].phflags & phLOCAL) {
    // this bit is set temporarily to incidate a local phoneme,
declared in
    // in the current phoneme file
    p[j].phflags &= ~phLOCAL;
    fwrite(&p[j], sizeof(PHONEME_TAB), 1, f_phtab);
   }
  }
  fwrite(&p[n], sizeof(PHONEME_TAB), 1, f_phtab); // include the
extra list-terminator phoneme entry
  free(p);
 }
}

static void EndPhonemeTable()
{
 int ix;

 if (n_phoneme_tabs == 0)
  return;
```

```
 // check that all referenced phonemes have been declared
 for (ix = 0; ix < n_phcodes; ix++) {
  if (phoneme_tab2[ix].type == phINVALID) {
   error("Phoneme [%s] not declared, referenced at line %d",
         WordToString(phoneme_tab2[ix].mnemonic),
(int)(phoneme_tab2[ix].program));
   error_count++;
   phoneme_tab2[ix].type = 0; // prevent the error message
repeating
  }
 }

 n_phcodes_list[n_phoneme_tabs-1] = n_phcodes;
}

static void StartPhonemeTable(const char *name)
{
 int ix;
 int j;
 PHONEME_TAB *p;

 if (n_phoneme_tabs >= N_PHONEME_TABS-1) {
  error("Too many phonemetables");
  return;
 }
 p = (PHONEME_TAB *)calloc(sizeof(PHONEME_TAB), N_PHONEME_TAB);

 if (p == NULL) {
  error("Out of memory");
  return;
 }

 memset(&phoneme_tab_list2[n_phoneme_tabs], 0,
sizeof(PHONEME_TAB_LIST));
 phoneme_tab_list2[n_phoneme_tabs].phoneme_tab_ptr = phoneme_tab2
= p;
 memset(phoneme_tab_list2[n_phoneme_tabs].name, 0,
```

```
sizeof(phoneme_tab_list2[n_phoneme_tabs].name));
 strncpy0(phoneme_tab_list2[n_phoneme_tabs].name, name,
N_PHONEME_TAB_NAME);
 n_phcodes = 1;
 phoneme_tab_list2[n_phoneme_tabs].includes = 0;

 if (n_phoneme_tabs > 0) {
  NextItem(tSTRING); // name of base phoneme table
  for (ix = 0; ix < n_phoneme_tabs; ix++) {
   if (strcmp(item_string, phoneme_tab_list2[ix].name) == 0) {
    phoneme_tab_list2[n_phoneme_tabs].includes = ix+1;

    // initialise the new phoneme table with the contents of this
one
    memcpy(phoneme_tab2, phoneme_tab_list2[ix].phoneme_tab_ptr,
sizeof(PHONEME_TAB)*N_PHONEME_TAB);
    n_phcodes = n_phcodes_list[ix];

    // clear "local phoneme" bit"
    for (j = 0; j < n_phcodes; j++)
     phoneme_tab2[j].phflags &= ~phLOCAL;
    break;
   }
  }
  if (ix == n_phoneme_tabs && strcmp(item_string, "_") != 0)
   error("Can't find base phonemetable '%s'", item_string);
 } else
  ReservePhCodes();

 n_phoneme_tabs++;
}

static void CompilePhonemeFiles()
{
 int item;
 FILE *f;
 char buf[sizeof(path_home)+120];
```

```
 linenum = 1;

 count_references = 0;
 duplicate_references = 0;
 count_frames = 0;
 n_procs = 0;

 for (;;) {
  if (feof(f_in)) {
   // end of file, go back to previous from, from which this was
included

   if (stack_ix == 0)
    break; // end of top level, finished
   fclose(f_in);
   f_in = stack[--stack_ix].file;
   strcpy(current_fname, stack[stack_ix].fname);
   linenum = stack[stack_ix].linenum;
  }

  item = NextItem(tKEYWORD);

  switch (item)
  {
  case kUTF8_BOM:
   break; // ignore bytes 0xef 0xbb 0xbf
  case kINCLUDE:
   NextItem(tSTRING);
   sprintf(buf, "%s/%s", phsrc, item_string);

   if ((stack_ix < N_STACK) && (f = fopen(buf, "rb")) != NULL) {
    stack[stack_ix].linenum = linenum;
    strcpy(stack[stack_ix].fname, current_fname);
    stack[stack_ix++].file = f_in;

    f_in = f;
```

```
    strncpy0(current_fname, item_string, sizeof(current_fname));
    linenum = 1;
   } else
    error("Missing file: %s", item_string);
   break;
  case kPHONEMETABLE:
   EndPhonemeTable();
   NextItem(tSTRING); // name of the new phoneme table
   StartPhonemeTable(item_string);
   break;
  case kPHONEMESTART:
   if (n_phoneme_tabs == 0) {
    error("phonemetable is missing");
    return;
   }
   CompilePhoneme(1);
   break;
  case kPROCEDURE:
   CompilePhoneme(0);
   break;
  default:
   if (!feof(f_in))
    error("Keyword 'phoneme' expected");
   break;
  }
 }
 memset(&phoneme_tab2[n_phcodes+1], 0,
sizeof(phoneme_tab2[n_phcodes+1]));
 phoneme_tab2[n_phcodes+1].mnemonic = 0; // terminator
}

#pragma GCC visibility push(default)

espeak_ng_STATUS
espeak_ng_CompilePhonemeData(long rate,
                             FILE *log,
                             espeak_ng_ERROR_CONTEXT *context)
```

```
{
 return espeak_ng_CompilePhonemeDataPath(rate, NULL, NULL, log,
context);
}

espeak_ng_STATUS
espeak_ng_CompilePhonemeDataPath(long rate,
                                 const char *source_path,
                                 const char *destination_path,
                                 FILE *log,
                                 espeak_ng_ERROR_CONTEXT
*context)
{
 if (!log) log = stderr;

 char fname[sizeof(path_home)+40];
 char phdst[sizeof(path_home)+40]; // Destination: path to the
phondata/phontab/phonindex output files.

 if (source_path) {
  sprintf(phsrc, "%s", source_path);
 } else {
  sprintf(phsrc, "%s/../phsource", path_home);
 }

 if (destination_path) {
  sprintf(phdst, "%s", destination_path);
 } else {
  sprintf(phdst, "%s", path_home);
 }

 samplerate_native = samplerate = rate;
 LoadPhData(NULL, NULL);
 if (LoadVoice("", 0) == NULL)
  return ENS_VOICE_NOT_FOUND;

 WavegenInit(rate, 0);
```

```
WavegenSetVoice(voice);

n_envelopes = 0;
error_count = 0;
resample_count = 0;
memset(markers_used, 0, sizeof(markers_used));

f_errors = log;

strncpy0(current_fname, "phonemes", sizeof(current_fname));

sprintf(fname, "%s/phonemes", phsrc);
fprintf(log, "Compiling phoneme data: %s\n", fname);
f_in = fopen(fname, "rb");
if (f_in == NULL)
 return create_file_error_context(context, errno, fname);

sprintf(fname, "%s/%s", phsrc, "compile_report");
f_report = fopen(fname, "w");
if (f_report == NULL) {
 int error = errno;
 fclose(f_in);
 return create_file_error_context(context, error, fname);
}

sprintf(fname, "%s/%s", phdst, "phondata-manifest");
if ((f_phcontents = fopen(fname, "w")) == NULL)
 f_phcontents = stderr;

fprintf(f_phcontents,
         "# This file lists the type of data that has been
compiled into the\n"
         "# phondata file\n"
         "#\n"
         "# The first character of a line indicates the type of
data:\n"
         "#   S - A SPECT_SEQ structure\n"
```

```
        "#   W - A wavefile segment\n"
        "#   E - An envelope\n"
        "#\n"
        "# Address is the displacement within phondata of this
item\n"
        "#\n"
        "#  Address  Data file\n"
        "#  -------  ---------\n");

 sprintf(fname, "%s/%s", phdst, "phondata");
 f_phdata = fopen(fname, "wb");
 if (f_phdata == NULL) {
  int error = errno;
  fclose(f_in);
  fclose(f_report);
  fclose(f_phcontents);
  return create_file_error_context(context, error, fname);
 }

 sprintf(fname, "%s/%s", phdst, "phonindex");
 f_phindex = fopen(fname, "wb");
 if (f_phindex == NULL) {
  int error = errno;
  fclose(f_in);
  fclose(f_report);
  fclose(f_phcontents);
  fclose(f_phdata);
  return create_file_error_context(context, error, fname);
 }

 sprintf(fname, "%s/%s", phdst, "phontab");
 f_phtab = fopen(fname, "wb");
 if (f_phtab == NULL) {
  int error = errno;
  fclose(f_in);
  fclose(f_report);
  fclose(f_phcontents);
```

```
  fclose(f_phdata);
  fclose(f_phindex);
  return create_file_error_context(context, error, fname);
 }

 sprintf(fname, "%s/compile_prog_log", phsrc);
 f_prog_log = fopen(fname, "wb");

 // write a word so that further data doesn't start at displ=0
 Write4Bytes(f_phdata, version_phdata);
 Write4Bytes(f_phdata, samplerate_native);
 Write4Bytes(f_phindex, version_phdata);

 memset(ref_hash_tab, 0, sizeof(ref_hash_tab));

 n_phoneme_tabs = 0;
 stack_ix = 0;
 StartPhonemeTable("base");
 CompilePhonemeFiles();

 EndPhonemeTable();
 WritePhonemeTables();

 fprintf(f_errors, "\nRefs %d,  Reused %d\n", count_references,
duplicate_references);

 fclose(f_in);
 fclose(f_phcontents);
 fclose(f_phdata);
 fclose(f_phindex);
 fclose(f_phtab);
 if (f_prog_log != NULL)
  fclose(f_prog_log);

 LoadPhData(NULL, NULL);

 CompileReport();
```

```
 fclose(f_report);

 if (resample_count > 0) {
  fprintf(f_errors, "\n%d WAV files resampled to %d Hz\n",
resample_count, samplerate_native);
  fprintf(log, "Compiled phonemes: %d errors, %d files resampled
to %d Hz.\n", error_count, resample_count, samplerate_native);
 } else
  fprintf(log, "Compiled phonemes: %d errors.\n", error_count);

 if (f_errors != stderr && f_errors != stdout)
  fclose(f_errors);

 espeak_ng_STATUS status = ReadPhondataManifest(context);
 if (status != ENS_OK)
  return status;

 return error_count > 0 ? ENS_COMPILE_ERROR : ENS_OK;
}

#pragma GCC visibility pop

static const char *preset_tune_names[] = {
 "s1", "c1", "q1", "e1", NULL
};

static const TUNE default_tune = {
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
 { 0, 0, 0, 0 },
 { 0, 40, 24, 8, 0, 0, 0, 0 },
 46, 57, PITCHfall, 16, 0, 0,
 255, 78, 50, 255,
 3, 5,
 { -7, -7, -7 }, { -7, -7, -7 },
 PITCHfall, 64, 8,
 PITCHfall, 70, 18, 24, 12,
```

```
 PITCHfall, 70, 18, 24, 12, 0,
 { 0, 0, 0, 0, 0, 0, 0, 0 }, 0
};

#define N_TUNE_NAMES  100

MNEM_TAB envelope_names[] = {
 { "fall", 0 },
 { "rise", 2 },
 { "fall-rise", 4 },
 { "fall-rise2", 6 },
 { "rise-fall",  8 },
 { "fall-rise3", 10 },
 { "fall-rise4", 12 },
 { "fall2", 14 },
 { "rise2", 16 },
 { "rise-fall-rise", 18 },
 { NULL, -1 }
};

static int LookupEnvelopeName(const char *name)
{
 return LookupMnem(envelope_names, name);
}

#pragma GCC visibility push(default)

espeak_ng_STATUS espeak_ng_CompileIntonation(FILE *log,
espeak_ng_ERROR_CONTEXT *context)
{
 if (!log) log = stderr;

 int ix;
 char *p;
 char c;
 int keyword;
 int n_tune_names = 0;
```

```
bool done_split = false;
bool done_onset = false;
bool done_last = false;
int n_preset_tunes = 0;
int found = 0;
int tune_number = 0;
FILE *f_out;
TUNE *tune_data;
TUNE new_tune;

char name[12];
char tune_names[N_TUNE_NAMES][12];
char buf[sizeof(path_home)+150];

error_count = 0;
f_errors = log;

sprintf(buf, "%s/../phsource/intonation.txt", path_home);
if ((f_in = fopen(buf, "r")) == NULL) {
 sprintf(buf, "%s/../phsource/intonation", path_home);
 if ((f_in = fopen(buf, "r")) == NULL) {
  int error = errno;
  fclose(f_errors);
  return create_file_error_context(context, error, buf);
 }
}

for (ix = 0; preset_tune_names[ix] != NULL; ix++)
 strcpy(tune_names[ix], preset_tune_names[ix]);
n_tune_names = ix;
n_preset_tunes = ix;

// make a list of the tune names
while (!feof(f_in)) {
 if (fgets(buf, sizeof(buf), f_in) == NULL)
  break;
```

```
  if ((memcmp(buf, "tune", 4) == 0) && isspace(buf[4])) {
   p = &buf[5];
   while (isspace(*p)) p++;

   ix = 0;
   while ((ix < (int)(sizeof(name) - 1)) && !isspace(*p))
    name[ix++] = *p++;
   name[ix] = 0;

   found = 0;
   for (ix = 0; ix < n_tune_names; ix++) {
    if (strcmp(name, tune_names[ix]) == 0) {
     found = 1;
     break;
    }
   }

   if (found == 0) {
    strncpy0(tune_names[n_tune_names++], name, sizeof(name));

    if (n_tune_names >= N_TUNE_NAMES)
     break;
   }
  }
 }
 rewind(f_in);
 linenum = 1;

 tune_data = (n_tune_names == 0) ? NULL : (TUNE
*)calloc(n_tune_names, sizeof(TUNE));
 if (tune_data == NULL) {
  fclose(f_in);
  fclose(f_errors);
  return ENOMEM;
 }

 sprintf(buf, "%s/intonations", path_home);
```

```
f_out = fopen(buf, "wb");
if (f_out == NULL) {
 int error = errno;
 fclose(f_in);
 fclose(f_errors);
 free(tune_data);
 return create_file_error_context(context, error, buf);
}

while (!feof(f_in)) {
 keyword = NextItem(tINTONATION);

 switch (keyword)
 {
 case kTUNE:
  done_split = false;

  memcpy(&new_tune, &default_tune, sizeof(TUNE));
  NextItem(tSTRING);
  strncpy0(new_tune.name, item_string, sizeof(new_tune.name));

  found = 0;
  tune_number = 0;
  for (ix = 0; ix < n_tune_names; ix++) {
   if (strcmp(new_tune.name, tune_names[ix]) == 0) {
    found = 1;
    tune_number = ix;

    if (tune_data[ix].name[0] != 0)
     found = 2;
    break;
   }
  }
  if (found == 2)
   error("Duplicate tune name: '%s'", new_tune.name);
  if (found == 0)
   error("Bad tune name: '%s;", new_tune.name);
```

```
 break;
case kENDTUNE:
 if (!found) continue;
 if (done_onset == false) {
  new_tune.unstr_start[0] = new_tune.unstr_start[1];
  new_tune.unstr_end[0] = new_tune.unstr_end[1];
 }
 if (done_last == false) {
  new_tune.unstr_start[2] = new_tune.unstr_start[1];
  new_tune.unstr_end[2] = new_tune.unstr_end[1];
 }
 memcpy(&tune_data[tune_number], &new_tune, sizeof(TUNE));
 break;
case kTUNE_PREHEAD:
 new_tune.prehead_start = NextItem(tNUMBER);
 new_tune.prehead_end = NextItem(tNUMBER);
 break;
case kTUNE_ONSET:
 new_tune.onset = NextItem(tNUMBER);
 new_tune.unstr_start[0] = NextItem(tSIGNEDNUMBER);
 new_tune.unstr_end[0] = NextItem(tSIGNEDNUMBER);
 done_onset = true;
 break;
case kTUNE_HEADLAST:
 new_tune.head_last = NextItem(tNUMBER);
 new_tune.unstr_start[2] = NextItem(tSIGNEDNUMBER);
 new_tune.unstr_end[2] = NextItem(tSIGNEDNUMBER);
 done_last = true;
 break;
case kTUNE_HEADENV:
 NextItem(tSTRING);
 if ((ix = LookupEnvelopeName(item_string)) < 0)
  error("Bad envelope name: '%s'", item_string);
 else
  new_tune.stressed_env = ix;
 new_tune.stressed_drop = NextItem(tNUMBER);
 break;
```

```
  case kTUNE_HEAD:
   new_tune.head_max_steps = NextItem(tNUMBER);
   new_tune.head_start = NextItem(tNUMBER);
   new_tune.head_end = NextItem(tNUMBER);
   new_tune.unstr_start[1] = NextItem(tSIGNEDNUMBER);
   new_tune.unstr_end[1] = NextItem(tSIGNEDNUMBER);
   break;
  case kTUNE_HEADEXTEND:
   // up to 8 numbers
   for (ix = 0; ix < (int)(sizeof(new_tune.head_extend)); ix++) {
    if (!isdigit(c = CheckNextChar()) && (c != '-'))
     break;

    new_tune.head_extend[ix] = (NextItem(tSIGNEDNUMBER) * 64) /
100; // convert from percentage to 64ths
   }
   new_tune.n_head_extend = ix; // number of values
   break;
  case kTUNE_NUCLEUS0:
   NextItem(tSTRING);
   if ((ix = LookupEnvelopeName(item_string)) < 0) {
    error("Bad envelope name: '%s'", item_string);
    break;
   }
   new_tune.nucleus0_env = ix;
   new_tune.nucleus0_max = NextItem(tNUMBER);
   new_tune.nucleus0_min = NextItem(tNUMBER);
   break;
  case kTUNE_NUCLEUS1:
   NextItem(tSTRING);
   if ((ix = LookupEnvelopeName(item_string)) < 0) {
    error("Bad envelope name: '%s'", item_string);
    break;
   }
   new_tune.nucleus1_env = ix;
   new_tune.nucleus1_max = NextItem(tNUMBER);
   new_tune.nucleus1_min = NextItem(tNUMBER);
```

```
 new_tune.tail_start = NextItem(tNUMBER);
 new_tune.tail_end = NextItem(tNUMBER);

 if (!done_split) {
  // also this as the default setting for 'split'
  new_tune.split_nucleus_env = ix;
  new_tune.split_nucleus_max = new_tune.nucleus1_max;
  new_tune.split_nucleus_min = new_tune.nucleus1_min;
  new_tune.split_tail_start = new_tune.tail_start;
  new_tune.split_tail_end = new_tune.tail_end;
 }
 break;
case kTUNE_SPLIT:
 NextItem(tSTRING);
 if ((ix = LookupEnvelopeName(item_string)) < 0) {
  error("Bad envelope name: '%s'", item_string);
  break;
 }
 done_split = true;
 new_tune.split_nucleus_env = ix;
 new_tune.split_nucleus_max = NextItem(tNUMBER);
 new_tune.split_nucleus_min = NextItem(tNUMBER);
 new_tune.split_tail_start = NextItem(tNUMBER);
 new_tune.split_tail_end = NextItem(tNUMBER);
 NextItem(tSTRING);
 item_string[12] = 0;
 for (ix = 0; ix < n_tune_names; ix++) {
  if (strcmp(item_string, tune_names[ix]) == 0)
   break;
 }

 if (ix == n_tune_names)
  error("Tune '%s' not found", item_string);
 else
  new_tune.split_tune = ix;
 break;
default:
```

```
   error("Unexpected: '%s'", item_string);
   break;
  }
 }

 for (ix = 0; ix < n_preset_tunes; ix++) {
  if (tune_data[ix].name[0] == 0)
   error("Tune '%s' not defined", preset_tune_names[ix]);
 }
 fwrite(tune_data, n_tune_names, sizeof(TUNE), f_out);
 free(tune_data);
 fclose(f_in);
 fclose(f_out);

 fprintf(log, "Compiled %d intonation tunes: %d errors.\n",
n_tune_names, error_count);

 LoadPhData(NULL, NULL);

 return error_count > 0 ? ENS_COMPILE_ERROR : ENS_OK;
}

#pragma GCC visibility pop
```

Chapter 40

./src/libespeak-ng/synthdata.c

```
#include "config.h"

#include <ctype.h>
#include <errno.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "readclause.h"
#include "synthdata.h"

#include "error.h"
#include "speech.h"
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
```

```
#include "translate.h"

const int version_phdata  = 0x014801;

// copy the current phoneme table into here
int n_phoneme_tab;
int current_phoneme_table;
PHONEME_TAB *phoneme_tab[N_PHONEME_TAB];
unsigned char phoneme_tab_flags[N_PHONEME_TAB];   // bit 0: not
inherited

unsigned short *phoneme_index = NULL;
char *phondata_ptr = NULL;
unsigned char *wavefile_data = NULL;
static unsigned char *phoneme_tab_data = NULL;

int n_phoneme_tables;
PHONEME_TAB_LIST phoneme_tab_list[N_PHONEME_TABS];
int phoneme_tab_number = 0;

int wavefile_ix; // a wavefile to play along with the synthesis
int wavefile_amp;

int seq_len_adjust;
int vowel_transition[4];

static espeak_ng_STATUS ReadPhFile(void **ptr, const char *fname,
int *size, espeak_ng_ERROR_CONTEXT *context)
{
 if (!ptr) return EINVAL;

 FILE *f_in;
 int length;
 char buf[sizeof(path_home)+40];

 sprintf(buf, "%s%c%s", path_home, PATHSEP, fname);
 length = GetFileLength(buf);
```

```
 if (length < 0) // length == -errno
  return create_file_error_context(context, -length, buf);

 if ((f_in = fopen(buf, "rb")) == NULL)
  return create_file_error_context(context, errno, buf);

 if (*ptr != NULL)
  free(*ptr);

 if ((*ptr = malloc(length)) == NULL) {
  fclose(f_in);
  return ENOMEM;
 }
 if (fread(*ptr, 1, length, f_in) != length) {
  int error = errno;
  fclose(f_in);
  free(*ptr);
  return create_file_error_context(context, error, buf);
 }

 fclose(f_in);
 if (size != NULL)
  *size = length;
 return ENS_OK;
}

espeak_ng_STATUS LoadPhData(int *srate, espeak_ng_ERROR_CONTEXT
*context)
{
 int ix;
 int n_phonemes;
 int version;
 int length = 0;
 int rate;
 unsigned char *p;

 espeak_ng_STATUS status;
```

```
 if ((status = ReadPhFile((void **)&phoneme_tab_data, "phontab",
NULL, context)) != ENS_OK)
  return status;
 if ((status = ReadPhFile((void **)&phoneme_index, "phonindex",
NULL, context)) != ENS_OK)
  return status;
 if ((status = ReadPhFile((void **)&phondata_ptr, "phondata",
NULL, context)) != ENS_OK)
  return status;
 if ((status = ReadPhFile((void **)&tunes, "intonations",
&length, context)) != ENS_OK)
  return status;
 wavefile_data = (unsigned char *)phondata_ptr;
 n_tunes = length / sizeof(TUNE);

 // read the version number and sample rate from the first 8
bytes of phondata
 version = 0; // bytes 0-3, version number
 rate = 0;    // bytes 4-7, sample rate
 for (ix = 0; ix < 4; ix++) {
  version += (wavefile_data[ix] << (ix*8));
  rate += (wavefile_data[ix+4] << (ix*8));
 }

 if (version != version_phdata)
  return create_version_mismatch_error_context(context,
path_home, version, version_phdata);

 // set up phoneme tables
 p = phoneme_tab_data;
 n_phoneme_tables = p[0];
 p += 4;

 for (ix = 0; ix < n_phoneme_tables; ix++) {
  n_phonemes = p[0];
  phoneme_tab_list[ix].n_phonemes = p[0];
  phoneme_tab_list[ix].includes = p[1];
```

```
  p += 4;
  memcpy(phoneme_tab_list[ix].name, p, N_PHONEME_TAB_NAME);
  p += N_PHONEME_TAB_NAME;
  phoneme_tab_list[ix].phoneme_tab_ptr = (PHONEME_TAB *)p;
  p += (n_phonemes * sizeof(PHONEME_TAB));
 }

 if (phoneme_tab_number >= n_phoneme_tables)
  phoneme_tab_number = 0;

 if (srate != NULL)
  *srate = rate;
 return ENS_OK;
}

void FreePhData(void)
{
 free(phoneme_tab_data);
 free(phoneme_index);
 free(phondata_ptr);
 free(tunes);
 phoneme_tab_data = NULL;
 phoneme_index = NULL;
 phondata_ptr = NULL;
 tunes = NULL;
}

int PhonemeCode(unsigned int mnem)
{
 int ix;

 for (ix = 0; ix < n_phoneme_tab; ix++) {
  if (phoneme_tab[ix] == NULL)
   continue;
  if (phoneme_tab[ix]->mnemonic == mnem)
   return phoneme_tab[ix]->code;
 }
```

```
 return 0;
}

int LookupPhonemeString(const char *string)
{
 int ix;
 unsigned char c;
 unsigned int mnem;

 // Pack up to 4 characters into a word
 mnem = 0;
 for (ix = 0; ix < 4; ix++) {
  if (string[ix] == 0) break;
  c = string[ix];
  mnem |= (c << (ix*8));
 }

 return PhonemeCode(mnem);
}

frameref_t *LookupSpect(PHONEME_TAB *this_ph, int which,
FMT_PARAMS *fmt_params,  int *n_frames, PHONEME_LIST *plist)
{
 int ix;
 int nf;
 int nf1;
 int seq_break;
 frameref_t *frames;
 int length1;
 int length_std;
 int length_factor;
 SPECT_SEQ *seq, *seq2;
 SPECT_SEQK *seqk, *seqk2;
 frame_t *frame;
 static frameref_t frames_buf[N_SEQ_FRAMES];

 seq = (SPECT_SEQ *)(&phondata_ptr[fmt_params->fmt_addr]);
```

```
 seqk = (SPECT_SEQK *)seq;
 nf = seq->n_frames;

 if (nf >= N_SEQ_FRAMES)
  nf = N_SEQ_FRAMES - 1;

 seq_len_adjust = fmt_params->fmt2_lenadj +
fmt_params->fmt_length;
 seq_break = 0;

 for (ix = 0; ix < nf; ix++) {
  if (seq->frame[0].frflags & FRFLAG_KLATT)
   frame = &seqk->frame[ix];
  else
   frame = (frame_t *)&seq->frame[ix];
  frames_buf[ix].frame = frame;
  frames_buf[ix].frflags = frame->frflags;
  frames_buf[ix].length = frame->length;
  if (frame->frflags & FRFLAG_VOWEL_CENTRE)
   seq_break = ix;
 }

 frames = &frames_buf[0];
 if (seq_break > 0) {
  if (which == 1)
   nf = seq_break + 1;
  else {
   frames = &frames_buf[seq_break]; // body of vowel, skip past
initial frames
   nf -= seq_break;
  }
 }

 // do we need to modify a frame for blending with a consonant?
 if ((this_ph->type == phVOWEL) && (fmt_params->fmt2_addr == 0)
&& (fmt_params->use_vowelin))
  seq_len_adjust += FormantTransition2(frames, &nf,
```

```
fmt_params->transition0, fmt_params->transition1, NULL, which);

 length1 = 0;
 nf1 = nf - 1;
 for (ix = 0; ix < nf1; ix++)
  length1 += frames[ix].length;

 if (fmt_params->fmt2_addr != 0) {
  // a secondary reference has been returned, which is not a
wavefile
  // add these spectra to the main sequence
  seq2 = (SPECT_SEQ *)(&phondata_ptr[fmt_params->fmt2_addr]);
  seqk2 = (SPECT_SEQK *)seq2;

  // first frame of the addition just sets the length of the last
frame of the main seq
  nf--;
  for (ix = 0; ix < seq2->n_frames; ix++) {
   if (seq2->frame[0].frflags & FRFLAG_KLATT)
    frame = &seqk2->frame[ix];
   else
    frame = (frame_t *)&seq2->frame[ix];

   frames[nf].length = frame->length;
   if (ix > 0) {
    frames[nf].frame = frame;
    frames[nf].frflags = frame->frflags;
   }
   nf++;
  }
  wavefile_ix = 0;
 }

 if (length1 > 0) {
  if (which == 2) {
   // adjust the length of the main part to match the standard
length specified for the vowel
```

```
   // less the front part of the vowel and any added suffix

   length_std = fmt_params->std_length + seq_len_adjust - 45;
   if (length_std < 10)
    length_std = 10;
   if (plist->synthflags & SFLAG_LENGTHEN)
    length_std += (phoneme_tab[phonLENGTHEN]->std_length * 2); //
phoneme was followed by an extra : symbol

   // can adjust vowel length for stressed syllables here

   length_factor = (length_std * 256)/ length1;

   for (ix = 0; ix < nf1; ix++)
    frames[ix].length = (frames[ix].length * length_factor)/256;
  } else {
   if (which == 1) {
    // front of a vowel
    if (fmt_params->fmt_control == 1) {
     // This is the default start of a vowel.
     // Allow very short vowels to have shorter front parts
     if (fmt_params->std_length < 130)
      frames[0].length = (frames[0].length *
fmt_params->std_length)/130;
    }
   } else {
    // not a vowel
    if (fmt_params->std_length > 0)
     seq_len_adjust += (fmt_params->std_length - length1);
   }

   if (seq_len_adjust != 0) {
    length_factor = ((length1 + seq_len_adjust) * 256)/length1;
    for (ix = 0; ix < nf1; ix++)
     frames[ix].length = (frames[ix].length * length_factor)/256;
   }
  }
```

```
 }

 return frames;
}

unsigned char *GetEnvelope(int index)
{
 if (index == 0) {
  fprintf(stderr, "espeak: No envelope\n");
  return envelope_data[0]; // not found, use a default envelope
 }
 return (unsigned char *)&phondata_ptr[index];
}

static void SetUpPhonemeTable(int number, bool recursing)
{
 int ix;
 int includes;
 int ph_code;
 PHONEME_TAB *phtab;

 if (recursing == false)
  memset(phoneme_tab_flags, 0, sizeof(phoneme_tab_flags));

 if ((includes = phoneme_tab_list[number].includes) > 0) {
  // recursively include base phoneme tables
  SetUpPhonemeTable(includes-1, true);
 }

 // now add the phonemes from this table
 phtab = phoneme_tab_list[number].phoneme_tab_ptr;
 for (ix = 0; ix < phoneme_tab_list[number].n_phonemes; ix++) {
  ph_code = phtab[ix].code;
  phoneme_tab[ph_code] = &phtab[ix];
  if (ph_code > n_phoneme_tab)
   n_phoneme_tab = ph_code;
```

```
  if (recursing == 0)
   phoneme_tab_flags[ph_code] |= 1; // not inherited
 }
}

void SelectPhonemeTable(int number)
{
 n_phoneme_tab = 0;
 SetUpPhonemeTable(number, false); // recursively for included
phoneme tables
 n_phoneme_tab++;
 current_phoneme_table = number;
}

int LookupPhonemeTable(const char *name)
{
 int ix;

 for (ix = 0; ix < n_phoneme_tables; ix++) {
  if (strcmp(name, phoneme_tab_list[ix].name) == 0) {
   phoneme_tab_number = ix;
   break;
  }
 }
 if (ix == n_phoneme_tables)
  return -1;

 return ix;
}

int SelectPhonemeTableName(const char *name)
{
 // Look up a phoneme set by name, and select it if it exists
 // Returns the phoneme table number
 int ix;

 if ((ix = LookupPhonemeTable(name)) == -1)
```

```
  return -1;

 SelectPhonemeTable(ix);
 return ix;
}

void LoadConfig(void)
{
 // Load configuration file, if one exists
 char buf[sizeof(path_home)+10];
 FILE *f;
 int ix;
 char c1;
 char string[200];

 for (ix = 0; ix < N_SOUNDICON_SLOTS; ix++) {
  soundicon_tab[ix].filename = NULL;
  soundicon_tab[ix].data = NULL;
 }

 sprintf(buf, "%s%c%s", path_home, PATHSEP, "config");
 if ((f = fopen(buf, "r")) == NULL)
  return;

 while (fgets(buf, sizeof(buf), f) != NULL) {
  if (buf[0] == '/')  continue;

  if (memcmp(buf, "tone", 4) == 0)
   ReadTonePoints(&buf[5], tone_points);
  else if (memcmp(buf, "soundicon", 9) == 0) {
   ix = sscanf(&buf[10], "_%c %s", &c1, string);
   if (ix == 2) {
    soundicon_tab[n_soundicon_tab].name = c1;
    soundicon_tab[n_soundicon_tab].filename = strdup(string);
    soundicon_tab[n_soundicon_tab++].length = 0;
   }
  }
```

```
 }
 fclose(f);
}

PHONEME_DATA this_ph_data;

static void InvalidInstn(PHONEME_TAB *ph, int instn)
{
 fprintf(stderr, "Invalid instruction %.4x for phoneme '%s'\n",
instn, WordToString(ph->mnemonic));
}

static bool StressCondition(Translator *tr, PHONEME_LIST *plist,
int condition, int control)
{
 int stress_level;
 PHONEME_LIST *pl;
 static int condition_level[4] = { 1, 2, 4, 15 };

 if (phoneme_tab[plist[0].phcode]->type == phVOWEL)
  pl = plist;
 else {
  // consonant, get stress from the following vowel
  if (phoneme_tab[plist[1].phcode]->type == phVOWEL)
   pl = &plist[1];
  else
   return false; // no stress elevel for this consonant
 }

 stress_level = pl->stresslevel & 0xf;

 if (tr != NULL) {
  if ((control & 1) && (plist->synthflags & SFLAG_DICTIONARY) &&
((tr->langopts.param[LOPT_REDUCE] & 1) == 0)) {
   // change phoneme.  Don't change phonemes which are given for
the word in the dictionary.
   return false;
```

```
  }

  if ((tr->langopts.param[LOPT_REDUCE] & 0x2) && (stress_level >=
pl->wordstress)) {
   // treat the most stressed syllable in an unstressed word as
stressed
   stress_level = STRESS_IS_PRIMARY;
  }
 }

 if (condition == STRESS_IS_PRIMARY)
  return stress_level >= pl->wordstress;

 if (condition == STRESS_IS_SECONDARY) {
  if (stress_level > STRESS_IS_SECONDARY)
   return true;
 } else {
  if (stress_level < condition_level[condition])
   return true;
 }
 return false;

}

static int CountVowelPosition(PHONEME_LIST *plist)
{
 int count = 0;

 for (;;) {
  if (plist->ph->type == phVOWEL)
   count++;
  if (plist->sourceix != 0)
   break;
  plist--;
 }
 return count;
}
```

```
static bool InterpretCondition(Translator *tr, int control,
PHONEME_LIST *plist, unsigned short *p_prog, WORD_PH_DATA
*worddata)
{
 int which;
 int ix;
 unsigned int data;
 int instn;
 int instn2;
 bool check_endtype = false;
 PHONEME_TAB *ph;
 PHONEME_LIST *plist_this;

 // instruction: 2xxx, 3xxx

 // bits 8-10 = 0 to 5,  which phoneme, =6 the 'which'
information is in the next instruction.
 // bit 11 = 0, bits 0-7 are a phoneme code
 // bit 11 = 1, bits 5-7 type of data, bits 0-4 data value

 // bits 8-10 = 7,  other conditions

 instn = (*p_prog) & 0xfff;
 data = instn & 0xff;
 instn2 = instn >> 8;

 if (instn2 < 14) {
  plist_this = plist;
  which = (instn2) % 7;

  if (which == 6) {
   // the 'which' code is in the next instruction
   p_prog++;
   which = (*p_prog);
  }
```

```
if (which == 4) {
 // nextPhW not word boundary
 if (plist[1].sourceix)
  return false;
}
if (which == 5) {
 // prevPhW, not word boundary
 if (plist[0].sourceix)
  return false;
}
if (which == 6) {
 // next2PhW, not word boundary
 if (plist[1].sourceix || plist[2].sourceix)
  return false;
}

switch (which)
{
case 0: // prevPh
case 5: // prevPhW
 plist--;
 check_endtype = true;
 break;
case 1: // thisPh
 break;
case 2: // nextPh
case 4: // nextPhW
 plist++;
 break;
case 3: // next2Ph
case 6: // next2PhW
 plist += 2;
 break;
case 7:
 // nextVowel, not word boundary
 for (which = 1;; which++) {
  if (plist[which].sourceix)
```

```
     return false;
    if (phoneme_tab[plist[which].phcode]->type == phVOWEL) {
     plist = &plist[which];
     break;
    }
   }
   break;
  case 8: // prevVowel in this word
   if ((worddata == NULL) || (worddata->prev_vowel.ph == NULL))
    return false; // no previous vowel
   plist = &(worddata->prev_vowel);
   check_endtype = true;
   break;
  case 9: // next3PhW
   for (ix = 1; ix <= 3; ix++) {
    if (plist[ix].sourceix)
     return false;
   }
   plist = &plist[3];
   break;
  case 10: // prev2PhW
   if ((plist[0].sourceix) || (plist[-1].sourceix))
    return false;
   plist -= 2;
   check_endtype = true;
   break;
  }

  if ((which == 0) || (which == 5)) {
   if (plist->phcode == 1) {
    // This is a NULL phoneme, a phoneme has been deleted so look
at the previous phoneme
    plist--;
   }
  }

  if (control & 0x100) {
```

```
   // "change phonemes" pass
   plist->ph = phoneme_tab[plist->phcode];
  }
  ph = plist->ph;

  if (instn2 < 7) {
   // 'data' is a phoneme number
   if ((phoneme_tab[data]->mnemonic == ph->mnemonic) == true)
    return true;

   //  not an exact match, check for a vowel type (eg. #i )
   if ((check_endtype) && (ph->type == phVOWEL))
    return data == ph->end_type; // prevPh() match on end_type
   return data == ph->start_type; // thisPh() or nextPh(), match
on start_type
  }

  data = instn & 0x1f;

  switch (instn & 0xe0)
  {
  case CONDITION_IS_PHONEME_TYPE:
   return ph->type == data;
  case CONDITION_IS_PLACE_OF_ARTICULATION:
   return ((ph->phflags >> 16) & 0xf) == data;
  case CONDITION_IS_PHFLAG_SET:
   return (ph->phflags & (1 << data)) != 0;
  case CONDITION_IS_OTHER:
   switch (data)
   {
   case STRESS_IS_DIMINISHED:
   case STRESS_IS_UNSTRESSED:
   case STRESS_IS_NOT_STRESSED:
   case STRESS_IS_SECONDARY:
   case STRESS_IS_PRIMARY:
    return StressCondition(tr, plist, data, 0);
   case isBreak:
```

```
    return (ph->type == phPAUSE) || (plist_this->synthflags &
SFLAG_NEXT_PAUSE);
   case isWordStart:
    return plist->sourceix != 0;
   case isWordEnd:
    return plist[1].sourceix || (plist[1].ph->type == phPAUSE);
   case isAfterStress:
    if (plist->sourceix != 0)
     return false;
    do {
     plist--;
     if ((plist->stresslevel & 0xf) >= 4)
      return true;

    } while (plist->sourceix == 0);
    break;
   case isNotVowel:
    return ph->type != phVOWEL;
   case isFinalVowel:
    for (;;) {
     plist++;
     if (plist->sourceix != 0)
      return true; // start of next word, without finding another
vowel
     if (plist->ph->type == phVOWEL)
      return false;
    }
   case isVoiced:
    return (ph->type == phVOWEL) || (ph->type == phLIQUID) ||
(ph->phflags & phVOICED);
   case isFirstVowel:
    return CountVowelPosition(plist) == 1;
   case isSecondVowel:
    return CountVowelPosition(plist) == 2;
   case isTranslationGiven:
    return (plist->synthflags & SFLAG_DICTIONARY) != 0;
   }
```

```
   break;

  }
  return false;
 } else if (instn2 == 0xf) {
  // Other conditions
  switch (data)
  {
  case 1: // PreVoicing
   return control & 1;
  case 2: // KlattSynth
   return voice->klattv[0] != 0;
  case 3: // MbrolaSynth
   return mbrola_name[0] != 0;
  }
 }
 return false;
}

static void SwitchOnVowelType(PHONEME_LIST *plist, PHONEME_DATA
*phdata, unsigned short **p_prog, int instn_type)
{
 unsigned short *prog;
 int voweltype;
 signed char x;

 if (instn_type == 2) {
  phdata->pd_control |= pd_FORNEXTPH;
  voweltype = plist[1].ph->start_type; // SwitchNextVowelType
 } else
  voweltype = plist[-1].ph->end_type; // SwitchPrevVowelType

 voweltype -= phonVOWELTYPES;
 if ((voweltype >= 0) && (voweltype < 6)) {
  prog = *p_prog + voweltype*2;
  phdata->sound_addr[instn_type] = (((prog[1] & 0xf) << 16) +
prog[2]) * 4;
```

```
  x = (prog[1] >> 4) & 0xff;
  phdata->sound_param[instn_type] = x; // sign extend
 }

}

int NumInstnWords(unsigned short *prog)
{
 int instn;
 int instn2;
 int instn_type;
 int n;
 int type2;
 static const char n_words[16] = { 0, 1, 0, 0, 1, 1, 0, 1, 1, 2,
4, 0, 0, 0, 0, 0 };

 instn = *prog;
 instn_type = instn >> 12;
 if ((n = n_words[instn_type]) > 0)
  return n;

 switch (instn_type)
 {
 case 0:
  if (((instn & 0xf00) >> 8) == i_IPA_NAME) {
   n = ((instn & 0xff) + 1) / 2;
   return n+1;
  }
  return 1;
 case 6:
  type2 = (instn & 0xf00) >> 9;
  if ((type2 == 5) || (type2 == 6))
   return 12; // switch on vowel type
  return 1;
 case 2:
 case 3:
  // a condition, check for a 2-word instruction
```

```
  if (((n = instn & 0x0f00) == 0x600) || (n == 0x0d00))
   return 2;
  return 1;
 default:
  // instn_type 11 to 15, 2 words
  instn2 = prog[2];
  if ((instn2 >> 12) == 0xf) {
   // This instruction is followed by addWav(), 2 more words
   return 4;
  }
  if (instn2 == INSTN_CONTINUE)
   return 3;
  return 2;
 }
}

void InterpretPhoneme(Translator *tr, int control, PHONEME_LIST
*plist, PHONEME_DATA *phdata, WORD_PH_DATA *worddata)
{
 // control:
 // bit 0:  PreVoicing
 // bit 8:  change phonemes

 PHONEME_TAB *ph;
 unsigned short *prog;
 unsigned short instn;
 int instn2;
 int or_flag;
 bool truth;
 bool truth2;
 int data;
 int end_flag;
 int ix;
 signed char param_sc;

 #define N_RETURN 10
 int n_return = 0;
```

```
 unsigned short *return_addr[N_RETURN]; // return address stack

 ph = plist->ph;

 if ((worddata != NULL) && (plist->sourceix)) {
  // start of a word, reset word data
  worddata->prev_vowel.ph = NULL;
 }

 memset(phdata, 0, sizeof(PHONEME_DATA));
 phdata->pd_param[i_SET_LENGTH] = ph->std_length;
 phdata->pd_param[i_LENGTH_MOD] = ph->length_mod;

 if (ph->program == 0)
  return;

 end_flag = 0;

 for (prog = &phoneme_index[ph->program]; end_flag != 1; prog++)
{
  instn = *prog;
  instn2 = (instn >> 8) & 0xf;

  switch (instn >> 12)
  {
  case 0: // 0xxx
   data = instn & 0xff;

   if (instn2 == 0) {
    // instructions with no operand
    switch (data)
    {
    case INSTN_RETURN:
     end_flag = 1;
     break;
    case INSTN_CONTINUE:
     break;
```

```
  default:
   InvalidInstn(ph, instn);
   break;
  }
 } else if (instn2 == i_APPEND_IFNEXTVOWEL) {
  if (phoneme_tab[plist[1].phcode]->type == phVOWEL)
   phdata->pd_param[i_APPEND_PHONEME] = data;
 } else if (instn2 == i_ADD_LENGTH) {
  if (data & 0x80) {
   // a negative value, do sign extension
   data = -(0x100 - data);
  }
  phdata->pd_param[i_SET_LENGTH] += data;
 } else if (instn2 == i_IPA_NAME) {
  // followed by utf-8 characters, 2 per instn word
  for (ix = 0; (ix < data) && (ix < 16); ix += 2) {
   prog++;
   phdata->ipa_string[ix] = prog[0] >> 8;
   phdata->ipa_string[ix+1] = prog[0] & 0xff;
  }
  phdata->ipa_string[ix] = 0;
 } else if (instn2 < N_PHONEME_DATA_PARAM) {
  phdata->pd_param[instn2] = data;
  if ((instn2 == i_CHANGE_PHONEME) && (control & 0x100)) {
   // found ChangePhoneme() in PhonemeList mode, exit
   end_flag = 1;
  }
 } else
  InvalidInstn(ph, instn);
 break;
case 1:
 if (tr == NULL)
  break; // ignore if in synthesis stage

 if (instn2 < 8) {
  // ChangeIf
  if (StressCondition(tr, plist, instn2 & 7, 1) == true) {
```

```
      phdata->pd_param[i_CHANGE_PHONEME] = instn & 0xff;
      end_flag = 1; // change phoneme, exit
     }
    }
    break;
   case 2:
   case 3:
    // conditions
    or_flag = 0;
    truth = true;
    while ((instn & 0xe000) == 0x2000) {
     // process a sequence of conditions, using  boolean
accumulator
     truth2 = InterpretCondition(tr, control, plist, prog,
worddata);
     prog += NumInstnWords(prog);
     if (*prog == i_NOT) {
      truth2 = truth2 ^ 1;
      prog++;
     }

     if (or_flag)
      truth = truth || truth2;
     else
      truth = truth && truth2;
     or_flag = instn & 0x1000;
     instn = *prog;
    }

    if (truth == false) {
     if ((instn & 0xf800) == i_JUMP_FALSE)
      prog += instn & 0xff;
     else {
      // instruction after a condition is not JUMP_FALSE, so skip
the instruction.
      prog += NumInstnWords(prog);
      if ((prog[0] & 0xfe00) == 0x6000)
```

```
      prog++; // and skip ELSE jump
     }
    }
    prog--;
    break;
   case 6:
    // JUMP
    switch (instn2 >> 1)
    {
    case 0:
     prog += (instn & 0xff) - 1;
     break;
    case 4:
     // conditional jumps should have been processed in the
Condition section
     break;
    case 5: // NexttVowelStarts
     SwitchOnVowelType(plist, phdata, &prog, 2);
     break;
    case 6: // PrevVowelTypeEndings
     SwitchOnVowelType(plist, phdata, &prog, 3);
     break;
    }
    break;
   case 9:
    data = ((instn & 0xf) << 16) + prog[1];
    prog++;
    switch (instn2)
    {
    case 1:
     // call a procedure or another phoneme
     if (n_return < N_RETURN) {
      return_addr[n_return++] = prog;
      prog = &phoneme_index[data] - 1;
     }
     break;
    case 2:
```

```
    // pitch envelope
    phdata->pitch_env = data;
    break;
   case 3:
    // amplitude envelope
    phdata->amp_env = data;
    break;
   }
   break;
  case 10: //  Vowelin, Vowelout
   if (instn2 == 1)
    ix = 0;
   else
    ix = 2;

   phdata->vowel_transition[ix] = ((prog[0] & 0xff) << 16) +
prog[1];
   phdata->vowel_transition[ix+1] = (prog[2] << 16) + prog[3];
   prog += 3;
   break;
  case 11: // FMT
  case 12: // WAV
  case 13: // VowelStart
  case 14: // VowelEnd
  case 15: // addWav
   instn2 = (instn >> 12) - 11;
   phdata->sound_addr[instn2] = ((instn & 0xf) << 18) + (prog[1]
<< 2);
   param_sc = phdata->sound_param[instn2] = (instn >> 4) & 0xff;
   prog++;

   if (prog[1] != INSTN_CONTINUE) {
    if (instn2 < 2) {
     // FMT() and WAV() imply Return
     end_flag = 1;
     if ((prog[1] >> 12) == 0xf) {
      // Return after the following addWav()
```

```
      end_flag = 2;
     }
    } else if (instn2 == pd_ADDWAV) {
     // addWav(), return if previous instruction was FMT() or
WAV()
     end_flag--;
    }

    if ((instn2 == pd_VWLSTART) || (instn2 == pd_VWLEND)) {
     // VowelStart or VowelEnding.
     phdata->sound_param[instn2] = param_sc;   // sign extend
    }
   }
   break;
  default:
   InvalidInstn(ph, instn);
   break;
  }

  if ((end_flag == 1) && (n_return > 0)) {
   // return from called procedure or phoneme
   end_flag = 0;
   prog = return_addr[--n_return];
  }
 }

 if ((worddata != NULL) && (plist->type == phVOWEL))
  memcpy(&worddata->prev_vowel, &plist[0], sizeof(PHONEME_LIST));

 plist->std_length = phdata->pd_param[i_SET_LENGTH];
 if (phdata->sound_addr[0] != 0) {
  plist->phontab_addr = phdata->sound_addr[0]; // FMT address
  plist->sound_param = phdata->sound_param[0];
 } else {
  plist->phontab_addr = phdata->sound_addr[1]; // WAV address
  plist->sound_param = phdata->sound_param[1];
 }
```

```
}

void InterpretPhoneme2(int phcode, PHONEME_DATA *phdata)
{
 // Examine the program of a single isolated phoneme
 int ix;
 PHONEME_LIST plist[4];
 memset(plist, 0, sizeof(plist));

 for (ix = 0; ix < 4; ix++) {
  plist[ix].phcode = phonPAUSE;
  plist[ix].ph = phoneme_tab[phonPAUSE];
 }

 plist[1].phcode = phcode;
 plist[1].ph = phoneme_tab[phcode];
 plist[2].sourceix = 1;

 InterpretPhoneme(NULL, 0, &plist[1], phdata, NULL);
}
```

Chapter 41

./src/libespeak-ng/speech.c

```
#include "config.h"

#include <assert.h>
#include <ctype.h>
#include <errno.h>
#include <locale.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#include <wchar.h>

#ifdef HAVE_PCAUDIOLIB_AUDIO_H
#include <pcaudiolib/audio.h>
#endif

#if defined(_WIN32) || defined(_WIN64)
#include <fcntl.h>
```

```
#include <io.h>
#include <windows.h>
#include <winreg.h>
#endif

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "dictionary.h"
#include "mbrola.h"
#include "readclause.h"
#include "synthdata.h"
#include "wavegen.h"

#include "speech.h"
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"
#include "espeak_command.h"
#include "fifo.h"
#include "event.h"

unsigned char *outbuf = NULL;
int outbuf_size = 0;

espeak_EVENT *event_list = NULL;
int event_list_ix = 0;
int n_event_list;
long count_samples;
#ifdef HAVE_PCAUDIOLIB_AUDIO_H
struct audio_object *my_audio = NULL;
#endif

static const char *option_device = NULL;
static unsigned int my_unique_identifier = 0;
```

```
static void *my_user_data = NULL;
static espeak_ng_OUTPUT_MODE my_mode = ENOUTPUT_MODE_SYNCHRONOUS;
static int out_samplerate = 0;
static int voice_samplerate = 22050;
static espeak_ng_STATUS err = ENS_OK;

t_espeak_callback *synth_callback = NULL;
int (*uri_callback)(int, const char *, const char *) = NULL;
int (*phoneme_callback)(const char *) = NULL;

char path_home[N_PATH_HOME]; // this is the espeak-ng-data
directory
extern int saved_parameters[N_SPEECH_PARAM]; // Parameters saved
on synthesis start

void cancel_audio(void)
{
#ifdef HAVE_PCAUDIOLIB_AUDIO_H
 if ((my_mode & ENOUTPUT_MODE_SPEAK_AUDIO) ==
ENOUTPUT_MODE_SPEAK_AUDIO) {
  audio_object_flush(my_audio);
 }
#endif
}

static int dispatch_audio(short *outbuf, int length, espeak_EVENT
*event)
{
 int a_wave_can_be_played = 1;
#ifdef USE_ASYNC
 if ((my_mode & ENOUTPUT_MODE_SYNCHRONOUS) == 0)
  a_wave_can_be_played = fifo_is_command_enabled();
#endif

 switch ((int)my_mode)
 {
 case ENOUTPUT_MODE_SPEAK_AUDIO:
```

```
 case ENOUTPUT_MODE_SPEAK_AUDIO | ENOUTPUT_MODE_SYNCHRONOUS:
 {
  int event_type = 0;
  if (event)
   event_type = event->type;

  if (event_type == espeakEVENT_SAMPLERATE) {
   voice_samplerate = event->id.number;

   if (out_samplerate != voice_samplerate) {
#ifdef HAVE_PCAUDIOLIB_AUDIO_H
    if (out_samplerate != 0) {
     // sound was previously open with a different sample rate
     audio_object_close(my_audio);
     out_samplerate = 0;
#ifdef HAVE_SLEEP
     sleep(1);
#endif
    }
#endif
#ifdef HAVE_PCAUDIOLIB_AUDIO_H
    int error = audio_object_open(my_audio,
AUDIO_OBJECT_FORMAT_S16LE, voice_samplerate, 1);
    if (error != 0) {
     fprintf(stderr, "error: %s\n",
audio_object_strerror(my_audio, error));
     err = ENS_AUDIO_ERROR;
     return -1;
    }
#endif
    out_samplerate = voice_samplerate;
#ifdef USE_ASYNC
    if ((my_mode & ENOUTPUT_MODE_SYNCHRONOUS) == 0)
     event_init();
#endif
   }
  }
```

```
#ifdef HAVE_PCAUDIOLIB_AUDIO_H
  if (out_samplerate == 0) {
   int error = audio_object_open(my_audio,
AUDIO_OBJECT_FORMAT_S16LE, voice_samplerate, 1);
   if (error != 0) {
    fprintf(stderr, "error: %s\n",
audio_object_strerror(my_audio, error));
    err = ENS_AUDIO_ERROR;
    return -1;
   }
   out_samplerate = voice_samplerate;
  }
#endif

#ifdef HAVE_PCAUDIOLIB_AUDIO_H
  if (outbuf && length && a_wave_can_be_played) {
   int error = audio_object_write(my_audio, (char *)outbuf,
2*length);
   if (error != 0)
    fprintf(stderr, "error: %s\n",
audio_object_strerror(my_audio, error));
  }
#endif

#ifdef USE_ASYNC
  while (event && a_wave_can_be_played) {
   // TBD: some event are filtered here but some insight might be
given
   // TBD: in synthesise.cpp for avoiding to create WORDs with
size=0.
   // TBD: For example sentence "or ALT)." returns three words
   // "or", "ALT" and "".
   // TBD: the last one has its size=0.
   if ((event->type == espeakEVENT_WORD) && (event->length == 0))
    break;
   if ((my_mode & ENOUTPUT_MODE_SYNCHRONOUS) == 0) {
```

```
    err = event_declare(event);
    if (err != ENS_EVENT_BUFFER_FULL)
     break;
    usleep(10000);
    a_wave_can_be_played = fifo_is_command_enabled();
   } else
    break;
  }
#endif
 }
  break;
 case 0:
  if (synth_callback)
   synth_callback(outbuf, length, event);
  break;
 }

 return a_wave_can_be_played == 0; // 1 = stop synthesis, -1 =
error
}

static int create_events(short *outbuf, int length, espeak_EVENT
*event_list)
{
 int finished;
 int i = 0;

 // The audio data are written to the output device.
 // The list of events in event_list (index: event_list_ix) is
read:
 // Each event is declared to the "event" object which stores
them internally.
 // The event object is responsible of calling the external
callback
 // as soon as the relevant audio sample is played.

 do { // for each event
```

```
  espeak_EVENT *event;
  if (event_list_ix == 0)
   event = NULL;
  else
   event = event_list + i;
  finished = dispatch_audio((short *)outbuf, length, event);
  length = 0; // the wave data are played once.
  i++;
 } while ((i < event_list_ix) && !finished);
 return finished;
}

#ifdef USE_ASYNC

int sync_espeak_terminated_msg(uint32_t unique_identifier, void
*user_data)
{
 int finished = 0;

 memset(event_list, 0, 2*sizeof(espeak_EVENT));

 event_list[0].type = espeakEVENT_MSG_TERMINATED;
 event_list[0].unique_identifier = unique_identifier;
 event_list[0].user_data = user_data;
 event_list[1].type = espeakEVENT_LIST_TERMINATED;
 event_list[1].unique_identifier = unique_identifier;
 event_list[1].user_data = user_data;

 if (my_mode == ENOUTPUT_MODE_SPEAK_AUDIO) {
  while (1) {
   err = event_declare(event_list);
   if (err != ENS_EVENT_BUFFER_FULL)
    break;
   usleep(10000);
  }
 } else if (synth_callback)
  finished = synth_callback(NULL, 0, event_list);
```

```
 return finished;
}

#endif

static int check_data_path(const char *path, int allow_directory)
{
 if (!path) return 0;

 snprintf(path_home, sizeof(path_home), "%s/espeak-ng-data",
path);
 if (GetFileLength(path_home) == -EISDIR)
  return 1;

 if (!allow_directory)
  return 0;

 snprintf(path_home, sizeof(path_home), "%s", path);
 return GetFileLength(path_home) == -EISDIR;
}

#pragma GCC visibility push(default)

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_InitializeOutput(espeak_ng_OUTPUT_MODE output_mode, int
buffer_length, const char *device)
{
 option_device = device;
 my_mode = output_mode;
 out_samplerate = 0;

#ifdef HAVE_PCAUDIOLIB_AUDIO_H
 if (my_audio == NULL)
  my_audio = create_audio_device_object(device, "eSpeak", "Text-
to-Speech");
#endif
```

```
 // buffer_length is in mS, allocate 2 bytes per sample
 if (buffer_length == 0)
  buffer_length = 60;

 outbuf_size = (buffer_length * samplerate)/500;
 out_start = (unsigned char *)realloc(outbuf, outbuf_size);
 if (out_start == NULL)
  return ENOMEM;
 else
  outbuf = out_start;

 // allocate space for event list.  Allow 200 events per second.
 // Add a constant to allow for very small buffer_length
 n_event_list = (buffer_length*200)/1000 + 20;
 espeak_EVENT *new_event_list = (espeak_EVENT
*)realloc(event_list, sizeof(espeak_EVENT) * n_event_list);
 if (new_event_list == NULL)
  return ENOMEM;
 event_list = new_event_list;

 return ENS_OK;
}

int GetFileLength(const char *filename)
{
 struct stat statbuf;

 if (stat(filename, &statbuf) != 0)
  return -errno;

 if (S_ISDIR(statbuf.st_mode))
  return -EISDIR;

 return statbuf.st_size;
}

ESPEAK_NG_API void espeak_ng_InitializePath(const char *path)
```

```
{
 if (check_data_path(path, 1))
  return;

#ifdef PLATFORM_WINDOWS
 HKEY RegKey;
 unsigned long size;
 unsigned long var_type;
 unsigned char buf[sizeof(path_home)-13];

 if (check_data_path(getenv("ESPEAK_DATA_PATH"), 1))
  return;

 buf[0] = 0;
 RegOpenKeyExA(HKEY_LOCAL_MACHINE, "Software\\eSpeak NG", 0,
KEY_READ, &RegKey);
 if (RegKey == NULL)
  RegOpenKeyExA(HKEY_LOCAL_MACHINE,
"Software\\WOW6432Node\\eSpeak NG", 0, KEY_READ, &RegKey);
 size = sizeof(buf);
 var_type = REG_SZ;
 RegQueryValueExA(RegKey, "Path", 0, &var_type, buf, &size);

 if (check_data_path(buf, 1))
  return;
#elif !defined(PLATFORM_DOS)
 if (check_data_path(getenv("ESPEAK_DATA_PATH"), 1))
  return;

 if (check_data_path(getenv("HOME"), 0))
  return;
#endif

 strcpy(path_home, PATH_ESPEAK_DATA);
}

const int param_defaults[N_SPEECH_PARAM] = {
```

```
 0,   // silence (internal use)
 espeakRATE_NORMAL, // rate wpm
 100, // volume
 50,  // pitch
 50,  // range
 0,   // punctuation
 0,   // capital letters
 0,   // wordgap
 0,   // options
 0,   // intonation
 0,
 0,   // emphasis
 0,   // line length
 0,   // voice type
};

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_Initialize(espeak_ng_ERROR_CONTEXT *context)
{
 int param;
 int srate = 22050; // default sample rate 22050 Hz

 // It seems that the wctype functions don't work until the
locale has been set
 // to something other than the default "C".  Then, not only
Latin1 but also the
 // other characters give the correct results with iswalpha()
etc.
 if (setlocale(LC_CTYPE, "C.UTF-8") == NULL) {
  if (setlocale(LC_CTYPE, "UTF-8") == NULL) {
   if (setlocale(LC_CTYPE, "en_US.UTF-8") == NULL)
    setlocale(LC_CTYPE, "");
  }
 }

 espeak_ng_STATUS result = LoadPhData(&srate, context);
 if (result != ENS_OK)
```

```
  return result;

 WavegenInit(srate, 0);
 LoadConfig();

 memset(&current_voice_selected, 0,
sizeof(current_voice_selected));
 SetVoiceStack(NULL, "");
 SynthesizeInit();
 InitNamedata();

 VoiceReset(0);

 for (param = 0; param < N_SPEECH_PARAM; param++)
  param_stack[0].parameter[param] = saved_parameters[param] =
param_defaults[param];

 SetParameter(espeakRATE, espeakRATE_NORMAL, 0);
 SetParameter(espeakVOLUME, 100, 0);
 SetParameter(espeakCAPITALS, option_capitals, 0);
 SetParameter(espeakPUNCTUATION, option_punctuation, 0);
 SetParameter(espeakWORDGAP, 0, 0);

#ifdef USE_ASYNC
 fifo_init();
#endif

 option_phonemes = 0;
 option_phoneme_events = 0;

 return ENS_OK;
}

ESPEAK_NG_API int espeak_ng_GetSampleRate(void)
{
 return samplerate;
}
```

```
#pragma GCC visibility pop

static espeak_ng_STATUS Synthesize(unsigned int
unique_identifier, const void *text, int flags)
{
 // Fill the buffer with output sound
 int length;
 int finished = 0;
 int count_buffers = 0;

 if ((outbuf == NULL) || (event_list == NULL))
  return ENS_NOT_INITIALIZED;

 option_ssml = flags & espeakSSML;
 option_phoneme_input = flags & espeakPHONEMES;
 option_endpause = flags & espeakENDPAUSE;

 count_samples = 0;

 espeak_ng_STATUS status;
 if (translator == NULL) {
  status = espeak_ng_SetVoiceByName("en");
  if (status != ENS_OK)
   return status;
 }

 if (p_decoder == NULL)
  p_decoder = create_text_decoder();

 status = text_decoder_decode_string_multibyte(p_decoder, text,
translator->encoding, flags);
 if (status != ENS_OK)
  return status;

 SpeakNextClause(0);
```

```
 for (;;) {
  out_ptr = outbuf;
  out_end = &outbuf[outbuf_size];
  event_list_ix = 0;
  WavegenFill();

  length = (out_ptr - outbuf)/2;
  count_samples += length;
  event_list[event_list_ix].type = espeakEVENT_LIST_TERMINATED;
// indicates end of event list
  event_list[event_list_ix].unique_identifier =
unique_identifier;
  event_list[event_list_ix].user_data = my_user_data;

  count_buffers++;
  if ((my_mode & ENOUTPUT_MODE_SPEAK_AUDIO) ==
ENOUTPUT_MODE_SPEAK_AUDIO) {
   finished = create_events((short *)outbuf, length, event_list);
   if (finished < 0)
    return ENS_AUDIO_ERROR;
  } else if (synth_callback)
   finished = synth_callback((short *)outbuf, length,
event_list);
  if (finished) {
   SpeakNextClause(2); // stop
   return ENS_SPEECH_STOPPED;
  }

  if (Generate(phoneme_list, &n_phoneme_list, 1) == 0) {
   if (WcmdqUsed() == 0) {
    // don't process the next clause until the previous clause
has finished generating speech.
    // This ensures that <audio> tag (which causes end-of-clause)
is at a sound buffer boundary

    event_list[0].type = espeakEVENT_LIST_TERMINATED;
    event_list[0].unique_identifier = my_unique_identifier;
```

```
    event_list[0].user_data = my_user_data;

    if (SpeakNextClause(1) == 0) {
     finished = 0;
     if ((my_mode & ENOUTPUT_MODE_SPEAK_AUDIO) ==
ENOUTPUT_MODE_SPEAK_AUDIO) {
      if (dispatch_audio(NULL, 0, NULL) < 0)
       return ENS_AUDIO_ERROR;
     } else if (synth_callback)
      finished = synth_callback(NULL, 0, event_list); // NULL
buffer ptr indicates end of data
     if (finished) {
      SpeakNextClause(2); // stop
      return ENS_SPEECH_STOPPED;
     }
     return ENS_OK;
    }
   }
  }
 }
}

void MarkerEvent(int type, unsigned int char_position, int value,
int value2, unsigned char *out_ptr)
{
 // type: 1=word, 2=sentence, 3=named mark, 4=play audio, 5=end,
7=phoneme
 espeak_EVENT *ep;
 double time;

 if ((event_list == NULL) || (event_list_ix >= (n_event_list-2)))
  return;

 ep = &event_list[event_list_ix++];
 ep->type = (espeak_EVENT_TYPE)type;
 ep->unique_identifier = my_unique_identifier;
 ep->user_data = my_user_data;
```

```
 ep->text_position = char_position & 0xffffff;
 ep->length = char_position >> 24;

 time = ((double)(count_samples + mbrola_delay + (out_ptr -
out_start)/2)*1000.0)/samplerate;
 ep->audio_position = (int)time;
 ep->sample = (count_samples + mbrola_delay + (out_ptr -
out_start)/2);

 if ((type == espeakEVENT_MARK) || (type == espeakEVENT_PLAY))
  ep->id.name = &namedata[value];
 else if (type == espeakEVENT_PHONEME) {
  int *p;
  p = (int *)(ep->id.string);
  p[0] = value;
  p[1] = value2;
 } else
  ep->id.number = value;
}

espeak_ng_STATUS sync_espeak_Synth(unsigned int
unique_identifier, const void *text,
                                   unsigned int position,
espeak_POSITION_TYPE position_type,
                                   unsigned int end_position,
unsigned int flags, void *user_data)
{
 InitText(flags);
 my_unique_identifier = unique_identifier;
 my_user_data = user_data;

 for (int i = 0; i < N_SPEECH_PARAM; i++)
  saved_parameters[i] = param_stack[0].parameter[i];

 switch (position_type)
 {
 case POS_CHARACTER:
```

```
  skip_characters = position;
  break;
 case POS_WORD:
  skip_words = position;
  break;
 case POS_SENTENCE:
  skip_sentences = position;
  break;

 }
 if (skip_characters || skip_words || skip_sentences)
  skipping_text = true;

 end_character_position = end_position;

 espeak_ng_STATUS aStatus = Synthesize(unique_identifier, text,
flags);
#ifdef HAVE_PCAUDIOLIB_AUDIO_H
 if ((my_mode & ENOUTPUT_MODE_SPEAK_AUDIO) ==
ENOUTPUT_MODE_SPEAK_AUDIO) {
  int error = (aStatus == ENS_SPEECH_STOPPED)
            ? audio_object_flush(my_audio)
            : audio_object_drain(my_audio);
  if (error != 0)
   fprintf(stderr, "error: %s\n", audio_object_strerror(my_audio,
error));
 }
#endif

 return aStatus;
}

espeak_ng_STATUS sync_espeak_Synth_Mark(unsigned int
unique_identifier, const void *text,
                                        const char *index_mark,
unsigned int end_position,
                                        unsigned int flags, void
```

```
*user_data)
{
 InitText(flags);

 my_unique_identifier = unique_identifier;
 my_user_data = user_data;

 if (index_mark != NULL) {
  strncpy0(skip_marker, index_mark, sizeof(skip_marker));
  skipping_text = true;
 }

 end_character_position = end_position;

 return Synthesize(unique_identifier, text, flags | espeakSSML);
}

espeak_ng_STATUS sync_espeak_Key(const char *key)
{
 // symbolic name, symbolicname_character  - is there a system
resource of symbolic names per language?
 int letter;
 int ix;

 ix = utf8_in(&letter, key);
 if (key[ix] == 0) // a single character
  return sync_espeak_Char(letter);

 my_unique_identifier = 0;
 my_user_data = NULL;
 return Synthesize(0, key, 0); // speak key as a text string
}

espeak_ng_STATUS sync_espeak_Char(wchar_t character)
{
 // is there a system resource of character names per language?
 char buf[80];
```

```
 my_unique_identifier = 0;
 my_user_data = NULL;

 sprintf(buf, "<say-as interpret-as=\"tts:char\">&#%d;</say-as>",
character);
 return Synthesize(0, buf, espeakSSML);
}

void sync_espeak_SetPunctuationList(const wchar_t *punctlist)
{
 // Set the list of punctuation which are spoken for "some".
 my_unique_identifier = 0;
 my_user_data = NULL;

 option_punctlist[0] = 0;
 if (punctlist != NULL) {
  wcsncpy(option_punctlist, punctlist, N_PUNCTLIST);
  option_punctlist[N_PUNCTLIST-1] = 0;
 }
}

#pragma GCC visibility push(default)

ESPEAK_API void espeak_SetSynthCallback(t_espeak_callback
*SynthCallback)
{
 synth_callback = SynthCallback;
#ifdef USE_ASYNC
 event_set_callback(synth_callback);
#endif
}

ESPEAK_API void espeak_SetUriCallback(int (*UriCallback)(int,
const char *, const char *))
{
 uri_callback = UriCallback;
}
```

```
ESPEAK_API void espeak_SetPhonemeCallback(int
(*PhonemeCallback)(const char *))
{
 phoneme_callback = PhonemeCallback;
}

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_Synthesize(const void *text, size_t size,
                     unsigned int position,
                     espeak_POSITION_TYPE position_type,
                     unsigned int end_position, unsigned int
flags,
                     unsigned int *unique_identifier, void
*user_data)
{
 (void)size; // unused in non-async modes

 static unsigned int temp_identifier;

 if (unique_identifier == NULL)
  unique_identifier = &temp_identifier;

 if (my_mode & ENOUTPUT_MODE_SYNCHRONOUS)
  return sync_espeak_Synth(0, text, position, position_type,
end_position, flags, user_data);

#ifdef USE_ASYNC
 // Create the text command
 t_espeak_command *c1 = create_espeak_text(text, size, position,
position_type, end_position, flags, user_data);
 if (c1) {
  // Retrieve the unique identifier
  *unique_identifier = c1->u.my_text.unique_identifier;
 }

 // Create the "terminated msg" command (same uid)
```

```
 t_espeak_command *c2 =
create_espeak_terminated_msg(*unique_identifier, user_data);

 // Try to add these 2 commands (single transaction)
 if (c1 && c2) {
  espeak_ng_STATUS status = fifo_add_commands(c1, c2);
  if (status != ENS_OK) {
   delete_espeak_command(c1);
   delete_espeak_command(c2);
  }
  return status;
 }

 delete_espeak_command(c1);
 delete_espeak_command(c2);
 return ENOMEM;
#else
 return sync_espeak_Synth(0, text, position, position_type,
end_position, flags, user_data);
#endif
}

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_SynthesizeMark(const void *text,
                         size_t size,
                         const char *index_mark,
                         unsigned int end_position,
                         unsigned int flags,
                         unsigned int *unique_identifier,
                         void *user_data)
{
 (void)size; // unused in non-async modes

 static unsigned int temp_identifier;

 if (unique_identifier == NULL)
  unique_identifier = &temp_identifier;
```

```
 if (my_mode & ENOUTPUT_MODE_SYNCHRONOUS)
  return sync_espeak_Synth_Mark(0, text, index_mark,
end_position, flags, user_data);

#ifdef USE_ASYNC
 // Create the mark command
 t_espeak_command *c1 = create_espeak_mark(text, size,
index_mark, end_position,
                                           flags, user_data);
 if (c1) {
  // Retrieve the unique identifier
  *unique_identifier = c1->u.my_mark.unique_identifier;
 }

 // Create the "terminated msg" command (same uid)
 t_espeak_command *c2 =
create_espeak_terminated_msg(*unique_identifier, user_data);

 // Try to add these 2 commands (single transaction)
 if (c1 && c2) {
  espeak_ng_STATUS status = fifo_add_commands(c1, c2);
  if (status != ENS_OK) {
   delete_espeak_command(c1);
   delete_espeak_command(c2);
  }
  return status;
 }

 delete_espeak_command(c1);
 delete_espeak_command(c2);
 return ENOMEM;
#else
 return sync_espeak_Synth_Mark(0, text, index_mark, end_position,
flags, user_data);
#endif
}
```

```
ESPEAK_NG_API espeak_ng_STATUS espeak_ng_SpeakKeyName(const char
*key_name)
{
 // symbolic name, symbolicname_character  - is there a system
resource of symbolicnames per language

 if (my_mode & ENOUTPUT_MODE_SYNCHRONOUS)
  return sync_espeak_Key(key_name);

#ifdef USE_ASYNC
 t_espeak_command *c = create_espeak_key(key_name, NULL);
 espeak_ng_STATUS status = fifo_add_command(c);
 if (status != ENS_OK)
  delete_espeak_command(c);
 return status;
#else
 return sync_espeak_Key(key_name);
#endif
}

ESPEAK_NG_API espeak_ng_STATUS espeak_ng_SpeakCharacter(wchar_t
character)
{
 // is there a system resource of character names per language?

#ifdef USE_ASYNC
 if (my_mode & ENOUTPUT_MODE_SYNCHRONOUS)
  return sync_espeak_Char(character);

 t_espeak_command *c = create_espeak_char(character, NULL);
 espeak_ng_STATUS status = fifo_add_command(c);
 if (status != ENS_OK)
  delete_espeak_command(c);
 return status;
#else
 return sync_espeak_Char(character);
```

```
#endif
}

ESPEAK_API int espeak_GetParameter(espeak_PARAMETER parameter,
int current)
{
 // current: 0=default value, 1=current value
 if (current)
  return param_stack[0].parameter[parameter];
 return param_defaults[parameter];
}

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_SetParameter(espeak_PARAMETER parameter, int value, int
relative)
{
#ifdef USE_ASYNC
 if (my_mode & ENOUTPUT_MODE_SYNCHRONOUS)
  return SetParameter(parameter, value, relative);

 t_espeak_command *c = create_espeak_parameter(parameter, value,
relative);

 espeak_ng_STATUS status = fifo_add_command(c);
 if (status != ENS_OK)
  delete_espeak_command(c);
 return status;
#else
 return SetParameter(parameter, value, relative);
#endif
}

ESPEAK_NG_API espeak_ng_STATUS espeak_ng_SetPunctuationList(const
wchar_t *punctlist)
{
 // Set the list of punctuation which are spoken for "some".
```

```
#ifdef USE_ASYNC
 if (my_mode & ENOUTPUT_MODE_SYNCHRONOUS) {
  sync_espeak_SetPunctuationList(punctlist);
  return ENS_OK;
 }

 t_espeak_command *c = create_espeak_punctuation_list(punctlist);
 espeak_ng_STATUS status = fifo_add_command(c);
 if (status != ENS_OK)
  delete_espeak_command(c);
 return status;
#else
 sync_espeak_SetPunctuationList(punctlist);
 return ENS_OK;
#endif
}

ESPEAK_API void espeak_SetPhonemeTrace(int phonememode, FILE
*stream)
{
 /* phonememode:  Controls the output of phoneme symbols for the
text
       bits 0-2:
          value=0  No phoneme output (default)
          value=1  Output the translated phoneme symbols for the
text
          value=2  as (1), but produces IPA phoneme names rather
than ascii
       bit 3:   output a trace of how the translation was done
(showing the matching rules and list entries)
       bit 4:   produce pho data for mbrola
       bit 7:   use (bits 8-23) as a tie within multi-letter
phonemes names
       bits 8-23:  separator character, between phoneme names

    stream   output stream for the phoneme symbols (and trace).
If stream=NULL then it uses stdout.
```

```
 option_phonemes = phonememode;
 f_trans = stream;
 if (stream == NULL)
  f_trans = stderr;
}

ESPEAK_API const char *espeak_TextToPhonemes(const void
**textptr, int textmode, int phonememode)
{
 /* phoneme_mode
     bit 1:   0=eSpeak's ascii phoneme names, 1= International
Phonetic Alphabet (as UTF-8 characters).
     bit 7:   use (bits 8-23) as a tie within multi-letter
phonemes names
     bits 8-23:  separator character, between phoneme names
  */

 if (p_decoder == NULL)
  p_decoder = create_text_decoder();

 if (text_decoder_decode_string_multibyte(p_decoder, *textptr,
translator->encoding, textmode) != ENS_OK)
  return NULL;

 TranslateClause(translator, NULL, NULL);

 return GetTranslatedPhonemeString(phonememode);
}

ESPEAK_NG_API espeak_ng_STATUS espeak_ng_Cancel(void)
{
#ifdef USE_ASYNC
 fifo_stop();
 event_clear_all();
#endif
```

```
#ifdef HAVE_PCAUDIOLIB_AUDIO_H
 if ((my_mode & ENOUTPUT_MODE_SPEAK_AUDIO) ==
ENOUTPUT_MODE_SPEAK_AUDIO)
  audio_object_flush(my_audio);
#endif
 embedded_value[EMBED_T] = 0; // reset echo for pronunciation
announcements

 for (int i = 0; i < N_SPEECH_PARAM; i++)
  SetParameter(i, saved_parameters[i], 0);

 return ENS_OK;
}

ESPEAK_API int espeak_IsPlaying(void)
{
#ifdef USE_ASYNC
 return fifo_is_busy();
#else
 return 0;
#endif
}

ESPEAK_NG_API espeak_ng_STATUS espeak_ng_Synchronize(void)
{
 espeak_ng_STATUS berr = err;
#ifdef USE_ASYNC
 while (espeak_IsPlaying())
  usleep(20000);
#endif
 err = ENS_OK;
 return berr;
}

ESPEAK_NG_API espeak_ng_STATUS espeak_ng_Terminate(void)
{
#ifdef USE_ASYNC
```

```
 fifo_stop();
 fifo_terminate();
 event_terminate();
#endif

 if ((my_mode & ENOUTPUT_MODE_SPEAK_AUDIO) ==
ENOUTPUT_MODE_SPEAK_AUDIO) {
#ifdef HAVE_PCAUDIOLIB_AUDIO_H
  audio_object_close(my_audio);
  audio_object_destroy(my_audio);
  my_audio = NULL;
#endif
  out_samplerate = 0;
 }

 free(event_list);
 event_list = NULL;

 free(outbuf);
 outbuf = NULL;

 FreePhData();
 FreeVoiceList();

 DeleteTranslator(translator);
 translator = NULL;

 if (p_decoder != NULL) {
  destroy_text_decoder(p_decoder);
  p_decoder = NULL;
 }

 return ENS_OK;
}

const char *version_string = PACKAGE_VERSION;
ESPEAK_API const char *espeak_Info(const char **ptr)
```

```
{
 if (ptr != NULL)
  *ptr = path_home;
 return version_string;
}

#pragma GCC visibility pop
```

Chapter 42

./src/libespeak-ng/wavegen.c

```
// this version keeps wavemult window as a constant fraction
// of the cycle length - but that spreads out the HF peaks too
much

#include "config.h"

#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>

#include "wavegen.h"

#include "synthesize.h"
#include "speech.h"
#include "phoneme.h"
#include "voice.h"
```

```
#ifdef INCLUDE_KLATT
#include "klatt.h"
#endif

#if HAVE_SONIC_H
#include "sonic.h"
#endif

#include "sintab.h"

#define N_WAV_BUF   10

voice_t *wvoice = NULL;

FILE *f_log = NULL;
static int option_harmonic1 = 10;
static int flutter_amp = 64;

static int general_amplitude = 60;
static int consonant_amp = 26;

int embedded_value[N_EMBEDDED_VALUES];

static int PHASE_INC_FACTOR;
int samplerate = 0; // this is set by Wavegeninit()
int samplerate_native = 0;

static wavegen_peaks_t peaks[N_PEAKS];
static int peak_harmonic[N_PEAKS];
static int peak_height[N_PEAKS];

int echo_head;
int echo_tail;
int echo_amp = 0;
short echo_buf[N_ECHO_BUF];
static int echo_length = 0; // period (in sample\) to ensure
```

```
completion of echo at the end of speech, set in WavegenSetEcho()

static int voicing;
static RESONATOR rbreath[N_PEAKS];

static int harm_sqrt_n = 0;

#define N_LOWHARM  30
#define MAX_HARMONIC 400 // 400 * 50Hz = 20 kHz, more than enough
static int harm_inc[N_LOWHARM]; // only for these harmonics do we
interpolate amplitude between steps
static int *harmspect;
static int hswitch = 0;
static int hspect[2][MAX_HARMONIC]; // 2 copies, we interpolate
between then
static int max_hval = 0;

static int nsamples = 0; // number to do
static int modulation_type = 0;
static int glottal_flag = 0;
static int glottal_reduce = 0;

WGEN_DATA wdata;

static int amp_ix;
static int amp_inc;
static unsigned char *amplitude_env = NULL;

static int samplecount = 0; // number done
static int samplecount_start = 0; // count at start of this
segment
static int end_wave = 0; // continue to end of wave cycle
static int wavephase;
static int phaseinc;
static int cycle_samples; // number of samples in a cycle at
current pitch
static int cbytes;
```

```
static int hf_factor;

static double minus_pi_t;
static double two_pi_t;

unsigned char *out_ptr;
unsigned char *out_start;
unsigned char *out_end;

// the queue of operations passed to wavegen from sythesize
intptr_t wcmdq[N_WCMDQ][4];
int wcmdq_head = 0;
int wcmdq_tail = 0;

// pitch,speed,
int embedded_default[N_EMBEDDED_VALUES]    = { 0,     50,
espeakRATE_NORMAL, 100, 50,  0,  0, 0, espeakRATE_NORMAL, 0, 0,
0, 0, 0, 0 };
static int embedded_max[N_EMBEDDED_VALUES] = { 0, 0x7fff, 750,
300, 99, 99, 99, 0, 750, 0, 0, 0, 0, 4, 0 };

int current_source_index = 0;

#if HAVE_SONIC_H
static sonicStream sonicSpeedupStream = NULL;
double sonicSpeed = 1.0;
#endif

// 1st index=roughness
// 2nd index=modulation_type
// value: bits 0-3  amplitude (16ths), bits 4-7 every n cycles
#define N_ROUGHNESS 8
static unsigned char modulation_tab[N_ROUGHNESS][8] = {
 { 0, 0x00, 0x00, 0x00, 0, 0x46, 0xf2, 0x29 },
 { 0, 0x2f, 0x00, 0x2f, 0, 0x45, 0xf2, 0x29 },
 { 0, 0x2f, 0x00, 0x2e, 0, 0x45, 0xf2, 0x28 },
 { 0, 0x2e, 0x00, 0x2d, 0, 0x34, 0xf2, 0x28 },
```

```
 { 0, 0x2d, 0x2d, 0x2c, 0, 0x34, 0xf2, 0x28 },
 { 0, 0x2b, 0x2b, 0x2b, 0, 0x34, 0xf2, 0x28 },
 { 0, 0x2a, 0x2a, 0x2a, 0, 0x34, 0xf2, 0x28 },
 { 0, 0x29, 0x29, 0x29, 0, 0x34, 0xf2, 0x28 },
};

// Flutter table, to add natural variations to the pitch
#define N_FLUTTER  0x170
static int Flutter_inc;
static const unsigned char Flutter_tab[N_FLUTTER] = {
 0x80, 0x9b, 0xb5, 0xcb, 0xdc, 0xe8, 0xed, 0xec,
 0xe6, 0xdc, 0xce, 0xbf, 0xb0, 0xa3, 0x98, 0x90,
 0x8c, 0x8b, 0x8c, 0x8f, 0x92, 0x94, 0x95, 0x92,
 0x8c, 0x83, 0x78, 0x69, 0x59, 0x49, 0x3c, 0x31,
 0x2a, 0x29, 0x2d, 0x36, 0x44, 0x56, 0x69, 0x7d,
 0x8f, 0x9f, 0xaa, 0xb1, 0xb2, 0xad, 0xa4, 0x96,
 0x87, 0x78, 0x69, 0x5c, 0x53, 0x4f, 0x4f, 0x55,
 0x5e, 0x6b, 0x7a, 0x88, 0x96, 0xa2, 0xab, 0xb0,

 0xb1, 0xae, 0xa8, 0xa0, 0x98, 0x91, 0x8b, 0x88,
 0x89, 0x8d, 0x94, 0x9d, 0xa8, 0xb2, 0xbb, 0xc0,
 0xc1, 0xbd, 0xb4, 0xa5, 0x92, 0x7c, 0x63, 0x4a,
 0x32, 0x1e, 0x0e, 0x05, 0x02, 0x05, 0x0f, 0x1e,
 0x30, 0x44, 0x59, 0x6d, 0x7f, 0x8c, 0x96, 0x9c,
 0x9f, 0x9f, 0x9d, 0x9b, 0x99, 0x99, 0x9c, 0xa1,
 0xa9, 0xb3, 0xbf, 0xca, 0xd5, 0xdc, 0xe0, 0xde,
 0xd8, 0xcc, 0xbb, 0xa6, 0x8f, 0x77, 0x60, 0x4b,

 0x3a, 0x2e, 0x28, 0x29, 0x2f, 0x3a, 0x48, 0x59,
 0x6a, 0x7a, 0x86, 0x90, 0x94, 0x95, 0x91, 0x89,
 0x80, 0x75, 0x6b, 0x62, 0x5c, 0x5a, 0x5c, 0x61,
 0x69, 0x74, 0x80, 0x8a, 0x94, 0x9a, 0x9e, 0x9d,
 0x98, 0x90, 0x86, 0x7c, 0x71, 0x68, 0x62, 0x60,
 0x63, 0x6b, 0x78, 0x88, 0x9b, 0xaf, 0xc2, 0xd2,
 0xdf, 0xe6, 0xe7, 0xe2, 0xd7, 0xc6, 0xb2, 0x9c,
 0x84, 0x6f, 0x5b, 0x4b, 0x40, 0x39, 0x37, 0x38,
```

```
 0x3d, 0x43, 0x4a, 0x50, 0x54, 0x56, 0x55, 0x52,
 0x4d, 0x48, 0x42, 0x3f, 0x3e, 0x41, 0x49, 0x56,
 0x67, 0x7c, 0x93, 0xab, 0xc3, 0xd9, 0xea, 0xf6,
 0xfc, 0xfb, 0xf4, 0xe7, 0xd5, 0xc0, 0xaa, 0x94,
 0x80, 0x71, 0x64, 0x5d, 0x5a, 0x5c, 0x61, 0x68,
 0x70, 0x77, 0x7d, 0x7f, 0x7f, 0x7b, 0x74, 0x6b,
 0x61, 0x57, 0x4e, 0x48, 0x46, 0x48, 0x4e, 0x59,
 0x66, 0x75, 0x84, 0x93, 0x9f, 0xa7, 0xab, 0xaa,

 0xa4, 0x99, 0x8b, 0x7b, 0x6a, 0x5b, 0x4e, 0x46,
 0x43, 0x45, 0x4d, 0x5a, 0x6b, 0x7f, 0x92, 0xa6,
 0xb8, 0xc5, 0xcf, 0xd3, 0xd2, 0xcd, 0xc4, 0xb9,
 0xad, 0xa1, 0x96, 0x8e, 0x89, 0x87, 0x87, 0x8a,
 0x8d, 0x91, 0x92, 0x91, 0x8c, 0x84, 0x78, 0x68,
 0x55, 0x41, 0x2e, 0x1c, 0x0e, 0x05, 0x01, 0x05,
 0x0f, 0x1f, 0x34, 0x4d, 0x68, 0x81, 0x9a, 0xb0,
 0xc1, 0xcd, 0xd3, 0xd3, 0xd0, 0xc8, 0xbf, 0xb5,

 0xab, 0xa4, 0x9f, 0x9c, 0x9d, 0xa0, 0xa5, 0xaa,
 0xae, 0xb1, 0xb0, 0xab, 0xa3, 0x96, 0x87, 0x76,
 0x63, 0x51, 0x42, 0x36, 0x2f, 0x2d, 0x31, 0x3a,
 0x48, 0x59, 0x6b, 0x7e, 0x8e, 0x9c, 0xa6, 0xaa,
 0xa9, 0xa3, 0x98, 0x8a, 0x7b, 0x6c, 0x5d, 0x52,
 0x4a, 0x48, 0x4a, 0x50, 0x5a, 0x67, 0x75, 0x82
};

// waveform shape table for HF peaks, formants 6,7,8
#define N_WAVEMULT 128
static int wavemult_offset = 0;
static int wavemult_max = 0;

// the presets are for 22050 Hz sample rate.
// A different rate will need to recalculate the presets in
WavegenInit()
static unsigned char wavemult[N_WAVEMULT] = {
   0,   0,   0,   2,   3,   5,   8,  11,  14,  18,  22,  27,  32,
37,  43,  49,
```

```
  55,  62,  69,  76,  83,  90,  98, 105, 113, 121, 128, 136, 144,
152, 159, 166,
 174, 181, 188, 194, 201, 207, 213, 218, 224, 228, 233, 237, 240,
244, 246, 249,
 251, 252, 253, 253, 253, 253, 252, 251, 249, 246, 244, 240, 237,
233, 228, 224,
 218, 213, 207, 201, 194, 188, 181, 174, 166, 159, 152, 144, 136,
128, 121, 113,
 105,  98,  90,  83,  76,  69,  62,  55,  49,  43,  37,  32,  27,
22,  18,  14,
  11,   8,   5,   3,   2,   0,   0,   0,   0,   0,   0,   0,   0,
0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,   0,   0
};

// set from y = pow(2,x) * 128,  x=-1 to 1
unsigned char pitch_adjust_tab[MAX_PITCH_VALUE+1] = {
  64,  65,  66,  67,  68,  69,  70,  71,
  72,  73,  74,  75,  76,  77,  78,  79,
  80,  81,  82,  83,  84,  86,  87,  88,
  89,  91,  92,  93,  94,  96,  97,  98,
 100, 101, 103, 104, 105, 107, 108, 110,
 111, 113, 115, 116, 118, 119, 121, 123,
 124, 126, 128, 130, 132, 133, 135, 137,
 139, 141, 143, 145, 147, 149, 151, 153,
 155, 158, 160, 162, 164, 167, 169, 171,
 174, 176, 179, 181, 184, 186, 189, 191,
 194, 197, 199, 202, 205, 208, 211, 214,
 217, 220, 223, 226, 229, 232, 236, 239,
 242, 246, 249, 252, 254, 255
};

void WcmdqStop()
{
 wcmdq_head = 0;
 wcmdq_tail = 0;
```

```
#if HAVE_SONIC_H
 if (sonicSpeedupStream != NULL) {
  sonicDestroyStream(sonicSpeedupStream);
  sonicSpeedupStream = NULL;
 }
#endif

 if (mbrola_name[0] != 0)
  MbrolaReset();
}

int WcmdqFree()
{
 int i;
 i = wcmdq_head - wcmdq_tail;
 if (i <= 0) i += N_WCMDQ;
 return i;
}

int WcmdqUsed()
{
 return N_WCMDQ - WcmdqFree();
}

void WcmdqInc()
{
 wcmdq_tail++;
 if (wcmdq_tail >= N_WCMDQ) wcmdq_tail = 0;
}

static void WcmdqIncHead()
{
 wcmdq_head++;
 if (wcmdq_head >= N_WCMDQ) wcmdq_head = 0;
}
```

```
#define PEAKSHAPEW 256

unsigned char pk_shape1[PEAKSHAPEW+1] = {
 255, 254, 254, 254, 254, 254, 253, 253, 252, 251, 251, 250, 249,
248, 247, 246,
 245, 244, 242, 241, 239, 238, 236, 234, 233, 231, 229, 227, 225,
223, 220, 218,
 216, 213, 211, 209, 207, 205, 203, 201, 199, 197, 195, 193, 191,
189, 187, 185,
 183, 180, 178, 176, 173, 171, 169, 166, 164, 161, 159, 156, 154,
151, 148, 146,
 143, 140, 138, 135, 132, 129, 126, 123, 120, 118, 115, 112, 108,
105, 102,  99,
  96,  95,  93,  91,  90,  88,  86,  85,  83,  82,  80,  79,  77,
76,  74,  73,
  72,  70,  69,  68,  67,  66,  64,  63,  62,  61,  60,  59,  58,
57,  56,  55,
  55,  54,  53,  52,  52,  51,  50,  50,  49,  48,  48,  47,  47,
46,  46,  46,
  45,  45,  45,  44,  44,  44,  44,  44,  44,  44,  43,  43,  43,
43,  44,  43,
  42,  42,  41,  40,  40,  39,  38,  38,  37,  36,  36,  35,  35,
34,  33,  33,
  32,  32,  31,  30,  30,  29,  29,  28,  28,  27,  26,  26,  25,
25,  24,  24,
  23,  23,  22,  22,  21,  21,  20,  20,  19,  19,  18,  18,  18,
17,  17,  16,
  16,  15,  15,  15,  14,  14,  13,  13,  13,  12,  12,  11,  11,
11,  10,  10,
  10,   9,   9,   9,   8,   8,   8,   7,   7,   7,   7,   6,   6,
6,   5,   5,
   5,   5,   4,   4,   4,   4,   4,   3,   3,   3,   3,   2,   2,
2,   2,   2,
   2,   1,   1,   1,   1,   1,   1,   0,   0,   0,   0,   0,   0,
0,   0,   0,
   0
};
```

```
static unsigned char pk_shape2[PEAKSHAPEW+1] = {
 255, 254, 254, 254, 254, 254, 254, 254, 254, 254, 253, 253, 253,
253, 252, 252,
 252, 251, 251, 251, 250, 250, 249, 249, 248, 248, 247, 247, 246,
245, 245, 244,
 243, 243, 242, 241, 239, 237, 235, 233, 231, 229, 227, 225, 223,
221, 218, 216,
 213, 211, 208, 205, 203, 200, 197, 194, 191, 187, 184, 181, 178,
174, 171, 167,
 163, 160, 156, 152, 148, 144, 140, 136, 132, 127, 123, 119, 114,
110, 105, 100,
  96,  94,  91,  88,  86,  83,  81,  78,  76,  74,  71,  69,  66,
64,  62,  60,
  57,  55,  53,  51,  49,  47,  44,  42,  40,  38,  36,  34,  32,
30,  29,  27,
  25,  23,  21,  19,  18,  16,  14,  12,  11,   9,   7,   6,   4,
3,   1,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,   0,   0,
   0
};

static unsigned char *pk_shape;

void WavegenInit(int rate, int wavemult_fact)
{
 int ix;
 double x;

 if (wavemult_fact == 0)
  wavemult_fact = 60; // default

 wvoice = NULL;
 samplerate = samplerate_native = rate;
 PHASE_INC_FACTOR = 0x8000000 / samplerate; // assumes pitch is
Hz*32
```

```
 Flutter_inc = (64 * samplerate)/rate;
 samplecount = 0;
 nsamples = 0;
 wavephase = 0x7fffffff;
 max_hval = 0;

 wdata.amplitude = 32;
 wdata.amplitude_fmt = 100;

 for (ix = 0; ix < N_EMBEDDED_VALUES; ix++)
  embedded_value[ix] = embedded_default[ix];

 // set up window to generate a spread of harmonics from a
 // single peak for HF peaks
 wavemult_max = (samplerate * wavemult_fact)/(256 * 50);
 if (wavemult_max > N_WAVEMULT) wavemult_max = N_WAVEMULT;

 wavemult_offset = wavemult_max/2;

 if (samplerate != 22050) {
  // wavemult table has preset values for 22050 Hz, we only need
to
  // recalculate them if we have a different sample rate
  for (ix = 0; ix < wavemult_max; ix++) {
   x = 127*(1.0 - cos((M_PI*2)*ix/wavemult_max));
   wavemult[ix] = (int)x;
  }
 }

 pk_shape = pk_shape2;

#ifdef INCLUDE_KLATT
 KlattInit();
#endif
}

int GetAmplitude(void)
```

```
{
 int amp;

 // normal, none, reduced, moderate, strong
 static const unsigned char amp_emphasis[5] = { 16, 16, 10, 16,
22 };

 amp = (embedded_value[EMBED_A])*55/100;
 general_amplitude = amp * amp_emphasis[embedded_value[EMBED_F]]
/ 16;
 return general_amplitude;
}

static void WavegenSetEcho(void)
{
 if (wvoice == NULL)
  return;

 int delay;
 int amp;

 voicing = wvoice->voicing;
 delay = wvoice->echo_delay;
 amp = wvoice->echo_amp;

 if (delay >= N_ECHO_BUF)
  delay = N_ECHO_BUF-1;
 if (amp > 100)
  amp = 100;

 memset(echo_buf, 0, sizeof(echo_buf));
 echo_tail = 0;

 if (embedded_value[EMBED_H] > 0) {
  // set echo from an embedded command in the text
  amp = embedded_value[EMBED_H];
  delay = 130;
```

```
 }

 if (delay == 0)
  amp = 0;

 echo_head = (delay * samplerate)/1000;
 echo_length = echo_head; // ensure completion of echo at the end
of speech. Use 1 delay period?
 if (amp == 0)
  echo_length = 0;
 if (amp > 20)
  echo_length = echo_head * 2; // perhaps allow 2 echo periods if
the echo is loud.

 // echo_amp units are 1/256ths of the amplitude of the original
sound.
 echo_amp = amp;
 // compensate (partially) for increase in amplitude due to echo
 general_amplitude = GetAmplitude();
 general_amplitude = ((general_amplitude * (500-amp))/500);
}

int PeaksToHarmspect(wavegen_peaks_t *peaks, int pitch, int
*htab, int control)
{
 if (wvoice == NULL)
  return 1;

 // Calculate the amplitude of each  harmonics from the formants
 // Only for formants 0 to 5

 // control 0=initial call, 1=every 64 cycles

 // pitch and freqs are Hz<<16

 int f;
 wavegen_peaks_t *p;
```

```
 int fp;  // centre freq of peak
 int fhi; // high freq of peak
 int h;   // harmonic number
 int pk;
 int hmax;
 int hmax_samplerate; // highest harmonic allowed for the
samplerate
 int x;
 int ix;
 int h1;

 // initialise as much of *out as we will need
 hmax = (peaks[wvoice->n_harmonic_peaks].freq +
peaks[wvoice->n_harmonic_peaks].right)/pitch;
 if (hmax >= MAX_HARMONIC)
  hmax = MAX_HARMONIC-1;

 // restrict highest harmonic to half the samplerate
 hmax_samplerate = (((samplerate * 19)/40) << 16)/pitch; // only
95% of Nyquist freq

 if (hmax > hmax_samplerate)
  hmax = hmax_samplerate;

 for (h = 0; h <= hmax; h++)
  htab[h] = 0;

 for (pk = 0; pk <= wvoice->n_harmonic_peaks; pk++) {
  p = &peaks[pk];
  if ((p->height == 0) || (fp = p->freq) == 0)
   continue;

  fhi = p->freq + p->right;
  h = ((p->freq - p->left) / pitch) + 1;
  if (h <= 0) h = 1;

  for (f = pitch*h; f < fp; f += pitch)
```

```
   htab[h++] += pk_shape[(fp-f)/(p->left>>8)] * p->height;
  for (; f < fhi; f += pitch)
   htab[h++] += pk_shape[(f-fp)/(p->right>>8)] * p->height;
 }

 int y;
 int h2;
 // increase bass
 y = peaks[1].height * 10; // addition as a multiple of 1/256s
 h2 = (1000<<16)/pitch; // decrease until 1000Hz
 if (h2 > 0) {
  x = y/h2;
  h = 1;
  while (y > 0) {
   htab[h++] += y;
   y -= x;
  }
 }

 // find the nearest harmonic for HF peaks where we don't use
shape
 for (; pk < N_PEAKS; pk++) {
  x = peaks[pk].height >> 14;
  peak_height[pk] = (x * x * 5)/2;

  // find the nearest harmonic for HF peaks where we don't use
shape
  if (control == 0) {
   // set this initially, but make changes only at the quiet
point
   peak_harmonic[pk] = peaks[pk].freq / pitch;
  }
  // only use harmonics up to half the samplerate
  if (peak_harmonic[pk] >= hmax_samplerate)
   peak_height[pk] = 0;
 }
```

```
 // convert from the square-rooted values
 f = 0;
 for (h = 0; h <= hmax; h++, f += pitch) {
  x = htab[h] >> 15;
  htab[h] = (x * x) >> 8;

  if ((ix = (f >> 19)) < N_TONE_ADJUST)
   htab[h] = (htab[h] * wvoice->tone_adjust[ix]) >> 13; // index
tone_adjust with Hz/8
 }

 // adjust the amplitude of the first harmonic, affects tonal
quality
 h1 = htab[1] * option_harmonic1;
 htab[1] = h1/8;

 // calc intermediate increments of LF harmonics
 if (control & 1) {
  for (h = 1; h < N_LOWHARM; h++)
   harm_inc[h] = (htab[h] - harmspect[h]) >> 3;
 }

 return hmax; // highest harmonic number
}

static void AdvanceParameters()
{
 // Called every 64 samples to increment the formant freq,
height, and widths
 if (wvoice == NULL)
  return;

 int x;
 int ix;
 static int Flutter_ix = 0;

 // advance the pitch
```

```
wdata.pitch_ix += wdata.pitch_inc;
if ((ix = wdata.pitch_ix>>8) > 127) ix = 127;
x = wdata.pitch_env[ix] * wdata.pitch_range;
wdata.pitch = (x>>8) + wdata.pitch_base;

amp_ix += amp_inc;

/* add pitch flutter */
if (Flutter_ix >= (N_FLUTTER*64))
 Flutter_ix = 0;
x = ((int)(Flutter_tab[Flutter_ix >> 6])-0x80) * flutter_amp;
Flutter_ix += Flutter_inc;
wdata.pitch += x;
if (wdata.pitch < 102400)
 wdata.pitch = 102400; // min pitch, 25 Hz  (25 << 12)

if (samplecount == samplecount_start)
 return;

for (ix = 0; ix <= wvoice->n_harmonic_peaks; ix++) {
 peaks[ix].freq1 += peaks[ix].freq_inc;
 peaks[ix].freq = (int)peaks[ix].freq1;
 peaks[ix].height1 += peaks[ix].height_inc;
 if ((peaks[ix].height = (int)peaks[ix].height1) < 0)
  peaks[ix].height = 0;
 peaks[ix].left1 += peaks[ix].left_inc;
 peaks[ix].left = (int)peaks[ix].left1;
 if (ix < 3) {
  peaks[ix].right1 += peaks[ix].right_inc;
  peaks[ix].right = (int)peaks[ix].right1;
 } else
  peaks[ix].right = peaks[ix].left;
}
for (; ix < 8; ix++) {
 // formants 6,7,8 don't have a width parameter
 if (ix < 7) {
  peaks[ix].freq1 += peaks[ix].freq_inc;
```

```
   peaks[ix].freq = (int)peaks[ix].freq1;
  }
  peaks[ix].height1 += peaks[ix].height_inc;
  if ((peaks[ix].height = (int)peaks[ix].height1) < 0)
   peaks[ix].height = 0;
 }
}

static double resonator(RESONATOR *r, double input)
{
 double x;

 x = r->a * input + r->b * r->x1 + r->c * r->x2;
 r->x2 = r->x1;
 r->x1 = x;

 return x;
}

static void setresonator(RESONATOR *rp, int freq, int bwidth, int
init)
{
 // freq    Frequency of resonator in Hz
 // bwidth  Bandwidth of resonator in Hz
 // init    Initialize internal data

 double x;
 double arg;

 if (init) {
  rp->x1 = 0;
  rp->x2 = 0;
 }

 arg = minus_pi_t * bwidth;
 x = exp(arg);
```

```
 rp->c = -(x * x);

 arg = two_pi_t * freq;
 rp->b = x * cos(arg) * 2.0;

 rp->a = 1.0 - rp->b - rp->c;
}

void InitBreath(void)
{
 int ix;

 minus_pi_t = -M_PI / samplerate;
 two_pi_t = -2.0 * minus_pi_t;

 for (ix = 0; ix < N_PEAKS; ix++)
  setresonator(&rbreath[ix], 2000, 200, 1);
}

static void SetBreath()
{
 int pk;

 if (wvoice == NULL || wvoice->breath[0] == 0)
  return;

 for (pk = 1; pk < N_PEAKS; pk++) {
  if (wvoice->breath[pk] != 0) {
   // breath[0] indicates that some breath formants are needed
   // set the freq from the current synthesis formant and the
width from the voice data
   setresonator(&rbreath[pk], peaks[pk].freq >> 16,
wvoice->breathw[pk], 0);
  }
 }
}
```

```
static int ApplyBreath(void)
{
 if (wvoice == NULL)
  return 0;

 int value = 0;
 int noise;
 int ix;
 int amp;

 // use two random numbers, for alternate formants
 noise = (rand() & 0x3fff) - 0x2000;

 for (ix = 1; ix < N_PEAKS; ix++) {
  if ((amp = wvoice->breath[ix]) != 0) {
   amp *= (peaks[ix].height >> 14);
   value += (int)resonator(&rbreath[ix], noise) * amp;
  }
 }
 return value;
}

static int Wavegen()
{
 if (wvoice == NULL)
  return 0;

 unsigned short waveph;
 unsigned short theta;
 int total;
 int h;
 int ix;
 int z, z1, z2;
 int echo;
 int ov;
 static int maxh, maxh2;
 int pk;
```

```
 signed char c;
 int sample;
 int amp;
 int modn_amp = 1, modn_period;
 static int agc = 256;
 static int h_switch_sign = 0;
 static int cycle_count = 0;
 static int amplitude2 = 0; // adjusted for pitch

 // continue until the output buffer is full, or
 // the required number of samples have been produced

 for (;;) {
  if ((end_wave == 0) && (samplecount == nsamples))
   return 0;

  if ((samplecount & 0x3f) == 0) {
   // every 64 samples, adjust the parameters
   if (samplecount == 0) {
    hswitch = 0;
    harmspect = hspect[0];
    maxh2 = PeaksToHarmspect(peaks, wdata.pitch<<4, hspect[0],
0);

    // adjust amplitude to compensate for fewer harmonics at
higher pitch
    amplitude2 = (wdata.amplitude * (wdata.pitch >> 8) *
wdata.amplitude_fmt)/(10000 << 3);

    // switch sign of harmonics above about 900Hz, to reduce max
peak amplitude
    h_switch_sign = 890 / (wdata.pitch >> 12);
   } else
    AdvanceParameters();

   // pitch is Hz<<12
   phaseinc = (wdata.pitch>>7) * PHASE_INC_FACTOR;
```

```
   cycle_samples = samplerate/(wdata.pitch >> 12); //
sr/(pitch*2)
   hf_factor = wdata.pitch >> 11;

   maxh = maxh2;
   harmspect = hspect[hswitch];
   hswitch ^= 1;
   maxh2 = PeaksToHarmspect(peaks, wdata.pitch<<4,
hspect[hswitch], 1);

   SetBreath();
  } else if ((samplecount & 0x07) == 0) {
   for (h = 1; h < N_LOWHARM && h <= maxh2 && h <= maxh; h++)
    harmspect[h] += harm_inc[h];

   // bring automatic gain control back towards unity
   if (agc < 256) agc++;
  }

  samplecount++;

  if (wavephase > 0) {
   wavephase += phaseinc;
   if (wavephase < 0) {
    // sign has changed, reached a quiet point in the waveform
    cbytes = wavemult_offset - (cycle_samples)/2;
    if (samplecount > nsamples)
     return 0;

    cycle_count++;

    for (pk = wvoice->n_harmonic_peaks+1; pk < N_PEAKS; pk++) {
     // find the nearest harmonic for HF peaks where we don't use
shape
     peak_harmonic[pk] = ((peaks[pk].freq / (wdata.pitch*8)) + 1)
/ 2;
    }
```

```
    // adjust amplitude to compensate for fewer harmonics at
higher pitch
    amplitude2 = (wdata.amplitude * (wdata.pitch >> 8) *
wdata.amplitude_fmt)/(10000 << 3);

    if (glottal_flag > 0) {
     if (glottal_flag == 3) {
      if ((nsamples-samplecount) < (cycle_samples*2)) {
       // Vowel before glottal-stop.
       // This is the start of the penultimate cycle, reduce its
amplitude
       glottal_flag = 2;
       amplitude2 = (amplitude2 *  glottal_reduce)/256;
      }
     } else if (glottal_flag == 4) {
      // Vowel following a glottal-stop.
      // This is the start of the second cycle, reduce its
amplitude
      glottal_flag = 2;
      amplitude2 = (amplitude2 * glottal_reduce)/256;
     } else
      glottal_flag--;
    }

    if (amplitude_env != NULL) {
     // amplitude envelope is only used for creaky voice effect
on certain vowels/tones
     if ((ix = amp_ix>>8) > 127) ix = 127;
     amp = amplitude_env[ix];
     amplitude2 = (amplitude2 * amp)/128;
    }

    // introduce roughness into the sound by reducing the
amplitude of
    modn_period = 0;
    if (voice->roughness < N_ROUGHNESS) {
```

```
     modn_period =
modulation_tab[voice->roughness][modulation_type];
     modn_amp = modn_period & 0xf;
     modn_period = modn_period >> 4;
    }

    if (modn_period != 0) {
     if (modn_period == 0xf) {
      // just once */
      amplitude2 = (amplitude2 * modn_amp)/16;
      modulation_type = 0;
     } else {
      // reduce amplitude every [modn_period} cycles
      if ((cycle_count % modn_period) == 0)
       amplitude2 = (amplitude2 * modn_amp)/16;
     }
    }
   }
  } else
   wavephase += phaseinc;
  waveph = (unsigned short)(wavephase >> 16);
  total = 0;

  // apply HF peaks, formants 6,7,8
  // add a single harmonic and then spread this my multiplying by
a
  // window.  This is to reduce the processing power needed to
add the
  // higher frequence harmonics.
  cbytes++;
  if (cbytes >= 0 && cbytes < wavemult_max) {
   for (pk = wvoice->n_harmonic_peaks+1; pk < N_PEAKS; pk++) {
    theta = peak_harmonic[pk] * waveph;
    total += (long)sin_tab[theta >> 5] * peak_height[pk];
   }

   // spread the peaks by multiplying by a window
```

```
   total = (long)(total / hf_factor) * wavemult[cbytes];
  }

  // apply main peaks, formants 0 to 5
#ifdef USE_ASSEMBLER_1
  // use an optimised routine for this loop, if available
  total += AddSineWaves(waveph, h_switch_sign, maxh, harmspect);
// call an assembler code routine
#else
  theta = waveph;

  for (h = 1; h <= h_switch_sign; h++) {
   total += ((int)sin_tab[theta >> 5] * harmspect[h]);
   theta += waveph;
  }
  while (h <= maxh) {
   total -= ((int)sin_tab[theta >> 5] * harmspect[h]);
   theta += waveph;
   h++;
  }
#endif

  if (voicing != 64)
   total = (total >> 6) * voicing;

  if (wvoice->breath[0])
   total +=  ApplyBreath();

  // mix with sampled wave if required
  z2 = 0;
  if (wdata.mix_wavefile_ix < wdata.n_mix_wavefile) {
   if (wdata.mix_wave_scale == 0) {
    // a 16 bit sample
    c = wdata.mix_wavefile[wdata.mix_wavefile_ix+wdata.mix_wavefi
le_offset+1];
    sample = wdata.mix_wavefile[wdata.mix_wavefile_ix+wdata.mix_w
avefile_offset] + (c * 256);
```

```
    wdata.mix_wavefile_ix += 2;
   } else {
    // a 8 bit sample, scaled
    sample = (signed char)wdata.mix_wavefile[wdata.mix_wavefile_o
ffset+wdata.mix_wavefile_ix++] * wdata.mix_wave_scale;
   }
   z2 = (sample * wdata.amplitude_v) >> 10;
   z2 = (z2 * wdata.mix_wave_amp)/32;

   if ((wdata.mix_wavefile_ix + wdata.mix_wavefile_offset) >=
wdata.mix_wavefile_max)  // reached the end of available WAV data
    wdata.mix_wavefile_offset -= (wdata.mix_wavefile_max*3)/4;
  }

  z1 = z2 + (((total>>8) * amplitude2) >> 13);

  echo = (echo_buf[echo_tail++] * echo_amp);
  z1 += echo >> 8;
  if (echo_tail >= N_ECHO_BUF)
   echo_tail = 0;

  z = (z1 * agc) >> 8;

  // check for overflow, 16bit signed samples
  if (z >= 32768) {
   ov = 8388608/z1 - 1;      // 8388608 is 2^23, i.e. max value *
256
   if (ov < agc) agc = ov;    // set agc to number of 1/256ths to
multiply the sample by
   z = (z1 * agc) >> 8;      // reduce sample by agc value to
prevent overflow
  } else if (z <= -32768) {
   ov = -8388608/z1 - 1;
   if (ov < agc) agc = ov;
   z = (z1 * agc) >> 8;
  }
  *out_ptr++ = z;
```

```
  *out_ptr++ = z >> 8;

  echo_buf[echo_head++] = z;
  if (echo_head >= N_ECHO_BUF)
   echo_head = 0;

  if (out_ptr + 2 > out_end)
   return 1;
 }
}

static int PlaySilence(int length, bool resume)
{
 static int n_samples;
 int value = 0;

 nsamples = 0;
 samplecount = 0;
 wavephase = 0x7fffffff;

 if (length == 0)
  return 0;

 if (resume == false)
  n_samples = length;

 while (n_samples-- > 0) {
  value = (echo_buf[echo_tail++] * echo_amp) >> 8;

  if (echo_tail >= N_ECHO_BUF)
   echo_tail = 0;

  *out_ptr++ = value;
  *out_ptr++ = value >> 8;

  echo_buf[echo_head++] = value;
  if (echo_head >= N_ECHO_BUF)
```

```
   echo_head = 0;

  if (out_ptr + 2 > out_end)
   return 1;
 }
 return 0;
}

static int PlayWave(int length, bool resume, unsigned char *data,
int scale, int amp)
{
 static int n_samples;
 static int ix = 0;
 int value;
 signed char c;

 if (resume == false) {
  n_samples = length;
  ix = 0;
 }

 nsamples = 0;
 samplecount = 0;

 while (n_samples-- > 0) {
  if (scale == 0) {
   // 16 bits data
   c = data[ix+1];
   value = data[ix] + (c * 256);
   ix += 2;
  } else {
   // 8 bit data, shift by the specified scale factor
   value = (signed char)data[ix++] * scale;
  }
  value *= (consonant_amp * general_amplitude); // reduce
strength of consonant
  value = value >> 10;
```

```
  value = (value * amp)/32;

  value += ((echo_buf[echo_tail++] * echo_amp) >> 8);

  if (value > 32767)
   value = 32768;
  else if (value < -32768)
   value = -32768;

  if (echo_tail >= N_ECHO_BUF)
   echo_tail = 0;

  out_ptr[0] = value;
  out_ptr[1] = value >> 8;
  out_ptr += 2;

  echo_buf[echo_head++] = (value*3)/4;
  if (echo_head >= N_ECHO_BUF)
   echo_head = 0;

  if (out_ptr + 2 > out_end)
   return 1;
 }
 return 0;
}

static int SetWithRange0(int value, int max)
{
 if (value < 0)
  return 0;
 if (value > max)
  return max;
 return value;
}

static void SetPitchFormants()
{
```

```
 if (wvoice == NULL)
  return;

 int ix;
 int factor = 256;
 int pitch_value;

 // adjust formants to give better results for a different voice
pitch
 if ((pitch_value = embedded_value[EMBED_P]) > MAX_PITCH_VALUE)
  pitch_value = MAX_PITCH_VALUE;

 if (pitch_value > 50) {
  // only adjust if the pitch is higher than normal
  factor = 256 + (25 * (pitch_value - 50))/50;
 }

 for (ix = 0; ix <= 5; ix++)
  wvoice->freq[ix] = (wvoice->freq2[ix] * factor)/256;

 factor = embedded_value[EMBED_T]*3;
 wvoice->height[0] = (wvoice->height2[0] * (256 - factor*2))/256;
 wvoice->height[1] = (wvoice->height2[1] * (256 - factor))/256;
}

void SetEmbedded(int control, int value)
{
 // there was an embedded command in the text at this point
 int sign = 0;
 int command;

 command = control & 0x1f;
 if ((control & 0x60) == 0x60)
  sign = -1;
 else if ((control & 0x60) == 0x40)
  sign = 1;
```

```
 if (command < N_EMBEDDED_VALUES) {
  if (sign == 0)
   embedded_value[command] = value;
  else
   embedded_value[command] += (value * sign);
  embedded_value[command] =
SetWithRange0(embedded_value[command], embedded_max[command]);
 }

 switch (command)
 {
 case EMBED_T:
  WavegenSetEcho(); // and drop through to case P
 case EMBED_P:
  SetPitchFormants();
  break;
 case EMBED_A: // amplitude
  general_amplitude = GetAmplitude();
  break;
 case EMBED_F: // emphasis
  general_amplitude = GetAmplitude();
  break;
 case EMBED_H:
  WavegenSetEcho();
  break;
 }
}

void WavegenSetVoice(voice_t *v)
{
 static voice_t v2;

 memcpy(&v2, v, sizeof(v2));
 wvoice = &v2;

 if (v->peak_shape == 0)
  pk_shape = pk_shape1;
```

```
 else
  pk_shape = pk_shape2;

 consonant_amp = (v->consonant_amp * 26) /100;
 if (samplerate <= 11000) {
  consonant_amp = consonant_amp*2; // emphasize consonants at low
sample rates
  option_harmonic1 = 6;
 }
 WavegenSetEcho();
 SetPitchFormants();
 MarkerEvent(espeakEVENT_SAMPLERATE, 0, wvoice->samplerate, 0,
out_ptr);
}

static void SetAmplitude(int length, unsigned char *amp_env, int
value)
{
 if (wvoice == NULL)
  return;

 amp_ix = 0;
 if (length == 0)
  amp_inc = 0;
 else
  amp_inc = (256 * ENV_LEN * STEPSIZE)/length;

 wdata.amplitude = (value * general_amplitude)/16;
 wdata.amplitude_v = (wdata.amplitude * wvoice->consonant_ampv *
15)/100; // for wave mixed with voiced sounds

 amplitude_env = amp_env;
}

void SetPitch2(voice_t *voice, int pitch1, int pitch2, int
*pitch_base, int *pitch_range)
{
```

```
 int x;
 int base;
 int range;
 int pitch_value;

 if (pitch1 > pitch2) {
  x = pitch1; // swap values
  pitch1 = pitch2;
  pitch2 = x;
 }

 if ((pitch_value = embedded_value[EMBED_P]) > MAX_PITCH_VALUE)
  pitch_value = MAX_PITCH_VALUE;
 pitch_value -= embedded_value[EMBED_T]; // adjust tone for
announcing punctuation
 if (pitch_value < 0)
  pitch_value = 0;

 base = (voice->pitch_base * pitch_adjust_tab[pitch_value])/128;
 range =  (voice->pitch_range * embedded_value[EMBED_R])/50;

 // compensate for change in pitch when the range is narrowed or
widened
 base -= (range - voice->pitch_range)*18;

}

static void SetPitch(int length, unsigned char *env, int pitch1,
int pitch2)
{
 if (wvoice == NULL)
  return;

 // length in samples

 if ((wdata.pitch_env = env) == NULL)
  wdata.pitch_env = env_fall; // default
```

```
 wdata.pitch_ix = 0;
 if (length == 0)
  wdata.pitch_inc = 0;
 else
  wdata.pitch_inc = (256 * ENV_LEN * STEPSIZE)/length;

 SetPitch2(wvoice, pitch1, pitch2, &wdata.pitch_base,
&wdata.pitch_range);
 // set initial pitch
 wdata.pitch = ((wdata.pitch_env[0] * wdata.pitch_range) >>8) +
wdata.pitch_base; // Hz << 12

 flutter_amp = wvoice->flutter;
}

static void SetSynth(int length, int modn, frame_t *fr1, frame_t
*fr2, voice_t *v)
{
 if (wvoice == NULL || v == NULL)
  return;

 int ix;
 double next;
 int length2;
 int length4;
 int qix;
 int cmd;
 static int glottal_reduce_tab1[4] = { 0x30, 0x30, 0x40, 0x50 };
// vowel before [?], amp * 1/256
 static int glottal_reduce_tab2[4] = { 0x90, 0xa0, 0xb0, 0xc0 };
// vowel after [?], amp * 1/256

 harm_sqrt_n = 0;
 end_wave = 1;

 // any additional information in the param1 ?
```

```
 modulation_type = modn & 0xff;

 glottal_flag = 0;
 if (modn & 0x400) {
  glottal_flag = 3; // before a glottal stop
  glottal_reduce = glottal_reduce_tab1[(modn >> 8) & 3];
 }
 if (modn & 0x800) {
  glottal_flag = 4; // after a glottal stop
  glottal_reduce = glottal_reduce_tab2[(modn >> 8) & 3];
 }

 for (qix = wcmdq_head+1;; qix++) {
  if (qix >= N_WCMDQ) qix = 0;
  if (qix == wcmdq_tail) break;

  cmd = wcmdq[qix][0];
  if (cmd == WCMD_SPECT) {
   end_wave = 0; // next wave generation is from another spectrum
   break;
  }
  if ((cmd == WCMD_WAVE) || (cmd == WCMD_PAUSE))
   break; // next is not from spectrum, so continue until end of
wave cycle
 }

 // round the length to a multiple of the stepsize
 length2 = (length + STEPSIZE/2) & ~0x3f;
 if (length2 == 0)
  length2 = STEPSIZE;

 // add this length to any left over from the previous synth
 samplecount_start = samplecount;
 nsamples += length2;

 length4 = length2/4;
```

```
 peaks[7].freq = (7800  * v->freq[7] + v->freqadd[7]*256) << 8;
 peaks[8].freq = (9000  * v->freq[8] + v->freqadd[8]*256) << 8;

 for (ix = 0; ix < 8; ix++) {
  if (ix < 7) {
   peaks[ix].freq1 = (fr1->ffreq[ix] * v->freq[ix] +
v->freqadd[ix]*256) << 8;
   peaks[ix].freq = (int)peaks[ix].freq1;
   next = (fr2->ffreq[ix] * v->freq[ix] + v->freqadd[ix]*256) <<
8;
   peaks[ix].freq_inc =  ((next - peaks[ix].freq1) *
(STEPSIZE/4)) / length4; // lower headroom for fixed point math
  }

  peaks[ix].height1 = (fr1->fheight[ix] * v->height[ix]) << 6;
  peaks[ix].height = (int)peaks[ix].height1;
  next = (fr2->fheight[ix] * v->height[ix]) << 6;
  peaks[ix].height_inc =  ((next - peaks[ix].height1) * STEPSIZE)
/ length2;

  if ((ix <= 5) && (ix <= wvoice->n_harmonic_peaks)) {
   peaks[ix].left1 = (fr1->fwidth[ix] * v->width[ix]) << 10;
   peaks[ix].left = (int)peaks[ix].left1;
   next = (fr2->fwidth[ix] * v->width[ix]) << 10;
   peaks[ix].left_inc =  ((next - peaks[ix].left1) * STEPSIZE) /
length2;

   if (ix < 3) {
    peaks[ix].right1 = (fr1->fright[ix] * v->width[ix]) << 10;
    peaks[ix].right = (int)peaks[ix].right1;
    next = (fr2->fright[ix] * v->width[ix]) << 10;
    peaks[ix].right_inc = ((next - peaks[ix].right1) * STEPSIZE)
/ length2;
   } else
    peaks[ix].right = peaks[ix].left;
  }
 }
```

```
}

static int Wavegen2(int length, int modulation, bool resume,
frame_t *fr1, frame_t *fr2)
{
 if (resume == false)
  SetSynth(length, modulation, fr1, fr2, wvoice);

 return Wavegen();
}

void Write4Bytes(FILE *f, int value)
{
 // Write 4 bytes to a file, least significant first
 int ix;

 for (ix = 0; ix < 4; ix++) {
  fputc(value & 0xff, f);
  value = value >> 8;
 }
}

static int WavegenFill2()
{
 // Pick up next wavegen commands from the queue
 // return: 0  output buffer has been filled
 // return: 1  input command queue is now empty
 intptr_t *q;
 int length;
 int result;
 int marker_type;
 static bool resume = false;
 static int echo_complete = 0;

 while (out_ptr < out_end) {
  if (WcmdqUsed() <= 0) {
   if (echo_complete > 0) {
```

```
    // continue to play silence until echo is completed
    resume = PlaySilence(echo_complete, resume);
    if (resume == true)
     return 0; // not yet finished
   }
   return 1; // queue empty, close sound channel
  }

  result = 0;
  q = wcmdq[wcmdq_head];
  length = q[1];

  switch (q[0] & 0xff)
  {
  case WCMD_PITCH:
   SetPitch(length, (unsigned char *)q[2], q[3] >> 16, q[3] &
0xffff);
   break;
  case WCMD_PAUSE:
   if (resume == false)
    echo_complete -= length;
   wdata.n_mix_wavefile = 0;
   wdata.amplitude_fmt = 100;
#ifdef INCLUDE_KLATT
   KlattReset(1);
#endif
   result = PlaySilence(length, resume);
   break;
  case WCMD_WAVE:
   echo_complete = echo_length;
   wdata.n_mix_wavefile = 0;
#ifdef INCLUDE_KLATT
   KlattReset(1);
#endif
   result = PlayWave(length, resume, (unsigned char *)q[2], q[3]
& 0xff, q[3] >> 8);
   break;
```

```
  case WCMD_WAVE2:
   // wave file to be played at the same time as synthesis
   wdata.mix_wave_amp = q[3] >> 8;
   wdata.mix_wave_scale = q[3] & 0xff;
   wdata.n_mix_wavefile = (length & 0xffff);
   wdata.mix_wavefile_max = (length >> 16) & 0xffff;
   if (wdata.mix_wave_scale == 0) {
    wdata.n_mix_wavefile *= 2;
    wdata.mix_wavefile_max *= 2;
   }
   wdata.mix_wavefile_ix = 0;
   wdata.mix_wavefile_offset = 0;
   wdata.mix_wavefile = (unsigned char *)q[2];
   break;
  case WCMD_SPECT2: // as WCMD_SPECT but stop any concurrent wave
file
   wdata.n_mix_wavefile = 0; // ... and drop through to
WCMD_SPECT case
  case WCMD_SPECT:
   echo_complete = echo_length;
   result = Wavegen2(length & 0xffff, q[1] >> 16, resume,
(frame_t *)q[2], (frame_t *)q[3]);
   break;
#ifdef INCLUDE_KLATT
  case WCMD_KLATT2: // as WCMD_SPECT but stop any concurrent wave
file
   wdata.n_mix_wavefile = 0; // ... and drop through to
WCMD_SPECT case
  case WCMD_KLATT:
   echo_complete = echo_length;
   result = Wavegen_Klatt2(length & 0xffff, resume, (frame_t
*)q[2], (frame_t *)q[3]);
   break;
#endif
  case WCMD_MARKER:
   marker_type = q[0] >> 8;
   MarkerEvent(marker_type, q[1], q[2], q[3], out_ptr);
```

```
   if (marker_type == 1) // word marker
    current_source_index = q[1] & 0xffffff;
   break;
  case WCMD_AMPLITUDE:
   SetAmplitude(length, (unsigned char *)q[2], q[3]);
   break;
  case WCMD_VOICE:
   WavegenSetVoice((voice_t *)q[2]);
   free((voice_t *)q[2]);
   break;
  case WCMD_EMBEDDED:
   SetEmbedded(q[1], q[2]);
   break;
  case WCMD_MBROLA_DATA:
   if (wvoice != NULL)
    result = MbrolaFill(length, resume, (general_amplitude *
wvoice->voicing)/64);
   break;
  case WCMD_FMT_AMPLITUDE:
   if ((wdata.amplitude_fmt = q[1]) == 0)
    wdata.amplitude_fmt = 100; // percentage, but value=0 means
100%
   break;
#if HAVE_SONIC_H
  case WCMD_SONIC_SPEED:
   sonicSpeed = (double)q[1] / 1024;
   break;
#endif
  }

  if (result == 0) {
   WcmdqIncHead();
   resume = false;
  } else
   resume = true;
 }
```

```
 return 0;
}

#if HAVE_SONIC_H
// Speed up the audio samples with libsonic.
static int SpeedUp(short *outbuf, int length_in, int length_out,
int end_of_text)
{
 if (length_in > 0) {
  if (sonicSpeedupStream == NULL)
   sonicSpeedupStream = sonicCreateStream(22050, 1);
  if (sonicGetSpeed(sonicSpeedupStream) != sonicSpeed)
   sonicSetSpeed(sonicSpeedupStream, sonicSpeed);

  sonicWriteShortToStream(sonicSpeedupStream, outbuf, length_in);
 }

 if (sonicSpeedupStream == NULL)
  return 0;

 if (end_of_text)
  sonicFlushStream(sonicSpeedupStream);
 return sonicReadShortFromStream(sonicSpeedupStream, outbuf,
length_out);
}
#endif

// Call WavegenFill2, and then speed up the output samples.
int WavegenFill(void)
{
 int finished;
 unsigned char *p_start;

 p_start = out_ptr;

 finished = WavegenFill2();
```

```
#if HAVE_SONIC_H
 if (sonicSpeed > 1.0) {
  int length;
  int max_length;

  max_length = (out_end - p_start);
  length =  2*SpeedUp((short *)p_start, (out_ptr-p_start)/2,
max_length/2, finished);
  out_ptr = p_start + length;

  if (length >= max_length)
   finished = 0; // there may be more data to flush
 }
#endif
 return finished;
}
```

Chapter 43

./src/libespeak-ng/mbrowrap.c

```
#include "config.h"

#if defined(_WIN32) || defined(_WIN64)
#include <windows.h>
#endif

#include "mbrowrap.h"

int (WINAPI *init_MBR)(char *voice_path);
void (WINAPI *close_MBR)(void);
void (WINAPI *reset_MBR)(void);
int (WINAPI *read_MBR)(short *buffer, int nb_samples);
int (WINAPI *write_MBR)(char *data);
int (WINAPI *flush_MBR)(void);
int (WINAPI *getFreq_MBR)(void);
void (WINAPI *setVolumeRatio_MBR)(float value);
char * (WINAPI *lastErrorStr_MBR)(char *buffer, int bufsize);
void (WINAPI *setNoError_MBR)(int no_error);

#if defined(_WIN32) || defined(_WIN64)

HINSTANCE hinstDllMBR = NULL;
```

```
BOOL load_MBR()
{
 if (hinstDllMBR != NULL)
  return TRUE;   // already loaded

 if ((hinstDllMBR = LoadLibraryA("mbrola.dll")) == 0)
  return FALSE;
 init_MBR = (void *)GetProcAddress(hinstDllMBR, "init_MBR");
 write_MBR = (void *)GetProcAddress(hinstDllMBR, "write_MBR");
 flush_MBR = (void *)GetProcAddress(hinstDllMBR, "flush_MBR");
 read_MBR = (void *)GetProcAddress(hinstDllMBR, "read_MBR");
 close_MBR = (void *)GetProcAddress(hinstDllMBR, "close_MBR");
 reset_MBR = (void *)GetProcAddress(hinstDllMBR, "reset_MBR");
 lastErrorStr_MBR = (void *)GetProcAddress(hinstDllMBR,
"lastErrorStr_MBR");
 setNoError_MBR = (void *)GetProcAddress(hinstDllMBR,
"setNoError_MBR");
 setVolumeRatio_MBR = (void *)GetProcAddress(hinstDllMBR,
"setVolumeRatio_MBR");
 return TRUE;
}

void unload_MBR()
{
 if (hinstDllMBR) {
  FreeLibrary(hinstDllMBR);
  hinstDllMBR = NULL;
 }
}

#else

#include <errno.h>
#include <fcntl.h>
#include <poll.h>
#include <signal.h>
```

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#include <espeak-ng/espeak_ng.h>

enum mbr_state {
 MBR_INACTIVE = 0,
 MBR_IDLE,
 MBR_NEWDATA,
 MBR_AUDIO,
 MBR_WEDGED
};

static enum mbr_state mbr_state;

static char *mbr_voice_path;
static int mbr_cmd_fd, mbr_audio_fd, mbr_error_fd, mbr_proc_stat;
static pid_t mbr_pid;
static int mbr_samplerate;
static float mbr_volume = 1.0;
static char mbr_errorbuf[160];

struct datablock {
 struct datablock *next;
 int done;
 int size;
 char buffer[1]; // 1 or more, dynamically allocated
};

static struct datablock *mbr_pending_data_head,
*mbr_pending_data_tail;
```

```
static void err(const char *errmsg, ...)
{
 va_list params;

 va_start(params, errmsg);
 vsnprintf(mbr_errorbuf, sizeof(mbr_errorbuf), errmsg, params);
 va_end(params);
 fprintf(stderr, "mbrowrap error: %s\n", mbr_errorbuf);
}

static int create_pipes(int p1[2], int p2[2], int p3[2])
{
 int error;

 if (pipe(p1) != -1) {
  if (pipe(p2) != -1) {
   if (pipe(p3) != -1)
    return 0;
   else
    error = errno;
   close(p2[0]);
   close(p2[1]);
  } else
   error = errno;
  close(p1[0]);
  close(p1[1]);
 } else
  error = errno;

 err("pipe(): %s", strerror(error));
 return -1;
}

static void close_pipes(int p1[2], int p2[2], int p3[2])
{
 close(p1[0]);
 close(p1[1]);
```

```
 close(p2[0]);
 close(p2[1]);
 close(p3[0]);
 close(p3[1]);
}

static int start_mbrola(const char *voice_path)
{
 int error, p_stdin[2], p_stdout[2], p_stderr[2];
 ssize_t written;
 char charbuf[20];

 if (mbr_state != MBR_INACTIVE) {
  err("mbrola init request when already initialized");
  return -1;
 }

 error = create_pipes(p_stdin, p_stdout, p_stderr);
 if (error)
  return -1;

 mbr_pid = fork();

 if (mbr_pid == -1) {
  error = errno;
  close_pipes(p_stdin, p_stdout, p_stderr);
  err("fork(): %s", strerror(error));
  return -1;
 }

 if (mbr_pid == 0) {
  int i;

  if (dup2(p_stdin[0], 0) == -1 ||
      dup2(p_stdout[1], 1) == -1 ||
      dup2(p_stderr[1], 2) == -1) {
   snprintf(mbr_errorbuf, sizeof(mbr_errorbuf),
```

```
          "dup2(): %s\n", strerror(errno));
   written = write(p_stderr[1], mbr_errorbuf,
strlen(mbr_errorbuf));
   (void)written;   // suppress 'variable not used' warning
   _exit(1);
  }

  for (i = p_stderr[1]; i > 2; i--)
   close(i);
  signal(SIGHUP, SIG_IGN);
  signal(SIGINT, SIG_IGN);
  signal(SIGQUIT, SIG_IGN);
  signal(SIGTERM, SIG_IGN);

  snprintf(charbuf, sizeof(charbuf), "%g", mbr_volume);
  execlp("mbrola", "mbrola", "-e", "-v", charbuf,
         voice_path, "-", "-.wav", (char *)NULL);
  /* if execution reaches this point then the exec() failed */
  snprintf(mbr_errorbuf, sizeof(mbr_errorbuf),
           "mbrola: %s\n", strerror(errno));
  written = write(2, mbr_errorbuf, strlen(mbr_errorbuf));
  (void)written;   // suppress 'variable not used' warning
  _exit(1);
 }

 snprintf(charbuf, sizeof(charbuf), "/proc/%d/stat", mbr_pid);
 mbr_proc_stat = open(charbuf, O_RDONLY);
 if (mbr_proc_stat == -1) {
  error = errno;
  close_pipes(p_stdin, p_stdout, p_stderr);
  waitpid(mbr_pid, NULL, 0);
  mbr_pid = 0;
  err("/proc is unaccessible: %s", strerror(error));
  return -1;
 }

 signal(SIGPIPE, SIG_IGN);
```

```
 if (fcntl(p_stdin[1], F_SETFL, O_NONBLOCK) == -1 ||
     fcntl(p_stdout[0], F_SETFL, O_NONBLOCK) == -1 ||
     fcntl(p_stderr[0], F_SETFL, O_NONBLOCK) == -1) {
  error = errno;
  close_pipes(p_stdin, p_stdout, p_stderr);
  waitpid(mbr_pid, NULL, 0);
  mbr_pid = 0;
  err("fcntl(): %s", strerror(error));
  return -1;
 }

 mbr_cmd_fd = p_stdin[1];
 mbr_audio_fd = p_stdout[0];
 mbr_error_fd = p_stderr[0];
 close(p_stdin[0]);
 close(p_stdout[1]);
 close(p_stderr[1]);

 mbr_state = MBR_IDLE;
 return 0;
}

static void stop_mbrola(void)
{
 if (mbr_state == MBR_INACTIVE)
  return;
 close(mbr_proc_stat);
 close(mbr_cmd_fd);
 close(mbr_audio_fd);
 close(mbr_error_fd);
 if (mbr_pid) {
  kill(mbr_pid, SIGTERM);
  waitpid(mbr_pid, NULL, 0);
  mbr_pid = 0;
 }
 mbr_state = MBR_INACTIVE;
```

```
}

static void free_pending_data(void)
{
 struct datablock *p, *head = mbr_pending_data_head;
 while (head) {
  p = head;
  head = head->next;
  free(p);
 }
 mbr_pending_data_head = NULL;
 mbr_pending_data_tail = NULL;
}

static int mbrola_died(void)
{
 pid_t pid;
 int status, len;
 const char *msg;
 char msgbuf[80];

 pid = waitpid(mbr_pid, &status, WNOHANG);
 if (!pid)
  msg = "mbrola closed stderr and did not exit";
 else if (pid != mbr_pid)
  msg = "waitpid() is confused";
 else {
  mbr_pid = 0;
  if (WIFSIGNALED(status)) {
   int sig = WTERMSIG(status);
   snprintf(msgbuf, sizeof(msgbuf),
            "mbrola died by signal %d", sig);
   msg = msgbuf;
  } else if (WIFEXITED(status)) {
   int exst = WEXITSTATUS(status);
   snprintf(msgbuf, sizeof(msgbuf),
            "mbrola exited with status %d", exst);
```

```
   msg = msgbuf;
  } else
   msg = "mbrola died and wait status is weird";
 }

 fprintf(stderr, "mbrowrap error: %s\n", msg);

 len = strlen(mbr_errorbuf);
 if (!len)
  snprintf(mbr_errorbuf, sizeof(mbr_errorbuf), "%s", msg);
 else
  snprintf(mbr_errorbuf + len, sizeof(mbr_errorbuf) - len,
           ", (%s)", msg);
 return -1;
}

static int mbrola_has_errors(void)
{
 int result;
 char buffer[256];
 char *buf_ptr, *lf;

 buf_ptr = buffer;
 for (;;) {
  result = read(mbr_error_fd, buf_ptr,
                sizeof(buffer) - (buf_ptr - buffer) - 1);
  if (result == -1) {
   if (errno == EAGAIN)
    return 0;
   err("read(error): %s", strerror(errno));
   return -1;
  }

  if (result == 0) {
   // EOF on stderr, assume mbrola died.
   return mbrola_died();
  }
```

```
  buf_ptr[result] = 0;

  for (; (lf = strchr(buf_ptr, '\n')); buf_ptr = lf + 1) {
   // inhibit the reset signal messages
   if (strncmp(buf_ptr, "Got a reset signal", 18) == 0 ||
       strncmp(buf_ptr, "Input Flush Signal", 18) == 0)
    continue;
   *lf = 0;
   fprintf(stderr, "mbrola: %s\n", buf_ptr);
   // is this the last line?
   if (lf == &buf_ptr[result - 1]) {
    snprintf(mbr_errorbuf, sizeof(mbr_errorbuf),
             "%s", buf_ptr);
    // don't consider this fatal at this point
    return 0;
   }
  }

  memmove(buffer, buf_ptr, result);
  buf_ptr = buffer + result;
 }
}

static int send_to_mbrola(const char *cmd)
{
 ssize_t result;
 int len;

 if (!mbr_pid)
  return -1;

 len = strlen(cmd);
 result = write(mbr_cmd_fd, cmd, len);

 if (result == -1) {
  int error = errno;
```

```
  if (error == EPIPE && mbrola_has_errors())
   return -1;
  else if (error == EAGAIN)
   result = 0;
  else {
   err("write(): %s", strerror(error));
   return -1;
  }
 }

 if (result != len) {
  struct datablock *data;
  data = (struct datablock *)malloc(sizeof(*data) + len -
result);
  if (data) {
   data->next = NULL;
   data->done = 0;
   data->size = len - result;
   memcpy(data->buffer, cmd + result, len - result);
   result = len;
   if (!mbr_pending_data_head)
    mbr_pending_data_head = data;
   else
    mbr_pending_data_tail->next = data;
   mbr_pending_data_tail = data;
  }
 }

 return result;
}

static int mbrola_is_idle(void)
{
 char *p;
 char buffer[20]; // looking for "12345 (mbrola) S" so 20 is
plenty
```

```
 // look in /proc to determine if mbrola is still running or
sleeping
 if (lseek(mbr_proc_stat, 0, SEEK_SET) != 0)
  return 0;
 if (read(mbr_proc_stat, buffer, sizeof(buffer)) !=
sizeof(buffer))
  return 0;
 p = (char *)memchr(buffer, ')', sizeof(buffer));
 if (!p || (unsigned)(p - buffer) >= sizeof(buffer) - 2)
  return 0;
 return p[1] == ' ' && p[2] == 'S';
}

static ssize_t receive_from_mbrola(void *buffer, size_t bufsize)
{
 int result, wait = 1;
 size_t cursize = 0;

 if (!mbr_pid)
  return -1;

 do {
  struct pollfd pollfd[3];
  nfds_t nfds = 0;
  int idle;

  pollfd[0].fd = mbr_audio_fd;
  pollfd[0].events = POLLIN;
  nfds++;

  pollfd[1].fd = mbr_error_fd;
  pollfd[1].events = POLLIN;
  nfds++;

  if (mbr_pending_data_head) {
   pollfd[2].fd = mbr_cmd_fd;
   pollfd[2].events = POLLOUT;
```

```
 nfds++;
}

idle = mbrola_is_idle();
result = poll(pollfd, nfds, idle ? 0 : wait);
if (result == -1) {
 err("poll(): %s", strerror(errno));
 return -1;
}
if (result == 0) {
 if (idle) {
  mbr_state = MBR_IDLE;
  break;
 } else {
  if (wait >= 5000 * (4-1)/4) {
   mbr_state = MBR_WEDGED;
   err("mbrola process is stalled");
   break;
  } else {
   wait *= 4;
   continue;
  }
 }
}
wait = 1;

if (pollfd[1].revents && mbrola_has_errors())
 return -1;

if (mbr_pending_data_head && pollfd[2].revents) {
 struct datablock *head = mbr_pending_data_head;
 char *data = head->buffer + head->done;
 int left = head->size - head->done;
 result = write(mbr_cmd_fd, data, left);
 if (result == -1) {
  int error = errno;
  if (error == EPIPE && mbrola_has_errors())
```

```
     return -1;
    err("write(): %s", strerror(error));
    return -1;
   }
   if (result != left)
    head->done += result;
   else {
    mbr_pending_data_head = head->next;
    free(head);
    if (!mbr_pending_data_head)
     mbr_pending_data_tail = NULL;
    else
     continue;
   }
  }

  if (pollfd[0].revents) {
   char *curpos = (char *)buffer + cursize;
   size_t space = bufsize - cursize;
   ssize_t obtained = read(mbr_audio_fd, curpos, space);
   if (obtained == -1) {
    err("read(): %s", strerror(errno));
    return -1;
   }
   cursize += obtained;
   mbr_state = MBR_AUDIO;
  }
 } while (cursize < bufsize);

 return cursize;
}

static int init_mbrola(char *voice_path)
{
 int error, result;
 unsigned char wavhdr[45];
```

```
 error = start_mbrola(voice_path);
 if (error)
  return -1;

 // Allow mbrola time to start when running on Windows Subsystem
for
 // Linux (WSL). Otherwise, the receive_from_mbrola call to read
the
 // wav header from mbrola will fail.
 usleep(100);

 result = send_to_mbrola("#\n");
 if (result != 2) {
  stop_mbrola();
  return -1;
 }

 // we should actually be getting only 44 bytes
 result = receive_from_mbrola(wavhdr, 45);
 if (result != 44) {
  if (result >= 0)
   err("unable to get .wav header from mbrola");
  stop_mbrola();
  return -1;
 }

 // parse wavhdr to get mbrola voice samplerate
 if (memcmp(wavhdr, "RIFF", 4) != 0 ||
     memcmp(wavhdr+8, "WAVEfmt ", 8) != 0) {
  err("mbrola did not return a .wav header");
  stop_mbrola();
  return -1;
 }
 mbr_samplerate = wavhdr[24] + (wavhdr[25]<<8) +
                  (wavhdr[26]<<16) + (wavhdr[27]<<24);

 // remember the voice path for setVolumeRatio_MBR()
```

```
 if (mbr_voice_path != voice_path) {
  free(mbr_voice_path);
  mbr_voice_path = strdup(voice_path);
 }

 return 0;
}

static void close_mbrola(void)
{
 stop_mbrola();
 free_pending_data();
 free(mbr_voice_path);
 mbr_voice_path = NULL;
 mbr_volume = 1.0;
}

static void reset_mbrola(void)
{
 int result, success = 1;
 char dummybuf[4096];

 if (mbr_state == MBR_IDLE)
  return;
 if (!mbr_pid)
  return;
 if (kill(mbr_pid, SIGUSR1) == -1)
  success = 0;
 free_pending_data();
 result = write(mbr_cmd_fd, "\n#\n", 3);
 if (result != 3)
  success = 0;
 do {
  result = read(mbr_audio_fd, dummybuf, sizeof(dummybuf));
 } while (result > 0);
 if (result != -1 || errno != EAGAIN)
  success = 0;
```

```
 if (!mbrola_has_errors() && success)
  mbr_state = MBR_IDLE;
}

static int read_mbrola(short *buffer, int nb_samples)
{
 int result = receive_from_mbrola(buffer, nb_samples * 2);
 if (result > 0)
  result /= 2;
 return result;
}

static int write_mbrola(char *data)
{
 mbr_state = MBR_NEWDATA;
 return send_to_mbrola(data);
}

static int flush_mbrola(void)
{
 return send_to_mbrola("\n#\n") == 3;
}

static int getFreq_mbrola(void)
{
 return mbr_samplerate;
}

static void setVolumeRatio_mbrola(float value)
{
 if (value == mbr_volume)
  return;
 mbr_volume = value;
 if (mbr_state != MBR_IDLE)
  return;
 /*
  * We have no choice but to kill and restart mbrola with
```

```
  * the new argument here.
  */
 stop_mbrola();
 init_MBR(mbr_voice_path);
}

static char *lastErrorStr_mbrola(char *buffer, int bufsize)
{
 if (mbr_pid)
  mbrola_has_errors();
 snprintf(buffer, bufsize, "%s", mbr_errorbuf);
 return buffer;
}

static void setNoError_mbrola(int no_error)
{
 (void)no_error; // unused
}

BOOL load_MBR(void)
{
 init_MBR = init_mbrola;
 close_MBR = close_mbrola;
 reset_MBR = reset_mbrola;
 read_MBR = read_mbrola;
 write_MBR = write_mbrola;
 flush_MBR = flush_mbrola;
 getFreq_MBR = getFreq_mbrola;
 setVolumeRatio_MBR = setVolumeRatio_mbrola;
 lastErrorStr_MBR = lastErrorStr_mbrola;
 setNoError_MBR = setNoError_mbrola;
 return 1;
}

void unload_MBR(void)
{
}
```

```
#endif
```

Chapter 44

./src/libespeak-ng/phoneme.c

```
#include "config.h"

#include <errno.h>
#include <stdint.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>

#include "phoneme.h"

phoneme_feature_t phoneme_feature_from_string(const char
*feature)
{
 if (!feature || strlen(feature) != 3)
  return inv;
 return (feature[0] << 16) | (feature[1] << 8) | feature[2];
}

espeak_ng_STATUS
phoneme_add_feature(PHONEME_TAB *phoneme,
                    phoneme_feature_t feature)
```

```
{
 if (!phoneme) return EINVAL;
 switch (feature)
 {
 // manner of articulation
 case nas:
  phoneme->type = phNASAL;
  break;
 case stp:
 case afr: // FIXME: eSpeak treats 'afr' as 'stp'.
  phoneme->type = phSTOP;
  break;
 case frc:
 case apr: // FIXME: eSpeak is using this for [h], with 'liquid'
used for [l] and [r].
  phoneme->type = phFRICATIVE;
  break;
 case flp: // FIXME: Why is eSpeak using a vstop (vcd + stp) for
this?
  phoneme->type = phVSTOP;
  break;
 case trl: // FIXME: 'trill' should be the type; 'liquid' should
be a flag (phoneme files specify both).
  phoneme->phflags |= phTRILL;
  break;
 case clk:
 case ejc:
 case imp:
 case lat:
  // Not supported by eSpeak.
  break;
 case vwl:
  phoneme->type = phVOWEL;
  break;
 case sib:
  phoneme->phflags |= phSIBILANT;
  break;
```

```
// place of articulation
case blb:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_BILABIAL << 16;
 break;
case lbd:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_LABIODENTAL << 16;
 break;
case dnt:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_DENTAL << 16;
 break;
case alv:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_ALVEOLAR << 16;
 break;
case rfx:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_RETROFLEX << 16;
 break;
case pla:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_PALATO_ALVEOLAR << 16;
 break;
case pal:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_PALATAL << 16;
 phoneme->phflags |= phPALATAL;
 break;
case vel:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_VELAR << 16;
 break;
case lbv:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_LABIO_VELAR << 16;
```

```
 break;
case uvl:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_UVULAR << 16;
 break;
case phr:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_PHARYNGEAL << 16;
 break;
case glt:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_GLOTTAL << 16;
 break;
case bld:
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_BILABIAL << 16;
 break;
case alp: // pla pzd
 phoneme->phflags &= ~phARTICULATION;
 phoneme->phflags |= phPLACE_PALATO_ALVEOLAR << 16;
 phoneme->phflags |= phPALATAL;
 break;
// voice
case vcd:
 phoneme->phflags |= phVOICED;
 break;
case vls:
 phoneme->phflags |= phVOICELESS;
 break;
// vowel height
case hgh:
case smh:
case umd:
case mid:
case lmd:
case sml:
case low:
```

```
 // Not supported by eSpeak.
 break;
// vowel backness
case fnt:
case cnt:
case bck:
 // Not supported by eSpeak.
 break;
// rounding
case unr:
case rnd:
 // Not supported by eSpeak.
 break;
// articulation
case lgl:
case idt:
case apc:
case lmn:
 // Not supported by eSpeak.
 break;
// air flow
case egs:
case igs:
 // Not supported by eSpeak.
 break;
// phonation
case brv:
case slv:
case stv:
case crv:
case glc:
 // Not supported by eSpeak.
 break;
// rounding and labialization
case ptr:
case cmp:
case mrd:
```

```
case lrd:
 // Not supported by eSpeak.
 break;
// syllabicity
case syl:
 // Not supported by eSpeak.
 break;
case nsy:
 phoneme->phflags |= phNONSYLLABIC;
 break;
// consonant release
case asp:
case nrs:
case lrs:
case unx:
 // Not supported by eSpeak.
 break;
// coarticulation
case pzd:
 phoneme->phflags |= phPALATAL;
 break;
case vzd:
case fzd:
case nzd:
case rzd:
 // Not supported by eSpeak.
 break;
// tongue root
case atr:
case rtr:
 // Not supported by eSpeak.
 break;
// fortis and lenis
case fts:
case lns:
 // Not supported by eSpeak.
 break;
```

```
  // length
  case est:
  case hlg:
   // Not supported by eSpeak.
   break;
  case elg: // FIXME: Should be longer than 'lng'.
  case lng:
   phoneme->phflags |= phLONG;
   break;
  // invalid phoneme feature
  default:
   return ENS_UNKNOWN_PHONEME_FEATURE;
  }
  return ENS_OK;
 }
```

Chapter 45

./src/libespeak-ng/klatt.c

```
// See URL: ftp://svr-
ftp.eng.cam.ac.uk/pub/comp.speech/synthesis/klatt.3.04.tar.gz

#include "config.h"

#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>

#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "klatt.h"

extern unsigned char *out_ptr;
extern unsigned char *out_start;
extern unsigned char *out_end;
```

```
extern WGEN_DATA wdata;
static int nsamples;
static int sample_count;

#ifdef _MSC_VER
#define getrandom(min, max)
((rand()%(int)(((max)+1)-(min)))+(min))
#else
#define getrandom(min, max)
((rand()%(long)(((max)+1)-(min)))+(min))
#endif

// function prototypes for functions private to this file

static void flutter(klatt_frame_ptr);
static double sampled_source(int);
static double impulsive_source(void);
static double natural_source(void);
static void pitch_synch_par_reset(klatt_frame_ptr);
static double gen_noise(double);
static double DBtoLIN(long);
static void frame_init(klatt_frame_ptr);
static void setabc(long, long, resonator_ptr);
static void setzeroabc(long, long, resonator_ptr);

static klatt_frame_t kt_frame;
static klatt_global_t kt_globals;

#define NUMBER_OF_SAMPLES 100

static int scale_wav_tab[] = { 45, 38, 45, 45, 55 }; // scale
output from different voicing sources

// For testing, this can be overwritten in KlattInit()
static short natural_samples2[256] = {
  2583,  2516,  2450,  2384,  2319,  2254,  2191,  2127,
  2067,  2005,  1946,  1890,  1832,  1779,  1726,  1675,
```

```
  1626,  1579,  1533,  1491,  1449,  1409,  1372,  1336,
  1302,  1271,  1239,  1211,  1184,  1158,  1134,  1111,
  1089,  1069,  1049,  1031,  1013,   996,   980,   965,
   950,   936,   921,   909,   895,   881,   869,   855,
   843,   830,   818,   804,   792,   779,   766,   754,
   740,   728,   715,   702,   689,   676,   663,   651,
   637,   626,   612,   601,   588,   576,   564,   552,
   540,   530,   517,   507,   496,   485,   475,   464,
   454,   443,   434,   424,   414,   404,   394,   385,
   375,   366,   355,   347,   336,   328,   317,   308,
   299,   288,   280,   269,   260,   250,   240,   231,
   220,   212,   200,   192,   181,   172,   161,   152,
   142,   133,   123,   113,   105,    94,    86,    76,
    67,    57,    49,    39,    30,    22,    11,     4,
    -5,   -14,   -23,   -32,   -41,   -50,   -60,   -69,
   -78,   -87,   -96,  -107,  -115,  -126,  -134,  -144,
  -154,  -164,  -174,  -183,  -193,  -203,  -213,  -222,
  -233,  -242,  -252,  -262,  -271,  -281,  -291,  -301,
  -310,  -320,  -330,  -339,  -349,  -357,  -368,  -377,
  -387,  -397,  -406,  -417,  -426,  -436,  -446,  -456,
  -467,  -477,  -487,  -499,  -509,  -521,  -532,  -543,
  -555,  -567,  -579,  -591,  -603,  -616,  -628,  -641,
  -653,  -666,  -679,  -692,  -705,  -717,  -732,  -743,
  -758,  -769,  -783,  -795,  -808,  -820,  -834,  -845,
  -860,  -872,  -885,  -898,  -911,  -926,  -939,  -955,
  -968,  -986,  -999, -1018, -1034, -1054, -1072, -1094,
 -1115, -1138, -1162, -1188, -1215, -1244, -1274, -1307,
 -1340, -1377, -1415, -1453, -1496, -1538, -1584, -1631,
 -1680, -1732, -1783, -1839, -1894, -1952, -2010, -2072,
 -2133, -2196, -2260, -2325, -2390, -2456, -2522, -2589,
};
static short natural_samples[100] = {
  -310,  -400,   530,   356,   224,    89,   23,  -10, -58, -16,
461,  599,  536,   701,   770,
   605,   497,   461,   560,   404,   110,  224,  131, 104, -97,
155,  278, -154, -1165,
  -598,   737,   125,  -592,    41,    11, -247,  -10,  65,  92,
```

```
80, -304,   71,   167,    -1, 122,
   233,   161,   -43,   278,   479,   485,  407,  266, 650, 134,
80,  236,   68,   260,   269, 179,
    53,   140,   275,   293,   296,   104,  257,  152, 311, 182,
263,  245,  125,   314,   140, 44,
   203,   230,  -235,  -286,    23,   107,   92,  -91,  38, 464,
443,  176,   98,  -784, -2449,
 -1891, -1045, -1600, -1462, -1384, -1261, -949, -730
};

   function RESONATOR

   This is a generic resonator function. Internal memory for the
resonator
   is stored in the globals structure.

static double resonator(resonator_ptr r, double input)
{
 double x;

 x = (double)((double)r->a * (double)input + (double)r->b *
(double)r->p1 + (double)r->c * (double)r->p2);
 r->p2 = (double)r->p1;
 r->p1 = (double)x;

 return (double)x;
}

static double resonator2(resonator_ptr r, double input)
{
 double x;

 x = (double)((double)r->a * (double)input + (double)r->b *
(double)r->p1 + (double)r->c * (double)r->p2);
 r->p2 = (double)r->p1;
 r->p1 = (double)x;
```

```
 r->a += r->a_inc;
 r->b += r->b_inc;
 r->c += r->c_inc;
 return (double)x;
}

static double antiresonator2(resonator_ptr r, double input)
{
 register double x = (double)r->a * (double)input + (double)r->b
* (double)r->p1 + (double)r->c * (double)r->p2;
 r->p2 = (double)r->p1;
 r->p1 = (double)input;

 r->a += r->a_inc;
 r->b += r->b_inc;
 r->c += r->c_inc;
 return (double)x;
}

   function FLUTTER

   This function adds F0 flutter, as specified in:

   "Analysis, synthesis and perception of voice quality
variations among
   female and male talkers" D.H. Klatt and L.C. Klatt JASA 87(2)
February 1990.

   Flutter is added by applying a quasi-random element
constructed from three
   slowly varying sine waves.

static void flutter(klatt_frame_ptr frame)
{
 static int time_count;
 double delta_f0;
 double fla, flb, flc, fld, fle;
```

```
 fla = (double)kt_globals.f0_flutter / 50;
 flb = (double)kt_globals.original_f0 / 100;
 flc = sin(M_PI*12.7*time_count); // because we are calling
flutter() more frequently, every 2.9mS
 fld = sin(M_PI*7.1*time_count);
 fle = sin(M_PI*4.7*time_count);
 delta_f0 =  fla * flb * (flc + fld + fle) * 10;
 frame->F0hz10 = frame->F0hz10 + (long)delta_f0;
 time_count++;
}

   function SAMPLED_SOURCE

   Allows the use of a glottal excitation waveform sampled from a
real
   voice.

static double sampled_source(int source_num)
{
 int itemp;
 double ftemp;
 double result;
 double diff_value;
 int current_value;
 int next_value;
 double temp_diff;
 short *samples;

 if (source_num == 0) {
  samples = natural_samples;
  kt_globals.num_samples = 100;
 } else {
  samples = natural_samples2;
  kt_globals.num_samples = 256;
 }
```

```
 if (kt_globals.T0 != 0) {
  ftemp = (double)kt_globals.nper;
  ftemp = ftemp / kt_globals.T0;
  ftemp = ftemp * kt_globals.num_samples;
  itemp = (int)ftemp;

  temp_diff = ftemp - (double)itemp;

  current_value = samples[itemp];
  next_value = samples[itemp+1];

  diff_value = (double)next_value - (double)current_value;
  diff_value = diff_value * temp_diff;

  result = samples[itemp] + diff_value;
  result = result * kt_globals.sample_factor;
 } else
  result = 0;
 return result;
}

   function PARWAVE

   Converts synthesis parameters to a waveform.

static int parwave(klatt_frame_ptr frame)
{
 double temp;
 int value;
 double outbypas;
 double out;
 long n4;
 double frics;
 double glotout;
 double aspiration;
 double casc_next_in;
 double par_glotout;
```

```
 static double noise;
 static double voice;
 static double vlast;
 static double glotlast;
 static double sourc;
 int ix;

 flutter(frame); // add f0 flutter

 // MAIN LOOP, for each output sample of current frame:

 for (kt_globals.ns = 0; kt_globals.ns < kt_globals.nspfr;
kt_globals.ns++) {
  // Get low-passed random number for aspiration and frication
noise
  noise = gen_noise(noise);

  // Amplitude modulate noise (reduce noise amplitude during
  // second half of glottal period) if voicing simultaneously
present.

  if (kt_globals.nper > kt_globals.nmod)
   noise *= (double)0.5;

  // Compute frication noise
  frics = kt_globals.amp_frica * noise;

  // Compute voicing waveform. Run glottal source simulation at 4
  // times normal sample rate to minimize quantization noise in
  // period of female voice.

  for (n4 = 0; n4 < 4; n4++) {
   switch (kt_globals.glsource)
   {
   case IMPULSIVE:
    voice = impulsive_source();
    break;
```

```
    case NATURAL:
     voice = natural_source();
     break;
    case SAMPLED:
     voice = sampled_source(0);
     break;
    case SAMPLED2:
     voice = sampled_source(1);
     break;
    }

    // Reset period when counter 'nper' reaches T0
    if (kt_globals.nper >= kt_globals.T0) {
     kt_globals.nper = 0;
     pitch_synch_par_reset(frame);
    }

    // Low-pass filter voicing waveform before downsampling from
4*samrate
    // to samrate samples/sec.  Resonator f=.09*samrate,
bw=.06*samrate

    voice = resonator(&(kt_globals.rsn[RLP]), voice);

    // Increment counter that keeps track of 4*samrate samples per
sec
    kt_globals.nper++;
   }

  // Tilt spectrum of voicing source down by soft low-pass
filtering, amount
  // of tilt determined by TLTdb

  voice = (voice * kt_globals.onemd) + (vlast *
kt_globals.decay);
  vlast = voice;
```

```
  // Add breathiness during glottal open phase. Amount of
breathiness
  // determined by parameter Aturb Use nrand rather than noise
because
  // noise is low-passed.

  if (kt_globals.nper < kt_globals.nopen)
   voice += kt_globals.amp_breth * kt_globals.nrand;

  // Set amplitude of voicing
  glotout = kt_globals.amp_voice * voice;
  par_glotout = kt_globals.par_amp_voice * voice;

  // Compute aspiration amplitude and add to voicing source
  aspiration = kt_globals.amp_aspir * noise;
  glotout += aspiration;

  par_glotout += aspiration;

  // Cascade vocal tract, excited by laryngeal sources.
  // Nasal antiresonator, then formants FNP, F5, F4, F3, F2, F1

  out = 0;
  if (kt_globals.synthesis_model != ALL_PARALLEL) {
   casc_next_in = antiresonator2(&(kt_globals.rsn[Rnz]),
glotout);
   casc_next_in = resonator(&(kt_globals.rsn[Rnpc]),
casc_next_in);
   casc_next_in = resonator(&(kt_globals.rsn[R8c]),
casc_next_in);
   casc_next_in = resonator(&(kt_globals.rsn[R7c]),
casc_next_in);
   casc_next_in = resonator(&(kt_globals.rsn[R6c]),
casc_next_in);
   casc_next_in = resonator2(&(kt_globals.rsn[R5c]),
casc_next_in);
   casc_next_in = resonator2(&(kt_globals.rsn[R4c]),
```

```
casc_next_in);
   casc_next_in = resonator2(&(kt_globals.rsn[R3c]),
casc_next_in);
   casc_next_in = resonator2(&(kt_globals.rsn[R2c]),
casc_next_in);
   out = resonator2(&(kt_globals.rsn[R1c]), casc_next_in);
  }

  // Excite parallel F1 and FNP by voicing waveform
  sourc = par_glotout; // Source is voicing plus aspiration

  // Standard parallel vocal tract Formants F6,F5,F4,F3,F2,
  // outputs added with alternating sign. Sound source for other
  // parallel resonators is frication plus first difference of
  // voicing waveform.

  out += resonator(&(kt_globals.rsn[R1p]), sourc);
  out += resonator(&(kt_globals.rsn[Rnpp]), sourc);

  sourc = frics + par_glotout - glotlast;
  glotlast = par_glotout;

  for (ix = R2p; ix <= R6p; ix++)
   out = resonator(&(kt_globals.rsn[ix]), sourc) - out;

  outbypas = kt_globals.amp_bypas * sourc;

  out = outbypas - out;

  out = resonator(&(kt_globals.rsn[Rout]), out);
  temp = (int)(out * wdata.amplitude * kt_globals.amp_gain0); //
Convert back to integer

  // mix with a recorded WAV if required for this phoneme
  signed char c;
  int sample;
```

```
  if (wdata.mix_wavefile_ix < wdata.n_mix_wavefile) {
   if (wdata.mix_wave_scale == 0) {
    // a 16 bit sample
    c = wdata.mix_wavefile[wdata.mix_wavefile_ix+1];
    sample = wdata.mix_wavefile[wdata.mix_wavefile_ix] + (c *
256);
    wdata.mix_wavefile_ix += 2;
   } else {
    // a 8 bit sample, scaled
    sample = (signed
char)wdata.mix_wavefile[wdata.mix_wavefile_ix++] *
wdata.mix_wave_scale;
   }
   int z2 = sample * wdata.amplitude_v / 1024;
   z2 = (z2 * wdata.mix_wave_amp)/40;
   temp += z2;
  }

  // if fadeout is set, fade to zero over 64 samples, to avoid
clicks at end of synthesis
  if (kt_globals.fadeout > 0) {
   kt_globals.fadeout--;
   temp = (temp * kt_globals.fadeout) / 64;
  }

  value = (int)temp + ((echo_buf[echo_tail++]*echo_amp) >> 8);
  if (echo_tail >= N_ECHO_BUF)
   echo_tail = 0;

  if (value < -32768)
   value = -32768;

  if (value > 32767)
   value =  32767;

  *out_ptr++ = value;
  *out_ptr++ = value >> 8;
```

```
  echo_buf[echo_head++] = value;
  if (echo_head >= N_ECHO_BUF)
   echo_head = 0;

  sample_count++;
  if (out_ptr + 2 > out_end)
   return 1;
 }
 return 0;
}

void KlattReset(int control)
{
 int r_ix;

 if (control == 2) {
  // Full reset
  kt_globals.FLPhz = (950 * kt_globals.samrate) / 10000;
  kt_globals.BLPhz = (630 * kt_globals.samrate) / 10000;
  kt_globals.minus_pi_t = -M_PI / kt_globals.samrate;
  kt_globals.two_pi_t = -2.0 * kt_globals.minus_pi_t;
  setabc(kt_globals.FLPhz, kt_globals.BLPhz,
&(kt_globals.rsn[RLP]));
 }

 if (control > 0) {
  kt_globals.nper = 0;
  kt_globals.T0 = 0;
  kt_globals.nopen = 0;
  kt_globals.nmod = 0;

  for (r_ix = RGL; r_ix < N_RSN; r_ix++) {
   kt_globals.rsn[r_ix].p1 = 0;
   kt_globals.rsn[r_ix].p2 = 0;
  }
 }
```

```
 for (r_ix = 0; r_ix <= R6p; r_ix++) {
  kt_globals.rsn[r_ix].p1 = 0;
  kt_globals.rsn[r_ix].p2 = 0;
 }
}
```

   function FRAME_INIT

   Use parameters from the input frame to set up resonator coefficients.

```
static void frame_init(klatt_frame_ptr frame)
{
 double amp_par[7];
 static double amp_par_factor[7] = { 0.6, 0.4, 0.15, 0.06, 0.04,
0.022, 0.03 };
 long Gain0_tmp;
 int ix;

 kt_globals.original_f0 = frame->F0hz10 / 10;

 frame->AVdb_tmp  = frame->AVdb - 7;
 if (frame->AVdb_tmp < 0)
  frame->AVdb_tmp = 0;

 kt_globals.amp_aspir = DBtoLIN(frame->ASP) * 0.05;
 kt_globals.amp_frica = DBtoLIN(frame->AF) * 0.25;
 kt_globals.par_amp_voice = DBtoLIN(frame->AVpdb);
 kt_globals.amp_bypas = DBtoLIN(frame->AB) * 0.05;

 for (ix = 0; ix <= 6; ix++) {
  // parallel amplitudes F1 to F6, and parallel nasal pole
  amp_par[ix] = DBtoLIN(frame->Ap[ix]) * amp_par_factor[ix];
 }

 Gain0_tmp = frame->Gain0 - 3;
```

```
 if (Gain0_tmp <= 0)
  Gain0_tmp = 57;
 kt_globals.amp_gain0 = DBtoLIN(Gain0_tmp) /
kt_globals.scale_wav;

 // Set coefficients of variable cascade resonators
 for (ix = 1; ix <= 9; ix++) {
  // formants 1 to 8, plus nasal pole
  setabc(frame->Fhz[ix], frame->Bhz[ix], &(kt_globals.rsn[ix]));

  if (ix <= 5) {
   setabc(frame->Fhz_next[ix], frame->Bhz_next[ix],
&(kt_globals.rsn_next[ix]));

   kt_globals.rsn[ix].a_inc = (kt_globals.rsn_next[ix].a -
kt_globals.rsn[ix].a) / 64.0;
   kt_globals.rsn[ix].b_inc = (kt_globals.rsn_next[ix].b -
kt_globals.rsn[ix].b) / 64.0;
   kt_globals.rsn[ix].c_inc = (kt_globals.rsn_next[ix].c -
kt_globals.rsn[ix].c) / 64.0;
  }
 }

 // nasal zero anti-resonator
 setzeroabc(frame->Fhz[F_NZ], frame->Bhz[F_NZ],
&(kt_globals.rsn[Rnz]));
 setzeroabc(frame->Fhz_next[F_NZ], frame->Bhz_next[F_NZ],
&(kt_globals.rsn_next[Rnz]));
 kt_globals.rsn[F_NZ].a_inc = (kt_globals.rsn_next[F_NZ].a -
kt_globals.rsn[F_NZ].a) / 64.0;
 kt_globals.rsn[F_NZ].b_inc = (kt_globals.rsn_next[F_NZ].b -
kt_globals.rsn[F_NZ].b) / 64.0;
 kt_globals.rsn[F_NZ].c_inc = (kt_globals.rsn_next[F_NZ].c -
kt_globals.rsn[F_NZ].c) / 64.0;

 // Set coefficients of parallel resonators, and amplitude of
outputs
```

```
 for (ix = 0; ix <= 6; ix++) {
  setabc(frame->Fhz[ix], frame->Bphz[ix],
&(kt_globals.rsn[Rparallel+ix]));
  kt_globals.rsn[Rparallel+ix].a *= amp_par[ix];
 }

 // output low-pass filter

 setabc((long)0.0, (long)(kt_globals.samrate/2),
&(kt_globals.rsn[Rout]));
}

   function IMPULSIVE_SOURCE

   Generate a low pass filtered train of impulses as an
approximation of
   a natural excitation waveform. Low-pass filter the
differentiated impulse
   with a critically-damped second-order filter, time constant
proportional
   to Kopen.

static double impulsive_source()
{
 static double doublet[] = { 0.0, 13000000.0, -13000000.0 };
 static double vwave;

 if (kt_globals.nper < 3)
  vwave = doublet[kt_globals.nper];
 else
  vwave = 0.0;

 return resonator(&(kt_globals.rsn[RGL]), vwave);
}

   function NATURAL_SOURCE
```

Vwave is the differentiated glottal flow waveform, there is a weak
spectral zero around 800 Hz, magic constants a,b reset pitch synchronously.

```
static double natural_source()
{
 double lgtemp;
 static double vwave;

 if (kt_globals.nper < kt_globals.nopen) {
  kt_globals.pulse_shape_a -= kt_globals.pulse_shape_b;
  vwave += kt_globals.pulse_shape_a;
  lgtemp = vwave * 0.028;

  return lgtemp;
 }
 vwave = 0.0;
 return 0.0;
}
```

function PITCH_SYNC_PAR_RESET

Reset selected parameters pitch-synchronously.

Constant B0 controls shape of glottal pulse as a function
of desired duration of open phase N0
(Note that N0 is specified in terms of 40,000 samples/sec of speech)

Assume voicing waveform V(t) has form: k1 t**2 - k2 t**3

```
   If the radiation characterivative, a temporal derivative
   is folded in, and we go from continuous time to discrete
   integers n:  dV/dt = vwave[n]
                      = sum over i=1,2,...,n of { a - (i * b) }
```

```
                    = a n  -  b/2 n**2

   where the  constants a and b control the detailed shape
   and amplitude of the voicing waveform over the open
   potion of the voicing cycle "nopen".

   Let integral of dV/dt have no net dc flow --> a = (b * nopen)
/ 3

   Let maximum of dUg(n)/dn be constant --> b = gain / (nopen *
nopen)
   meaning as nopen gets bigger, V has bigger peak proportional
to n

   Thus, to generate the table below for 40 <= nopen <= 263:

   B0[nopen - 40] = 1920000 / (nopen * nopen)

static void pitch_synch_par_reset(klatt_frame_ptr frame)
{
 long temp;
 double temp1;
 static long skew;
 static short B0[224] = {
  1200, 1142, 1088, 1038, 991, 948, 907, 869, 833, 799, 768, 738,
710, 683, 658,
   634,  612,  590,  570, 551, 533, 515, 499, 483, 468, 454, 440,
427, 415, 403,
   391,  380,  370,  360, 350, 341, 332, 323, 315, 307, 300, 292,
285, 278, 272,
   265,  259,  253,  247, 242, 237, 231, 226, 221, 217, 212, 208,
204, 199, 195,
   192,  188,  184,  180, 177, 174, 170, 167, 164, 161, 158, 155,
153, 150, 147,
   145,  142,  140,  137, 135, 133, 131, 128, 126, 124, 122, 120,
119, 117, 115,
   113,  111,  110,  108, 106, 105, 103, 102, 100,  99,  97,  96,
```

```
95,  93,  92, 91, 90,
    88,   87,   86,   85,  84,  83,  82,  80,  79,  78,  77,  76,
75,  75,  74, 73, 72, 71,
    70,   69,   68,   68,  67,  66,  65,  64,  64,  63,  62,  61,
61,  60,  59, 59, 58, 57,
    57,   56,   56,   55,  55,  54,  54,  53,  53,  52,  52,  51,
51,  50,  50, 49, 49, 48, 48,
    47,   47,   46,   46,  45,  45,  44,  44,  43,  43,  42,  42,
41,  41,  41, 41, 40, 40,
    39,   39,   38,   38,  38,  38,  37,  37,  36,  36,  36,  36,
35,  35,  35, 35, 34, 34, 33,
    33,   33,   33,   32,  32,  32,  32,  31,  31,  31,  31,  30,
30,  30,  30, 29, 29, 29, 29,
    28,   28,   28,   28,  27,  27
 };

 if (frame->F0hz10 > 0) {
  // T0 is 4* the number of samples in one pitch period

  kt_globals.T0 = (40 * kt_globals.samrate) / frame->F0hz10;

  kt_globals.amp_voice = DBtoLIN(frame->AVdb_tmp);

  // Duration of period before amplitude modulation

  kt_globals.nmod = kt_globals.T0;
  if (frame->AVdb_tmp > 0)
   kt_globals.nmod >>= 1;

  // Breathiness of voicing waveform

  kt_globals.amp_breth = DBtoLIN(frame->Aturb) * 0.1;

  // Set open phase of glottal period where  40 <= open phase <=
263

  kt_globals.nopen = 4 * frame->Kopen;
```

```
  if ((kt_globals.glsource == IMPULSIVE) && (kt_globals.nopen >
263))
   kt_globals.nopen = 263;

  if (kt_globals.nopen >= (kt_globals.T0-1))
   kt_globals.nopen = kt_globals.T0 - 2;

  if (kt_globals.nopen < 40) {
   // F0 max = 1000 Hz
   kt_globals.nopen = 40;
  }

  // Reset a & b, which determine shape of "natural" glottal
waveform

  kt_globals.pulse_shape_b = B0[kt_globals.nopen-40];
  kt_globals.pulse_shape_a = (kt_globals.pulse_shape_b *
kt_globals.nopen) * 0.333;

  // Reset width of "impulsive" glottal pulse

  temp = kt_globals.samrate / kt_globals.nopen;

  setabc((long)0, temp, &(kt_globals.rsn[RGL]));

  // Make gain at F1 about constant

  temp1 = kt_globals.nopen *.00833;
  kt_globals.rsn[RGL].a *= temp1 * temp1;

  // Truncate skewness so as not to exceed duration of closed
phase
  // of glottal period.

  temp = kt_globals.T0 - kt_globals.nopen;
  if (frame->Kskew > temp)
```

```
   frame->Kskew = temp;
  if (skew >= 0)
   skew = frame->Kskew;
  else
   skew = -frame->Kskew;

  // Add skewness to closed portion of voicing period
  kt_globals.T0 = kt_globals.T0 + skew;
  skew = -skew;
 } else {
  kt_globals.T0 = 4; // Default for f0 undefined
  kt_globals.amp_voice = 0.0;
  kt_globals.nmod = kt_globals.T0;
  kt_globals.amp_breth = 0.0;
  kt_globals.pulse_shape_a = 0.0;
  kt_globals.pulse_shape_b = 0.0;
 }

 // Reset these pars pitch synchronously or at update rate if
f0=0

 if ((kt_globals.T0 != 4) || (kt_globals.ns == 0)) {
  // Set one-pole low-pass filter that tilts glottal source

  kt_globals.decay = (0.033 * frame->TLTdb);

  if (kt_globals.decay > 0.0)
   kt_globals.onemd = 1.0 - kt_globals.decay;
  else
   kt_globals.onemd = 1.0;
 }
}

   function SETABC

   Convert formant freqencies and bandwidth into resonator
difference
```

   equation constants.

```
static void setabc(long int f, long int bw, resonator_ptr rp)
{
 double r;
 double arg;

 // Let r  =  exp(-pi bw t)
 arg = kt_globals.minus_pi_t * bw;
 r = exp(arg);

 // Let c  =  -r**2
 rp->c = -(r * r);

 // Let b = r * 2*cos(2 pi f t)
 arg = kt_globals.two_pi_t * f;
 rp->b = r * cos(arg) * 2.0;

 // Let a = 1.0 - b - c
 rp->a = 1.0 - rp->b - rp->c;
}
```

   function SETZEROABC

   Convert formant freqencies and bandwidth into anti-resonator difference
   equation constants.

```
static void setzeroabc(long int f, long int bw, resonator_ptr rp)
{
 double r;
 double arg;

 f = -f;

 // First compute ordinary resonator coefficients
 // Let r  =  exp(-pi bw t)
```

```
 arg = kt_globals.minus_pi_t * bw;
 r = exp(arg);

 // Let c  =  -r**2
 rp->c = -(r * r);

 // Let b = r * 2*cos(2 pi f t)
 arg = kt_globals.two_pi_t * f;
 rp->b = r * cos(arg) * 2.;

 // Let a = 1.0 - b - c
 rp->a = 1.0 - rp->b - rp->c;

 // Now convert to antiresonator coefficients (a'=1/a, b'=b/a,
c'=c/a)

 // If f == 0 then rp->a gets set to 0 which makes a'=1/a set a',
b' and c' to
 // INF, causing an audible sound spike when triggered (e.g.
apiration with the
 // nasal register set to f=0, bw=0).
 if (rp->a != 0) {
  // Now convert to antiresonator coefficients (a'=1/a, b'=b/a,
c'=c/a)
  rp->a = 1.0 / rp->a;
  rp->c *= -rp->a;
  rp->b *= -rp->a;
 }
}
```

function GEN_NOISE

Random number generator (return a number between -8191 and +8191)
Noise spectrum is tilted down by soft low-pass filter having a pole near
the origin in the z-plane, i.e. output = input + (0.75 *

```
lastoutput)

static double gen_noise(double noise)
{
 long temp;
 static double nlast;

 temp = (long)getrandom(-8191, 8191);
 kt_globals.nrand = (long)temp;

 noise = kt_globals.nrand + (0.75 * nlast);
 nlast = noise;

 return noise;
}

   function DBTOLIN

   Convert from decibels to a linear scale factor

   Conversion table, db to linear, 87 dB --> 32767
                                86 dB --> 29491 (1 dB down =
0.5**1/6)
                                 ...
                                81 dB --> 16384 (6 dB down = 0.5)
                                 ...
                                 0 dB -->     0

   The just noticeable difference for a change in intensity of a
vowel
   is approximately 1 dB.  Thus all amplitudes are quantized to 1
dB
   steps.

static double DBtoLIN(long dB)
{
 static short amptable[88] = {
```

```
     0,      0,      0,      0,      0,      0,      0,     0,      0,
0,   0,   0,  0, 6, 7,
     8,      9,     10,     11,     13,     14,     16,    18,     20,
22,  25,  28, 32,
     35,     40,     45,     51,     57,     64,     71,    80,     90,
101, 114, 128,
    142,    159,    179,    202,    227,    256,    284,   318,    359,
405,
    455,    512,    568,    638,    719,    881,    911,  1024,   1137,
1276,
   1438,   1622,   1823,   2048,   2273,   2552,   2875,  3244,   3645,
   4096,   4547,   5104,   5751,   6488,   7291,   8192,  9093,  10207,
  11502, 12976, 14582, 16384, 18350, 20644, 23429,
  26214, 29491, 32767
 };

 if ((dB < 0) || (dB > 87))
  return 0;

 return (double)(amptable[dB]) * 0.001;
}

extern voice_t *wvoice;
static klatt_peaks_t peaks[N_PEAKS];
static int end_wave;
static int klattp[N_KLATTP];
static double klattp1[N_KLATTP];
static double klattp_inc[N_KLATTP];

static int Wavegen_Klatt(int resume)
{
 int pk;
 int x;
 int ix;
 int fade;

 if (resume == 0)
```

```
  sample_count = 0;

 while (sample_count < nsamples) {
  kt_frame.F0hz10 = (wdata.pitch * 10) / 4096;

  // formants F6,F7,F8 are fixed values for cascade resonators,
set in KlattInit()
  // but F6 is used for parallel resonator
  // F0 is used for the nasal zero
  for (ix = 0; ix < 6; ix++) {
   kt_frame.Fhz[ix] = peaks[ix].freq;
   if (ix < 4)
    kt_frame.Bhz[ix] = peaks[ix].bw;
  }
  for (ix = 1; ix < 7; ix++)
   kt_frame.Ap[ix] = peaks[ix].ap;

  kt_frame.AVdb = klattp[KLATT_AV];
  kt_frame.AVpdb = klattp[KLATT_AVp];
  kt_frame.AF = klattp[KLATT_Fric];
  kt_frame.AB = klattp[KLATT_FricBP];
  kt_frame.ASP = klattp[KLATT_Aspr];
  kt_frame.Aturb = klattp[KLATT_Turb];
  kt_frame.Kskew = klattp[KLATT_Skew];
  kt_frame.TLTdb = klattp[KLATT_Tilt];
  kt_frame.Kopen = klattp[KLATT_Kopen];

  // advance formants
  for (pk = 0; pk < N_PEAKS; pk++) {
   peaks[pk].freq1 += peaks[pk].freq_inc;
   peaks[pk].freq = (int)peaks[pk].freq1;
   peaks[pk].bw1 += peaks[pk].bw_inc;
   peaks[pk].bw = (int)peaks[pk].bw1;
   peaks[pk].bp1 += peaks[pk].bp_inc;
   peaks[pk].bp = (int)peaks[pk].bp1;
   peaks[pk].ap1 += peaks[pk].ap_inc;
   peaks[pk].ap = (int)peaks[pk].ap1;
```

```
  }

  // advance other parameters
  for (ix = 0; ix < N_KLATTP; ix++) {
   klattp1[ix] += klattp_inc[ix];
   klattp[ix] = (int)klattp1[ix];
  }

  for (ix = 0; ix <= 6; ix++) {
   kt_frame.Fhz_next[ix] = peaks[ix].freq;
   if (ix < 4)
    kt_frame.Bhz_next[ix] = peaks[ix].bw;
  }

  // advance the pitch
  wdata.pitch_ix += wdata.pitch_inc;
  if ((ix = wdata.pitch_ix>>8) > 127) ix = 127;
  x = wdata.pitch_env[ix] * wdata.pitch_range;
  wdata.pitch = (x>>8) + wdata.pitch_base;

  kt_globals.nspfr = (nsamples - sample_count);
  if (kt_globals.nspfr > STEPSIZE)
   kt_globals.nspfr = STEPSIZE;

  frame_init(&kt_frame); // get parameters for next frame of
speech

  if (parwave(&kt_frame) == 1)
   return 1; // output buffer is full
 }

 if (end_wave > 0) {
  fade = 64; // not followed by formant synthesis

  // fade out to avoid a click
  kt_globals.fadeout = fade;
  end_wave = 0;
```

```
  sample_count -= fade;
  kt_globals.nspfr = fade;
  if (parwave(&kt_frame) == 1)
   return 1; // output buffer is full
 }

 return 0;
}

static void SetSynth_Klatt(int length, frame_t *fr1, frame_t
*fr2, voice_t *v, int control)
{
 int ix;
 double next;
 int qix;
 int cmd;
 frame_t *fr3;
 static frame_t prev_fr;

 if (wvoice != NULL) {
  if ((wvoice->klattv[0] > 0) && (wvoice->klattv[0] <= 4 )) {
   kt_globals.glsource = wvoice->klattv[0];
   kt_globals.scale_wav = scale_wav_tab[kt_globals.glsource];
  }
  kt_globals.f0_flutter = wvoice->flutter/32;
 }

 end_wave = 0;
 if (control & 2)
  end_wave = 1; // fadeout at the end
 if (control & 1) {
  end_wave = 1;
  for (qix = wcmdq_head+1;; qix++) {
   if (qix >= N_WCMDQ) qix = 0;
   if (qix == wcmdq_tail) break;

   cmd = wcmdq[qix][0];
```

```
   if (cmd == WCMD_KLATT) {
    end_wave = 0; // next wave generation is from another
spectrum

    fr3 = (frame_t *)wcmdq[qix][2];
    for (ix = 1; ix < 6; ix++) {
     if (fr3->ffreq[ix] != fr2->ffreq[ix]) {
      // there is a discontinuity in formants
      end_wave = 2;
      break;
     }
    }
    break;
   }
   if ((cmd == WCMD_WAVE) || (cmd == WCMD_PAUSE))
    break; // next is not from spectrum, so continue until end of
wave cycle
  }
 }

 if (control & 1) {
  for (ix = 1; ix < 6; ix++) {
   if (prev_fr.ffreq[ix] != fr1->ffreq[ix]) {
    // Discontinuity in formants.
    // end_wave was set in SetSynth_Klatt() to fade out the
previous frame
    KlattReset(0);
    break;
   }
  }
  memcpy(&prev_fr, fr2, sizeof(prev_fr));
 }

 for (ix = 0; ix < N_KLATTP; ix++) {
  if ((ix >= 5) && ((fr1->frflags & FRFLAG_KLATT) == 0)) {
   klattp1[ix] = klattp[ix] = 0;
   klattp_inc[ix] = 0;
```

```
  } else {
   klattp1[ix] = klattp[ix] = fr1->klattp[ix];
   klattp_inc[ix] = (double)((fr2->klattp[ix] - klattp[ix]) *
STEPSIZE)/length;
  }
 }

 nsamples = length;

 for (ix = 1; ix < 6; ix++) {
  peaks[ix].freq1 = (fr1->ffreq[ix] * v->freq[ix] / 256.0) +
v->freqadd[ix];
  peaks[ix].freq = (int)peaks[ix].freq1;
  next = (fr2->ffreq[ix] * v->freq[ix] / 256.0) + v->freqadd[ix];
  peaks[ix].freq_inc =  ((next - peaks[ix].freq1) * STEPSIZE) /
length;

  if (ix < 4) {
   // klatt bandwidth for f1, f2, f3 (others are fixed)
   peaks[ix].bw1 = fr1->bw[ix] * 2;
   peaks[ix].bw = (int)peaks[ix].bw1;
   next = fr2->bw[ix] * 2;
   peaks[ix].bw_inc =  ((next - peaks[ix].bw1) * STEPSIZE) /
length;
  }
 }

 // nasal zero frequency
 peaks[0].freq1 = fr1->klattp[KLATT_FNZ] * 2;
 if (peaks[0].freq1 == 0)
  peaks[0].freq1 = kt_frame.Fhz[F_NP]; // if no nasal zero, set
it to same freq as nasal pole

 peaks[0].freq = (int)peaks[0].freq1;
 next = fr2->klattp[KLATT_FNZ] * 2;
 if (next == 0)
  next = kt_frame.Fhz[F_NP];
```

```
 peaks[0].freq_inc = ((next - peaks[0].freq1) * STEPSIZE) /
length;

 peaks[0].bw1 = 89;
 peaks[0].bw = 89;
 peaks[0].bw_inc = 0;

 if (fr1->frflags & FRFLAG_KLATT) {
  // the frame contains additional parameters for parallel
resonators
  for (ix = 1; ix < 7; ix++) {
   peaks[ix].bp1 = fr1->klatt_bp[ix] * 4; // parallel bandwidth
   peaks[ix].bp = (int)peaks[ix].bp1;
   next = fr2->klatt_bp[ix] * 4;
   peaks[ix].bp_inc =  ((next - peaks[ix].bp1) * STEPSIZE) /
length;

   peaks[ix].ap1 = fr1->klatt_ap[ix]; // parallal amplitude
   peaks[ix].ap = (int)peaks[ix].ap1;
   next = fr2->klatt_ap[ix];
   peaks[ix].ap_inc =  ((next - peaks[ix].ap1) * STEPSIZE) /
length;
  }
 }
}

int Wavegen_Klatt2(int length, int resume, frame_t *fr1, frame_t
*fr2)
{
 if (resume == 0)
  SetSynth_Klatt(length, fr1, fr2, wvoice, 1);

 return Wavegen_Klatt(resume);
}

void KlattInit()
```

```
{

 static short formant_hz[10] = { 280, 688, 1064, 2806, 3260,
3700, 6500, 7000, 8000, 280 };
 static short bandwidth[10] = { 89, 160, 70, 160, 200, 200, 500,
500, 500, 89 };
 static short parallel_amp[10] = { 0, 59, 59, 59, 59, 59, 59, 0,
0, 0 };
 static short parallel_bw[10] = { 59, 59, 89, 149, 200, 200, 500,
0, 0, 0 };

 int ix;

 sample_count = 0;

 kt_globals.synthesis_model = CASCADE_PARALLEL;
 kt_globals.samrate = 22050;

 kt_globals.glsource = IMPULSIVE;
 kt_globals.scale_wav = scale_wav_tab[kt_globals.glsource];
 kt_globals.natural_samples = natural_samples;
 kt_globals.num_samples = NUMBER_OF_SAMPLES;
 kt_globals.sample_factor = 3.0;
 kt_globals.nspfr = (kt_globals.samrate * 10) / 1000;
 kt_globals.outsl = 0;
 kt_globals.f0_flutter = 20;

 KlattReset(2);

 // set default values for frame parameters
 for (ix = 0; ix <= 9; ix++) {
  kt_frame.Fhz[ix] = formant_hz[ix];
  kt_frame.Bhz[ix] = bandwidth[ix];
  kt_frame.Ap[ix] = parallel_amp[ix];
  kt_frame.Bphz[ix] = parallel_bw[ix];
 }
 kt_frame.Bhz_next[F_NZ] = bandwidth[F_NZ];
```

```
 kt_frame.F0hz10 = 1000;
 kt_frame.AVdb = 59;
 kt_frame.ASP = 0;
 kt_frame.Kopen = 40;
 kt_frame.Aturb = 0;
 kt_frame.TLTdb = 0;
 kt_frame.AF = 50;
 kt_frame.Kskew = 0;
 kt_frame.AB = 0;
 kt_frame.AVpdb = 0;
 kt_frame.Gain0 = 62;
}
```

Chapter 46

./src/libespeak-ng/error.c

```
#include "config.h"

#include <errno.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>

#include "error.h"
#include "speech.h"
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "dictionary.h"

espeak_ng_STATUS
create_file_error_context(espeak_ng_ERROR_CONTEXT *context,
                          espeak_ng_STATUS status,
                          const char *filename)
{
```

```
 if (context) {
  if (*context) {
   free((*context)->name);
  } else {
   *context = malloc(sizeof(espeak_ng_ERROR_CONTEXT_));
   if (!*context)
    return ENOMEM;
  }
  (*context)->type = ERROR_CONTEXT_FILE;
  (*context)->name = strdup(filename);
  (*context)->version = 0;
  (*context)->expected_version = 0;
 }
 return status;
}

espeak_ng_STATUS
create_version_mismatch_error_context(espeak_ng_ERROR_CONTEXT
*context,
                                      const char *path_home,
                                      int version,
                                      int expected_version)
{
 if (context) {
  if (*context) {
   free((*context)->name);
  } else {
   *context = malloc(sizeof(espeak_ng_ERROR_CONTEXT_));
   if (!*context)
    return ENOMEM;
  }
  (*context)->type = ERROR_CONTEXT_VERSION;
  (*context)->name = strdup(path_home);
  (*context)->version = version;
  (*context)->expected_version = expected_version;
 }
 return ENS_VERSION_MISMATCH;
```

```
}

#pragma GCC visibility push(default)

ESPEAK_NG_API void
espeak_ng_ClearErrorContext(espeak_ng_ERROR_CONTEXT *context)
{
 if (context && *context) {
  free((*context)->name);
  free(*context);
  *context = NULL;
 }
}

ESPEAK_NG_API void
espeak_ng_GetStatusCodeMessage(espeak_ng_STATUS status,
                               char *buffer,
                               size_t length)
{
 switch (status)
 {
 case ENS_COMPILE_ERROR:
  strncpy0(buffer, "Compile error", length);
  break;
 case ENS_VERSION_MISMATCH:
  strncpy0(buffer, "Wrong version of espeak-ng-data", length);
  break;
 case ENS_FIFO_BUFFER_FULL:
  strncpy0(buffer, "The FIFO buffer is full", length);
  break;
 case ENS_NOT_INITIALIZED:
  strncpy0(buffer, "The espeak-ng library has not been
initialized", length);
  break;
 case ENS_AUDIO_ERROR:
  strncpy0(buffer, "Cannot initialize the audio device", length);
  break;
```

```
 case ENS_VOICE_NOT_FOUND:
  strncpy0(buffer, "The specified espeak-ng voice does not
exist", length);
  break;
 case ENS_MBROLA_NOT_FOUND:
  strncpy0(buffer, "Could not load the mbrola.dll file", length);
  break;
 case ENS_MBROLA_VOICE_NOT_FOUND:
  strncpy0(buffer, "Could not load the specified mbrola voice
file", length);
  break;
 case ENS_EVENT_BUFFER_FULL:
  strncpy0(buffer, "The event buffer is full", length);
  break;
 case ENS_NOT_SUPPORTED:
  strncpy0(buffer, "The requested functionality has not been
built into espeak-ng", length);
  break;
 case ENS_UNSUPPORTED_PHON_FORMAT:
  strncpy0(buffer, "The phoneme file is not in a supported
format", length);
  break;
 case ENS_NO_SPECT_FRAMES:
  strncpy0(buffer, "The spectral file does not contain any frame
data", length);
  break;
 case ENS_EMPTY_PHONEME_MANIFEST:
  strncpy0(buffer, "The phoneme manifest file does not contain
any phonemes", length);
  break;
 case ENS_UNKNOWN_PHONEME_FEATURE:
  strncpy0(buffer, "The phoneme feature is not recognised",
length);
  break;
 case ENS_UNKNOWN_TEXT_ENCODING:
  strncpy0(buffer, "The text encoding is not supported", length);
  break;
```

```
 default:
  if ((status & ENS_GROUP_MASK) == ENS_GROUP_ERRNO)
   strerror_r(status, buffer, length);
  else
   snprintf(buffer, length, "Unspecified error 0x%x", status);
  break;
 }
}

ESPEAK_NG_API void
espeak_ng_PrintStatusCodeMessage(espeak_ng_STATUS status,
                                 FILE *out,
                                 espeak_ng_ERROR_CONTEXT context)
{
 char error[512];
 espeak_ng_GetStatusCodeMessage(status, error, sizeof(error));
 if (context) {
  switch (context->type)
  {
  case ERROR_CONTEXT_FILE:
   fprintf(out, "Error processing file '%s': %s.\n",
context->name, error);
   break;
  case ERROR_CONTEXT_VERSION:
   fprintf(out, "Error: %s at '%s' (expected 0x%x, got 0x%x).\n",
           error, context->name, context->expected_version,
context->version);
   break;
  }
 } else
  fprintf(out, "Error: %s.\n", error);
}

#pragma GCC visibility pop
```

Chapter 47

./src/libespeak-ng/ieee80.c

```
//#define TEST_FP

#include <stdio.h>
#include <math.h>
#include "ieee80.h"

typedef float Single;

#ifndef THINK_C
 typedef double Double;
#else THINK_C
 typedef short double Double;
#endif THINK_C

#ifndef HUGE_VAL
# define HUGE_VAL HUGE
#endif HUGE_VAL

#ifdef applec /* The Apple C compiler works */
# define FloatToUnsigned(f) ((unsigned long)(f))
# define UnsignedToFloat(u) ((defdouble)(u))
#else applec
```

```
# define FloatToUnsigned(f) ((unsigned long)(((long)((f) -
2147483648.0)) + 2147483647L + 1))
# define UnsignedToFloat(u) (((defdouble)((long)((u) -
2147483647L - 1))) + 2147483648.0)
#endif applec

#define SEXP_MAX  255
#define SEXP_OFFSET  127
#define SEXP_SIZE  8
#define SEXP_POSITION (32-SEXP_SIZE-1)

defdouble
ConvertFromIeeeSingle(bytes)
char* bytes;
{
 defdouble f;
 long mantissa, expon;
 long bits;

 bits = ((unsigned long)(bytes[0] & 0xFF) << 24)
  | ((unsigned long)(bytes[1] & 0xFF) << 16)
  | ((unsigned long)(bytes[2] & 0xFF) << 8)
  |  (unsigned long)(bytes[3] & 0xFF);  /* Assemble bytes into a
long */

 if ((bits & 0x7FFFFFFF) == 0) {
  f = 0;
 }

 else {
  expon = (bits & 0x7F800000) >> SEXP_POSITION;
  if (expon == SEXP_MAX) {  /* Infinity or NaN */
   f = HUGE_VAL;  /* Map NaN's to infinity */
  }
  else {
   if (expon == 0) { /* Denormalized number */
    mantissa = (bits & 0x7fffff);
```

```
    f = ldexp((defdouble)mantissa, expon - SEXP_OFFSET -
SEXP_POSITION + 1);
   }
   else {    /* Normalized number */
    mantissa = (bits & 0x7fffff) + 0x800000; /* Insert hidden bit
*/
    f = ldexp((defdouble)mantissa, expon - SEXP_OFFSET -
SEXP_POSITION);
   }
  }
 }

 if (bits & 0x80000000)
  return -f;
 else
  return f;
}

void
ConvertToIeeeSingle(num, bytes)
defdouble num;
char* bytes;
{
 long sign;
 register long bits;

 if (num < 0) { /* Can't distinguish a negative zero */
  sign = 0x80000000;
  num *= -1;
 } else {
  sign = 0;
 }

 if (num == 0) {
  bits = 0;
 }
```

```
 else {
  defdouble fMant;
  int expon;

  fMant = frexp(num, &expon);

  if ((expon > (SEXP_MAX-SEXP_OFFSET+1)) || !(fMant < 1)) {
   /* NaN's and infinities fail second test */
   bits = sign | 0x7F800000;  /* +/- infinity */
  }

  else {
   long mantissa;

   if (expon < -(SEXP_OFFSET-2)) { /* Smaller than normalized */
    int shift = (SEXP_POSITION+1) + (SEXP_OFFSET-2) + expon;
    if (shift < 0) { /* Way too small: flush to zero */
     bits = sign;
    }
    else {   /* Nonzero denormalized number */
     mantissa = (long)(fMant * (1L << shift));
     bits = sign | mantissa;
    }
   }

   else {    /* Normalized number */
    mantissa = (long)floor(fMant * (1L << (SEXP_POSITION+1)));
    mantissa -= (1L << SEXP_POSITION);   /* Hide MSB */
    bits = sign | ((long)((expon + SEXP_OFFSET - 1)) <<
SEXP_POSITION) | mantissa;
   }
  }
 }

 bytes[0] = bits >> 24; /* Copy to byte string */
 bytes[1] = bits >> 16;
 bytes[2] = bits >> 8;
```

```
 bytes[3] = bits;
}

#define DEXP_MAX  2047
#define DEXP_OFFSET  1023
#define DEXP_SIZE  11
#define DEXP_POSITION (32-DEXP_SIZE-1)

defdouble
ConvertFromIeeeDouble(bytes)
char* bytes;
{
 defdouble f;
 long mantissa, expon;
 unsigned long first, second;

 first = ((unsigned long)(bytes[0] & 0xFF) << 24)
  | ((unsigned long)(bytes[1] & 0xFF) << 16)
  | ((unsigned long)(bytes[2] & 0xFF) << 8)
  |  (unsigned long)(bytes[3] & 0xFF);
 second= ((unsigned long)(bytes[4] & 0xFF) << 24)
  | ((unsigned long)(bytes[5] & 0xFF) << 16)
  | ((unsigned long)(bytes[6] & 0xFF) << 8)
  |  (unsigned long)(bytes[7] & 0xFF);

 if (first == 0 && second == 0) {
  f = 0;
 }

 else {
  expon = (first & 0x7FF00000) >> DEXP_POSITION;
  if (expon == DEXP_MAX) {  /* Infinity or NaN */
   f = HUGE_VAL;  /* Map NaN's to infinity */
  }
  else {
   if (expon == 0) { /* Denormalized number */
    mantissa = (first & 0x000FFFFF);
```

```
    f = ldexp((defdouble)mantissa, expon - DEXP_OFFSET -
DEXP_POSITION + 1);
    f += ldexp(UnsignedToFloat(second), expon - DEXP_OFFSET -
DEXP_POSITION + 1 - 32);
   }
   else {    /* Normalized number */
    mantissa = (first & 0x000FFFFF) + 0x00100000; /* Insert
hidden bit */
    f = ldexp((defdouble)mantissa, expon - DEXP_OFFSET -
DEXP_POSITION);
    f += ldexp(UnsignedToFloat(second), expon - DEXP_OFFSET -
DEXP_POSITION - 32);
   }
  }
 }

 if (first & 0x80000000)
  return -f;
 else
  return f;
}

void
ConvertToIeeeDouble(num, bytes)
defdouble num;
char *bytes;
{
 long sign;
 long first, second;

 if (num < 0) { /* Can't distinguish a negative zero */
  sign = 0x80000000;
  num *= -1;
 } else {
  sign = 0;
 }
```

```
if (num == 0) {
 first = 0;
 second = 0;
}

else {
 defdouble fMant, fsMant;
 int expon;

 fMant = frexp(num, &expon);

 if ((expon > (DEXP_MAX-DEXP_OFFSET+1)) || !(fMant < 1)) {
  /* NaN's and infinities fail second test */
  first = sign | 0x7FF00000;  /* +/- infinity */
  second = 0;
 }

 else {
  long mantissa;

  if (expon < -(DEXP_OFFSET-2)) { /* Smaller than normalized */
   int shift = (DEXP_POSITION+1) + (DEXP_OFFSET-2) + expon;
   if (shift < 0) { /* Too small for something in the MS word */
    first = sign;
    shift += 32;
    if (shift < 0) { /* Way too small: flush to zero */
     second = 0;
    }
    else {   /* Pretty small demorn */
     second = FloatToUnsigned(floor(ldexp(fMant, shift)));
    }
   }
   else {   /* Nonzero denormalized number */
    fsMant = ldexp(fMant, shift);
    mantissa = (long)floor(fsMant);
    first = sign | mantissa;
    second = FloatToUnsigned(floor(ldexp(fsMant - mantissa,
```

```
32)));
    }
   }

   else {    /* Normalized number */
    fsMant = ldexp(fMant, DEXP_POSITION+1);
    mantissa = (long)floor(fsMant);
    mantissa -= (1L << DEXP_POSITION);   /* Hide MSB */
    fsMant -= (1L << DEXP_POSITION);
    first = sign | ((long)((expon + DEXP_OFFSET - 1)) <<
DEXP_POSITION) | mantissa;
    second = FloatToUnsigned(floor(ldexp(fsMant - mantissa,
32)));
   }
  }
 }

 bytes[0] = first >> 24;
 bytes[1] = first >> 16;
 bytes[2] = first >> 8;
 bytes[3] = first;
 bytes[4] = second >> 24;
 bytes[5] = second >> 16;
 bytes[6] = second >> 8;
 bytes[7] = second;
}

defdouble
ConvertFromIeeeExtended(bytes)
char* bytes;
{
 defdouble f;
 long expon;
 unsigned long hiMant, loMant;

 expon = ((bytes[0] & 0x7F) << 8) | (bytes[1] & 0xFF);
 hiMant = ((unsigned long)(bytes[2] & 0xFF) << 24)
```

```
    | ((unsigned long)(bytes[3] & 0xFF) << 16)
    | ((unsigned long)(bytes[4] & 0xFF) << 8)
    | ((unsigned long)(bytes[5] & 0xFF));
 loMant = ((unsigned long)(bytes[6] & 0xFF) << 24)
    | ((unsigned long)(bytes[7] & 0xFF) << 16)
    | ((unsigned long)(bytes[8] & 0xFF) << 8)
    | ((unsigned long)(bytes[9] & 0xFF));

 if (expon == 0 && hiMant == 0 && loMant == 0) {
  f = 0;
 }
 else {
  if (expon == 0x7FFF) { /* Infinity or NaN */
   f = HUGE_VAL;
  }
  else {
   expon -= 16383;
   f  = ldexp(UnsignedToFloat(hiMant), expon-=31);
   f += ldexp(UnsignedToFloat(loMant), expon-=32);
  }
 }

 if (bytes[0] & 0x80)
  return -f;
 else
  return f;
}

void
ConvertToIeeeExtended(num, bytes)
defdouble num;
char *bytes;
{
 int sign;
 int expon;
 defdouble fMant, fsMant;
 unsigned long hiMant, loMant;
```

```
 if (num < 0) {
  sign = 0x8000;
  num *= -1;
 } else {
  sign = 0;
 }

 if (num == 0) {
  expon = 0; hiMant = 0; loMant = 0;
 }
 else {
  fMant = frexp(num, &expon);
  if ((expon > 16384) || !(fMant < 1)) { /* Infinity or NaN */
   expon = sign|0x7FFF; hiMant = 0; loMant = 0; /* infinity */
  }
  else { /* Finite */
   expon += 16382;
   if (expon < 0) { /* denormalized */
    fMant = ldexp(fMant, expon);
    expon = 0;
   }
   expon |= sign;
   fMant = ldexp(fMant, 32);          fsMant = floor(fMant);
hiMant = FloatToUnsigned(fsMant);
   fMant = ldexp(fMant - fsMant, 32); fsMant = floor(fMant);
loMant = FloatToUnsigned(fsMant);
  }
 }

 bytes[0] = expon >> 8;
 bytes[1] = expon;
 bytes[2] = hiMant >> 24;
 bytes[3] = hiMant >> 16;
 bytes[4] = hiMant >> 8;
 bytes[5] = hiMant;
 bytes[6] = loMant >> 24;
```

```
 bytes[7] = loMant >> 16;
 bytes[8] = loMant >> 8;
 bytes[9] = loMant;
}

#ifdef applec
# define IEEE
#endif applec
#ifdef THINK_C
# define IEEE
#endif THINK_C
#ifdef sgi
# define IEEE
#endif sgi
#ifdef sequent
# define IEEE
# define LITTLE_ENDIAN
#endif sequent
#ifdef sun
# define IEEE
#endif sun
#ifdef NeXT
# define IEEE
#endif NeXT

#ifdef TEST_FP

union SParts {
 Single s;
 long i;
};
union DParts {
 Double d;
 long i[2];
};
union EParts {
 defdouble e;
```

```
 short i[6];
};

int
GetHexValue(x)
register int x;
{
 x &= 0x7F;

 if ('0' <= x && x <= '9')
  x -= '0';
 else if ('a' <= x && x <= 'f')
  x = x - 'a' + 0xA;
 else if ('A' <= x && x <= 'F')
  x = x - 'A' + 0xA;
 else
  x = 0;

 return(x);
}

void
Hex2Bytes(hex, bytes)
register char *hex, *bytes;
{
 for ( ; *hex; hex += 2) {
  *bytes++ = (GetHexValue(hex[0]) << 4) | GetHexValue(hex[1]);
  if (hex[1] == 0)
   break; /* Guard against odd bytes */
 }
}

int
GetHexSymbol(x)
register int x;
{
 x &= 0xF;
```

```
 if (x <= 9)
  x += '0';
 else
  x += 'A' - 0xA;
 return(x);
}

void
Bytes2Hex(bytes, hex, nBytes)
register char *bytes, *hex;
register int nBytes;
{
 for ( ; nBytes--; bytes++) {
  *hex++ = GetHexSymbol(*bytes >> 4);
  *hex++ = GetHexSymbol(*bytes);
 }

}

void
MaybeSwapBytes(bytes, nBytes)
char* bytes;
int nBytes;
{
#ifdef LITTLE_ENDIAN
 register char *p, *q, t;
 for (p = bytes, q = bytes+nBytes-1; p < q; p++, q--) {
  t = *p;
  *p = *q;
  *q = t;
 }
#else
 if (bytes, nBytes);  /* Just so it's used, to avoid warnings */
#endif LITTLE_ENDIAN

}
```

```
float
MachineIEEESingle(bytes)
char* bytes;
{
 float t;
 MaybeSwapBytes(bytes, 4);
 t = *((float*)(bytes));
 MaybeSwapBytes(bytes, 4);
 return (t);
}

Double
MachineIEEEDouble(bytes)
char* bytes;
{
 Double t;
 MaybeSwapBytes(bytes, 8);
 t = *((Double*)(bytes));
 MaybeSwapBytes(bytes, 8);
 return (t);
}

void
TestFromIeeeSingle(hex)
char *hex;
{
 defdouble f;
 union SParts p;
 char bytes[4];

 Hex2Bytes(hex, bytes);
 f = ConvertFromIeeeSingle(bytes);
 p.s = f;

#ifdef IEEE
 printf("IEEE(%g) [%s] --> float(%g) [%08lX]\n",
 MachineIEEESingle(bytes),
```

```
 hex, f, p.i);
#else IEEE
 printf("IEEE[%s] --> float(%g) [%08lX]\n", hex, f, p.i);
#endif IEEE
}

void
TestToIeeeSingle(f)
defdouble f;
{
 union SParts p;
 char bytes[4];
 char hex[8+1];

 p.s = f;

 ConvertToIeeeSingle(f, bytes);
 Bytes2Hex(bytes, hex, 4);
#ifdef IEEE
 printf("float(%g) [%08lX] --> IEEE(%g) [%s]\n",
  f, p.i,
  MachineIEEESingle(bytes),
  hex
 );
#else IEEE
 printf("float(%g) [%08lX] --> IEEE[%s]\n", f, p.i, hex);
#endif IEEE
}

void
TestFromIeeeDouble(hex)
char *hex;
{
 defdouble f;
 union DParts p;
 char bytes[8];
```

```
 Hex2Bytes(hex, bytes);
 f = ConvertFromIeeeDouble(bytes);
 p.d = f;

#ifdef IEEE
 printf("IEEE(%g) [%.8s %.8s] --> double(%g) [%08lX %08lX]\n",
 MachineIEEEDouble(bytes),
 hex, hex+8, f, p.i[0], p.i[1]);
#else IEEE
 printf("IEEE[%.8s %.8s] --> double(%g) [%08lX %08lX]\n",
  hex, hex+8, f, p.i[0], p.i[1]);
#endif IEEE

}

void
TestToIeeeDouble(f)
defdouble f;
{
 union DParts p;
 char bytes[8];
 char hex[16+1];

 p.d = f;

 ConvertToIeeeDouble(f, bytes);
 Bytes2Hex(bytes, hex, 8);
#ifdef IEEE
 printf("double(%g) [%08lX %08lX] --> IEEE(%g) [%.8s %.8s]\n",
  f, p.i[0], p.i[1],
  MachineIEEEDouble(bytes),
  hex, hex+8
 );
#else IEEE
 printf("double(%g) [%08lX %08lX] --> IEEE[%.8s %.8s]\n",
  f, p.i[0], p.i[1], hex, hex+8
 );
```

```
#endif IEEE

}

void
TestFromIeeeExtended(hex)
char *hex;
{
 defdouble f;
 union EParts p;
 char bytes[12];

 Hex2Bytes(hex, bytes);
 f = ConvertFromIeeeExtended(bytes);
 p.e = f;

 bytes[11] = bytes[9];
 bytes[10] = bytes[8];
 bytes[9] = bytes[7];
 bytes[8] = bytes[6];
 bytes[7] = bytes[5];
 bytes[6] = bytes[4];
 bytes[5] = bytes[3];
 bytes[4] = bytes[2];
 bytes[3] = 0;
 bytes[2] = 0;

#if defined(applec) || defined(THINK_C)
 printf("IEEE(%g) [%.4s %.8s %.8s] --> extended(%g) [%04X
%04X%04X %04X%04X]\n",
  *((defdouble*)(bytes)),
  hex, hex+4, hex+12, f,
  p.i[0]&0xFFFF, p.i[2]&0xFFFF, p.i[3]&0xFFFF, p.i[4]&0xFFFF,
p.i[5]&0xFFFF
 );
#else /* !Macintosh */
 printf("IEEE[%.4s %.8s %.8s] --> extended(%g) [%04X %04X%04X
```

```
%04X%04X]\n",
   hex, hex+4, hex+12, f,
   p.i[0]&0xFFFF, p.i[2]&0xFFFF, p.i[3]&0xFFFF, p.i[4]&0xFFFF,
p.i[5]&0xFFFF
 );
#endif /* Macintosh */
}

void
TestToIeeeExtended(f)
defdouble f;
{
 char bytes[12];
 char hex[24+1];

 ConvertToIeeeExtended(f, bytes);
 Bytes2Hex(bytes, hex, 10);

 bytes[11] = bytes[9];
 bytes[10] = bytes[8];
 bytes[9] = bytes[7];
 bytes[8] = bytes[6];
 bytes[7] = bytes[5];
 bytes[6] = bytes[4];
 bytes[5] = bytes[3];
 bytes[4] = bytes[2];
 bytes[3] = 0;
 bytes[2] = 0;

#if defined(applec) || defined(THINK_C)
 printf("extended(%g) --> IEEE(%g) [%.4s %.8s %.8s]\n",
   f, *((defdouble*)(bytes)),
   hex, hex+4, hex+12
 );
#else /* !Macintosh */
 printf("extended(%g) --> IEEE[%.4s %.8s %.8s]\n",
   f,
```

```
  hex, hex+4, hex+12
 );
#endif /* Macintosh */
}

#include <signal.h>

void SignalFPE(i, j)
int i;
void (*j)();
{
 printf("[Floating Point Interrupt Caught.]\n", i, j);
 signal(SIGFPE, SignalFPE);
}

void
main()
{
 long d[3];
 char bytes[12];

 signal(SIGFPE, SignalFPE);

 TestFromIeeeSingle("00000000");
 TestFromIeeeSingle("80000000");
 TestFromIeeeSingle("3F800000");
 TestFromIeeeSingle("BF800000");
 TestFromIeeeSingle("40000000");
 TestFromIeeeSingle("C0000000");
 TestFromIeeeSingle("7F800000");
 TestFromIeeeSingle("FF800000");
 TestFromIeeeSingle("00800000");
 TestFromIeeeSingle("00400000");
 TestFromIeeeSingle("00000001");
 TestFromIeeeSingle("80000001");
 TestFromIeeeSingle("3F8FEDCB");
 TestFromIeeeSingle("7FC00100"); /* Quiet NaN(1) */
```

```
 TestFromIeeeSingle("7F800100"); /* Signalling NaN(1) */

 TestToIeeeSingle(0.0);
 TestToIeeeSingle(-0.0);
 TestToIeeeSingle(1.0);
 TestToIeeeSingle(-1.0);
 TestToIeeeSingle(2.0);
 TestToIeeeSingle(-2.0);
 TestToIeeeSingle(3.0);
 TestToIeeeSingle(-3.0);
#if !(defined(sgi) || defined(NeXT))
 TestToIeeeSingle(HUGE_VAL);
 TestToIeeeSingle(-HUGE_VAL);
#endif /* !sgi, !NeXT */

#ifdef IEEE
 /* These only work on big-endian IEEE machines */
 d[0] = 0x00800000L; MaybeSwapBytes((char*)d,4);
TestToIeeeSingle(*((float*)(&d[0])));  /* Smallest normalized */
 d[0] = 0x00400000L; MaybeSwapBytes((char*)d,4);
TestToIeeeSingle(*((float*)(&d[0])));  /* Almost largest
denormalized */
 d[0] = 0x00000001L; MaybeSwapBytes((char*)d,4);
TestToIeeeSingle(*((float*)(&d[0])));  /* Smallest denormalized
*/
 d[0] = 0x00000001L; MaybeSwapBytes((char*)d,4);
TestToIeeeSingle(*((float*)(&d[0])) * 0.5); /* Smaller than
smallest denorm */
 d[0] = 0x3F8FEDCBL; MaybeSwapBytes((char*)d,4);
TestToIeeeSingle(*((float*)(&d[0])));
#if !(defined(sgi) || defined(NeXT))
 d[0] = 0x7FC00100L; MaybeSwapBytes((char*)d,4);
TestToIeeeSingle(*((float*)(&d[0])));  /* Quiet NaN(1) */
 d[0] = 0x7F800100L; MaybeSwapBytes((char*)d,4);
TestToIeeeSingle(*((float*)(&d[0])));  /* Signalling NaN(1) */
#endif /* !sgi, !NeXT */
#endif IEEE
```

```
 TestFromIeeeDouble("0000000000000000");
 TestFromIeeeDouble("8000000000000000");
 TestFromIeeeDouble("3FF0000000000000");
 TestFromIeeeDouble("BFF0000000000000");
 TestFromIeeeDouble("4000000000000000");
 TestFromIeeeDouble("C000000000000000");
 TestFromIeeeDouble("7FF0000000000000");
 TestFromIeeeDouble("FFF0000000000000");
 TestFromIeeeDouble("0010000000000000");
 TestFromIeeeDouble("0008000000000000");
 TestFromIeeeDouble("0000000000000001");
 TestFromIeeeDouble("8000000000000001");
 TestFromIeeeDouble("3FFFEDCBA9876543");
 TestFromIeeeDouble("7FF8002000000000"); /* Quiet NaN(1) */
 TestFromIeeeDouble("7FF0002000000000"); /* Signalling NaN(1) */

 TestToIeeeDouble(0.0);
 TestToIeeeDouble(-0.0);
 TestToIeeeDouble(1.0);
 TestToIeeeDouble(-1.0);
 TestToIeeeDouble(2.0);
 TestToIeeeDouble(-2.0);
 TestToIeeeDouble(3.0);
 TestToIeeeDouble(-3.0);
#if !(defined(sgi) || defined(NeXT))
 TestToIeeeDouble(HUGE_VAL);
 TestToIeeeDouble(-HUGE_VAL);
#endif /* !sgi, !NeXT */

#ifdef IEEE
 /* These only work on IEEE machines */
 Hex2Bytes("0010000000000000", bytes); MaybeSwapBytes(bytes,8);
TestToIeeeDouble(*((Double*)(bytes))); /* Smallest normalized */
 Hex2Bytes("0010000080000000", bytes); MaybeSwapBytes(bytes,8);
TestToIeeeDouble(*((Double*)(bytes))); /* Normalized, problem
with unsigned */
```

```
 Hex2Bytes("0008000000000000", bytes); MaybeSwapBytes(bytes,8);
TestToIeeeDouble(*((Double*)(bytes))); /* Almost largest
denormalized */
 Hex2Bytes("0000000080000000", bytes); MaybeSwapBytes(bytes,8);
TestToIeeeDouble(*((Double*)(bytes))); /* Denorm problem with
unsigned */
 Hex2Bytes("0000000000000001", bytes); MaybeSwapBytes(bytes,8);
TestToIeeeDouble(*((Double*)(bytes))); /* Smallest denormalized
*/
 Hex2Bytes("0000000000000001", bytes); MaybeSwapBytes(bytes,8);
TestToIeeeDouble(*((Double*)(bytes)) * 0.5); /* Smaller than
smallest denorm */
 Hex2Bytes("3FFFEDCBA9876543", bytes); MaybeSwapBytes(bytes,8);
TestToIeeeDouble(*((Double*)(bytes))); /* accuracy test */
#if !(defined(sgi) || defined(NeXT))
 Hex2Bytes("7FF8002000000000", bytes); MaybeSwapBytes(bytes,8);
TestToIeeeDouble(*((Double*)(bytes))); /* Quiet NaN(1) */
 Hex2Bytes("7FF0002000000000", bytes); MaybeSwapBytes(bytes,8);
TestToIeeeDouble(*((Double*)(bytes))); /* Signalling NaN(1) */
#endif /* !sgi, !NeXT */
#endif IEEE

 TestFromIeeeExtended("00000000000000000000"); /* +0 */
 TestFromIeeeExtended("80000000000000000000"); /* -0 */
 TestFromIeeeExtended("3FFF8000000000000000"); /* +1 */
 TestFromIeeeExtended("BFFF8000000000000000"); /* -1 */
 TestFromIeeeExtended("40008000000000000000"); /* +2 */
 TestFromIeeeExtended("C0008000000000000000"); /* -2 */
 TestFromIeeeExtended("7FFF0000000000000000"); /* +infinity */
 TestFromIeeeExtended("FFFF0000000000000000"); /* -infinity */
 TestFromIeeeExtended("7FFF8001000000000000"); /* Quiet NaN(1) */
 TestFromIeeeExtended("7FFF0001000000000000"); /* Signalling
NaN(1) */
 TestFromIeeeExtended("3FFFFEDCBA9876543210"); /* accuracy test
*/

 TestToIeeeExtended(0.0);
```

```
 TestToIeeeExtended(-0.0);
 TestToIeeeExtended(1.0);
 TestToIeeeExtended(-1.0);
 TestToIeeeExtended(2.0);
 TestToIeeeExtended(-2.0);
#if !(defined(sgi) || defined(NeXT))
 TestToIeeeExtended(HUGE_VAL);
 TestToIeeeExtended(-HUGE_VAL);
#endif /* !sgi, !NeXT */

#if defined(applec) || defined(THINK_C)
 Hex2Bytes("7FFF00008001000000000000", bytes);
TestToIeeeExtended(*((long double*)(bytes))); /* Quiet NaN(1) */
 Hex2Bytes("7FFF00000001000000000000", bytes);
TestToIeeeExtended(*((long double*)(bytes))); /* Signalling
NaN(1) */
 Hex2Bytes("7FFE00008000000000000000", bytes);
TestToIeeeExtended(*((long double*)(bytes)));
 Hex2Bytes("000000008000000000000000", bytes);
TestToIeeeExtended(*((long double*)(bytes)));
 Hex2Bytes("000000000000000000000001", bytes);
TestToIeeeExtended(*((long double*)(bytes)));
 Hex2Bytes("3FFF0000FEDCBA9876543210", bytes);
TestToIeeeExtended(*((long double*)(bytes)));
#endif /* applec, THINK_C */
}
```

```
IEEE(0) [00000000] --> float(0) [00000000]
IEEE(-0) [80000000] --> float(-0) [80000000]
IEEE(1) [3F800000] --> float(1) [3F800000]
IEEE(-1) [BF800000] --> float(-1) [BF800000]
IEEE(2) [40000000] --> float(2) [40000000]
IEEE(-2) [C0000000] --> float(-2) [C0000000]
IEEE(INF) [7F800000] --> float(INF) [7F800000]
IEEE(-INF) [FF800000] --> float(-INF) [FF800000]
IEEE(1.17549e-38) [00800000] --> float(1.17549e-38) [00800000]
IEEE(5.87747e-39) [00400000] --> float(5.87747e-39) [00400000]
```

```
IEEE(1.4013e-45) [00000001] --> float(1.4013e-45) [00000001]
IEEE(-1.4013e-45) [80000001] --> float(-1.4013e-45) [80000001]
IEEE(1.12444) [3F8FEDCB] --> float(1.12444) [3F8FEDCB]
IEEE(NAN(001)) [7FC00100] --> float(INF) [7F800000]
IEEE(NAN(001)) [7F800100] --> float(INF) [7F800000]
float(0) [00000000] --> IEEE(0) [00000000]
float(-0) [80000000] --> IEEE(0) [00000000]
float(1) [3F800000] --> IEEE(1) [3F800000]
float(-1) [BF800000] --> IEEE(-1) [BF800000]
float(2) [40000000] --> IEEE(2) [40000000]
float(-2) [C0000000] --> IEEE(-2) [C0000000]
float(3) [40400000] --> IEEE(3) [40400000]
float(-3) [C0400000] --> IEEE(-3) [C0400000]
float(INF) [7F800000] --> IEEE(INF) [7F800000]
float(-INF) [FF800000] --> IEEE(-INF) [FF800000]
float(1.17549e-38) [00800000] --> IEEE(1.17549e-38) [00800000]
float(5.87747e-39) [00400000] --> IEEE(5.87747e-39) [00400000]
float(1.4013e-45) [00000001] --> IEEE(1.4013e-45) [00000001]
float(7.00649e-46) [00000000] --> IEEE(0) [00000000]
float(1.12444) [3F8FEDCB] --> IEEE(1.12444) [3F8FEDCB]
float(NAN(001)) [7FC00100] --> IEEE(INF) [7F800000]
IEEE(0) [00000000 00000000] --> double(0) [00000000 00000000]
IEEE(-0) [80000000 00000000] --> double(-0) [80000000 00000000]
IEEE(1) [3FF00000 00000000] --> double(1) [3FF00000 00000000]
IEEE(-1) [BFF00000 00000000] --> double(-1) [BFF00000 00000000]
IEEE(2) [40000000 00000000] --> double(2) [40000000 00000000]
IEEE(-2) [C0000000 00000000] --> double(-2) [C0000000 00000000]
IEEE(INF) [7FF00000 00000000] --> double(INF) [7FF00000 00000000]
IEEE(-INF) [FFF00000 00000000] --> double(-INF) [FFF00000
00000000]
IEEE(2.22507e-308) [00100000 00000000] --> double(2.22507e-308)
[00100000 00000000]
IEEE(1.11254e-308) [00080000 00000000] --> double(1.11254e-308)
[00080000 00000000]
IEEE(4.94066e-324) [00000000 00000001] --> double(4.94066e-324)
[00000000 00000001]
IEEE(-4.94066e-324) [80000000 00000001] --> double(-4.94066e-324)
```

```
[80000000 00000001]
IEEE(1.99556) [3FFFEDCB A9876543] --> double(1.99556) [3FFFEDCB
A9876543]
IEEE(NAN(001)) [7FF80020 00000000] --> double(INF) [7FF00000
00000000]
IEEE(NAN(001)) [7FF00020 00000000] --> double(INF) [7FF00000
00000000]
double(0) [00000000 00000000] --> IEEE(0) [00000000 00000000]
double(-0) [80000000 00000000] --> IEEE(0) [00000000 00000000]
double(1) [3FF00000 00000000] --> IEEE(1) [3FF00000 00000000]
double(-1) [BFF00000 00000000] --> IEEE(-1) [BFF00000 00000000]
double(2) [40000000 00000000] --> IEEE(2) [40000000 00000000]
double(-2) [C0000000 00000000] --> IEEE(-2) [C0000000 00000000]
double(3) [40080000 00000000] --> IEEE(3) [40080000 00000000]
double(-3) [C0080000 00000000] --> IEEE(-3) [C0080000 00000000]
double(INF) [7FF00000 00000000] --> IEEE(INF) [7FF00000 00000000]
double(-INF) [FFF00000 00000000] --> IEEE(-INF) [FFF00000
00000000]
double(2.22507e-308) [00100000 00000000] --> IEEE(2.22507e-308)
[00100000 00000000]
double(2.22507e-308) [00100000 80000000] --> IEEE(2.22507e-308)
[00100000 80000000]
double(1.11254e-308) [00080000 00000000] --> IEEE(1.11254e-308)
[00080000 00000000]
double(1.061e-314) [00000000 80000000] --> IEEE(1.061e-314)
[00000000 80000000]
double(4.94066e-324) [00000000 00000001] --> IEEE(4.94066e-324)
[00000000 00000001]
double(2.47033e-324) [00000000 00000000] --> IEEE(0) [00000000
00000000]
double(1.99556) [3FFFEDCB A9876543] --> IEEE(1.99556) [3FFFEDCB
A9876543]
double(NAN(001)) [7FF80020 00000000] --> IEEE(INF) [7FF00000
00000000]
IEEE(0) [0000 00000000 00000000] --> extended(0) [0000 00000000
00000000]
IEEE(-0) [8000 00000000 00000000] --> extended(-0) [8000 00000000
```

```
00000000]
IEEE(1) [3FFF 80000000 00000000] --> extended(1) [3FFF 80000000
00000000]
IEEE(-1) [BFFF 80000000 00000000] --> extended(-1) [BFFF 80000000
00000000]
IEEE(2) [4000 80000000 00000000] --> extended(2) [4000 80000000
00000000]
IEEE(-2) [C000 80000000 00000000] --> extended(-2) [C000 80000000
00000000]
IEEE(INF) [7FFF 00000000 00000000] --> extended(INF) [7FFF
00000000 00000000]
IEEE(-INF) [FFFF 00000000 00000000] --> extended(-INF) [FFFF
00000000 00000000]
IEEE(NAN(001)) [7FFF 80010000 00000000] --> extended(INF) [7FFF
00000000 00000000]
IEEE(NAN(001)) [7FFF 00010000 00000000] --> extended(INF) [7FFF
00000000 00000000]
IEEE(1.99111) [3FFF FEDCBA98 76543210] --> extended(1.99111)
[3FFF FEDCBA98 76543210]
extended(0) --> IEEE(0) [0000 00000000 00000000]
extended(-0) --> IEEE(0) [0000 00000000 00000000]
extended(1) --> IEEE(1) [3FFF 80000000 00000000]
extended(-1) --> IEEE(-1) [BFFF 80000000 00000000]
extended(2) --> IEEE(2) [4000 80000000 00000000]
extended(-2) --> IEEE(-2) [C000 80000000 00000000]
extended(INF) --> IEEE(INF) [7FFF 00000000 00000000]
extended(-INF) --> IEEE(-INF) [FFFF 00000000 00000000]
extended(NAN(001)) --> IEEE(INF) [7FFF 00000000 00000000]
extended(5.94866e+4931) --> IEEE(5.94866e+4931) [7FFE 80000000
00000000]
extended(1e-4927) --> IEEE(1e-4927) [0000 80000000 00000000]
extended(1e-4927) --> IEEE(1e-4927) [0000 00000000 00000001]
extended(1.99111) --> IEEE(1.99111) [3FFF FEDCBA98 76543210]
*/

#endif TEST_FP
```

Chapter 48

./src/libespeak-ng/phonemelist.c

```
#include "config.h"

#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "phonemelist.h"
#include "synthdata.h"

#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

const unsigned char pause_phonemes[8] = {
 0, phonPAUSE_VSHORT, phonPAUSE_SHORT, phonPAUSE, phonPAUSE_LONG,
```

```
phonGLOTTALSTOP, phonPAUSE_LONG, phonPAUSE_LONG
};

static int SubstitutePhonemes(PHONEME_LIST *plist_out, int
n_ph_list2, PHONEME_LIST2 *ph_list2)
{
 // Copy the phonemes list and perform any substitutions that are
required for the
 // current voice
 int ix;
 int k;
 int replace_flags;
 int n_plist_out = 0;
 bool word_end;
 PHONEME_LIST2 *plist2;
 PHONEME_TAB *next = NULL;
 int deleted_sourceix = -1;

 for (ix = 0; (ix < n_ph_list2) && (n_plist_out <
N_PHONEME_LIST); ix++) {
  plist2 = &ph_list2[ix];
  if (deleted_sourceix != -1) {
   plist2->sourceix = deleted_sourceix;
   deleted_sourceix = -1;
  }

  // don't do any substitution if the language has been
temporarily changed
  if (!(plist2->synthflags & SFLAG_SWITCHED_LANG)) {
   if (ix < (n_ph_list2 -1))
    next = phoneme_tab[ph_list2[ix+1].phcode];

   word_end = false;
   if ((plist2+1)->sourceix || ((next != 0) && (next->type ==
phPAUSE)))
    word_end = true; // this phoneme is the end of a word
```

```
   // check whether a Voice has specified that we should replace
this phoneme
   for (k = 0; k < n_replace_phonemes; k++) {
    if (plist2->phcode == replace_phonemes[k].old_ph) {
     replace_flags = replace_phonemes[k].type;

     if ((replace_flags & 1) && (word_end == false))
      continue; // this replacement only occurs at the end of a
word

     if ((replace_flags & 2) && ((plist2->stresslevel & 0x7) >
3))
      continue; // this replacement doesn't occur in stressed
syllables

     if ((replace_flags & 4) && (plist2->sourceix == 0))
      continue; // this replacement only occurs at the start of a
word

     // substitute the replacement phoneme
     plist2->phcode = replace_phonemes[k].new_ph;
     if ((plist2->stresslevel > 1) &&
(phoneme_tab[plist2->phcode]->phflags & phUNSTRESSED))
      plist2->stresslevel = 0; // the replacement must be
unstressed
     break;
    }
   }

   if (plist2->phcode == 0) {
    deleted_sourceix = plist2->sourceix;
    continue; // phoneme has been replaced by NULL, so don't copy
it
   }
  }

  // copy phoneme into the output list
```

```
  memcpy(&plist_out[n_plist_out], plist2, sizeof(PHONEME_LIST2));
  plist_out[n_plist_out].ph = phoneme_tab[plist2->phcode];
  plist_out[n_plist_out].type = plist_out[n_plist_out].ph->type;
  n_plist_out++;
 }
 return n_plist_out;
}

void MakePhonemeList(Translator *tr, int post_pause, bool
start_sentence, int *n_ph_list2, PHONEME_LIST2 *ph_list2)
{
 int ix = 0;
 int j;
 int insert_ph = 0;
 PHONEME_LIST *phlist;
 PHONEME_TAB *ph;
 PHONEME_TAB *next, *next2;
 int unstress_count = 0;
 int word_stress = 0;
 int current_phoneme_tab;
 int max_stress;
 int voicing;
 int regression;
 int end_sourceix;
 int alternative;
 int delete_count;
 int word_start;
 bool inserted;
 bool deleted;
 PHONEME_DATA phdata;
 bool start_of_clause = true;

 int n_ph_list3;
 PHONEME_LIST *plist3;
 PHONEME_LIST *plist3_inserted = NULL;
 PHONEME_LIST ph_list3[N_PHONEME_LIST];
```

```
 PHONEME_LIST2 *plist2;
 WORD_PH_DATA worddata;

 memset(&worddata, 0, sizeof(worddata));
 plist2 = ph_list2;
 phlist = phoneme_list;
 end_sourceix = plist2[*n_ph_list2 - 1].sourceix;

 // is the last word of the clause unstressed ?
 max_stress = 0;
 for (j = *n_ph_list2 - 3; j >= 0; j--) {
  // start with the last phoneme (before the terminating pauses)
and move backwards
  if ((plist2[j].stresslevel & 0x7f) > max_stress)
   max_stress = plist2[j].stresslevel & 0x7f;
  if (plist2[j].sourceix != 0)
   break;
 }
 if (max_stress < 4) {
  // the last word is unstressed, look for a previous word that
can be stressed
  while (--j >= 0) {
   if (plist2[j].synthflags & SFLAG_PROMOTE_STRESS) { //
dictionary flags indicated that this stress can be promoted
    plist2[j].stresslevel = 4; // promote to stressed
    break;
   }
   if (plist2[j].stresslevel >= 4) {
    // found a stressed syllable, so stop looking
    break;
   }
  }
 }

 // look for switch of phoneme tables
 delete_count = 0;
 current_phoneme_tab = tr->phoneme_tab_ix;
```

```
 int deleted_sourceix = -1;
 for (j = 0; j < *n_ph_list2; j++) {
  if (current_phoneme_tab != tr->phoneme_tab_ix)
   plist2[j].synthflags |= SFLAG_SWITCHED_LANG;

  if (delete_count > 0) {
   memcpy(&plist2[j-delete_count], &plist2[j],
sizeof(plist2[0]));
   if (deleted_sourceix != -1) {
    plist2[j-delete_count].sourceix = deleted_sourceix;
    deleted_sourceix = -1;
   }
  }

  if (plist2[j].phcode == phonSWITCH) {
   if ((!(plist2[j].synthflags & SFLAG_EMBEDDED)) && (
           (plist2[j].tone_ph == current_phoneme_tab) ||
           (plist2[j+1].phcode == phonSWITCH) ||
           ((plist2[j+1].phcode == phonPAUSE) &&
(plist2[j+2].phcode == phonSWITCH))
           )) {
    // delete this phonSWITCH if it's switching to the current
phoneme table, or
    // delete this phonSWITCH if its followed by another
phonSWITCH
    if (deleted_sourceix == -1 && plist2[j].sourceix != 0)
     deleted_sourceix = plist2[j].sourceix;
    delete_count++;
   } else
    current_phoneme_tab = plist2[j].tone_ph;
  }

 }

 if ((regression = tr->langopts.param[LOPT_REGRESSIVE_VOICING])
!= 0) {
  // set consonant clusters to all voiced or all unvoiced
```

```
// Regressive
int type;
bool stop_propagation = false;
voicing = 0;

for (j = *n_ph_list2 - 1; j >= 0; j--) {
 ph = phoneme_tab[plist2[j].phcode];
 if (ph == NULL)
  continue;

 if (plist2[j].synthflags & SFLAG_SWITCHED_LANG) {
  stop_propagation = false;
  voicing = 0;
  if (regression & 0x100)
   voicing = 1; // word-end devoicing
  continue;
 }

 type = ph->type;

 if (regression & 0x2) {
  // [v] amd [v;] don't cause regression, or [R^]
  if (((ph->mnemonic & 0xff) == 'v') || ((ph->mnemonic & 0xff)
== 'R')) {
   stop_propagation = true;
   if (regression & 0x10)
    voicing = 0;
  }
 }

 if ((type == phSTOP) || type == (phFRICATIVE)) {
  if ((voicing == 0) && (regression & 0xf))
   voicing = 1;
  else if ((voicing == 2) && (ph->end_type != 0)) // use
end_type field for voicing_switch for consonants
   plist2[j].phcode = ph->end_type; // change to voiced
equivalent
```

```
   } else if ((type == phVSTOP) || type == (phVFRICATIVE)) {
    if ((voicing == 0) && (regression & 0xf))
     voicing = 2;
    else if ((voicing == 1) && (ph->end_type != 0))
     plist2[j].phcode = ph->end_type; // change to unvoiced
equivalent
   } else {
    if (regression & 0x8) {
     // LANG=Polish, propagate through liquids and nasals
     if ((type == phPAUSE) || (type == phVOWEL))
      voicing = 0;
    } else
     voicing = 0;
   }
   if (stop_propagation) {
    voicing = 0;
    stop_propagation = false;
   }

   if (plist2[j].sourceix) {
    if (regression & 0x04) {
     // stop propagation at a word boundary
     voicing = 0;
    }
    if (regression & 0x100) {
     // devoice word-final consonants, unless propagating voiced
     if (voicing == 0)
      voicing = 1;
    }
   }
  }
 }

 n_ph_list3 = SubstitutePhonemes(ph_list3, (int) *n_ph_list2,
ph_list2) - 2;

 for (j = 0; (j < n_ph_list3) && (ix < N_PHONEME_LIST-3);) {
```

```
  if (ph_list3[j].sourceix) {
   // start of a word
   int k;
   int nextw;
   word_stress = 0;

   // find the highest stress level in this word
   for (nextw = j; nextw < n_ph_list3;) {
    if (ph_list3[nextw].stresslevel > word_stress)
     word_stress = ph_list3[nextw].stresslevel;

    nextw++;
    if (ph_list3[nextw].sourceix)
     break; // start of the next word
   }
   for (k = j; k < nextw; k++)
    ph_list3[k].wordstress = word_stress;
   j = nextw;
  } else
   j++;
 }

 // transfer all the phonemes of the clause into phoneme_list
 ph = phoneme_tab[phonPAUSE];
 ph_list3[0].ph = ph;
 word_start = 1;

 for (j = 0; insert_ph || ((j < n_ph_list3) && (ix <
N_PHONEME_LIST-3)); j++) {
  plist3 = &ph_list3[j];

  inserted = false;
  deleted = false;
  if (insert_ph != 0) {
   // we have a (linking) phoneme which we need to insert here
   next = phoneme_tab[plist3->phcode];      // this phoneme, i.e.
after the insert
```

```
   // re-use the previous entry for the inserted phoneme.
   // That's OK because we don't look backwards from plist3   ***
but CountVowelPosition() and isAfterStress does !!!
   j--;
   plist3 = plist3_inserted = &ph_list3[j];
   if (j > 0) {
    // move all previous phonemes in the word back one place
    int k;
    if (word_start > 0) {
     k = word_start;
     word_start--;
    } else
     k = 2;   // No more space, don't loose the start of word
mark at ph_list2[word_start]
    for (; k <= j; k++)
     memcpy(&ph_list3[k-1], &ph_list3[k], sizeof(*plist3));
   }
   memset(&plist3[0], 0, sizeof(*plist3));
   plist3->phcode = insert_ph;
   ph = phoneme_tab[insert_ph];
   plist3->ph = ph;
   insert_ph = 0;
   inserted = true; // don't insert the same phoneme repeatedly
  } else {
   // otherwise get the next phoneme from the list
   if (plist3->sourceix != 0)
    word_start = j;

   ph = phoneme_tab[plist3->phcode];
   plist3[0].ph = ph;

   if (plist3->phcode == phonSWITCH) {
    // change phoneme table
    SelectPhonemeTable(plist3->tone_ph);
   }
   next = phoneme_tab[plist3[1].phcode]; // the phoneme after
```

```
this one
   plist3[1].ph = next;
  }

  if (ph == NULL) continue;

  InterpretPhoneme(tr, 0x100, plist3, &phdata, &worddata);

  if ((alternative = phdata.pd_param[pd_CHANGE_NEXTPHONEME]) > 0)
{
   ph_list3[j+1].ph = phoneme_tab[alternative];
   ph_list3[j+1].phcode = alternative;
   ph_list3[j+1].type = phoneme_tab[alternative]->type;
   next = phoneme_tab[alternative];
  }

  if (((alternative = phdata.pd_param[pd_INSERTPHONEME]) > 0) &&
(inserted == false)) {
   // PROBLEM: if we insert a phoneme before a vowel then we
loose the stress.
   PHONEME_TAB *ph2;
   ph2 = ph;

   insert_ph = plist3->phcode;
   ph = phoneme_tab[alternative];
   plist3->ph = ph;
   plist3->phcode = alternative;

   if (ph->type == phVOWEL) {
    plist3->synthflags |= SFLAG_SYLLABLE;
    if (ph2->type != phVOWEL)
     plist3->stresslevel = 0; // change from non-vowel to vowel,
make sure it's unstressed
   } else
    plist3->synthflags &= ~SFLAG_SYLLABLE;

   // re-interpret the changed phoneme
```

```
   // But it doesn't obey a second ChangePhoneme()
   InterpretPhoneme(tr, 0x100, plist3, &phdata, &worddata);
  }

  if ((alternative = phdata.pd_param[pd_CHANGEPHONEME]) > 0) {
   PHONEME_TAB *ph2;
   ph2 = ph;
   ph = phoneme_tab[alternative];
   plist3->ph = ph;
   plist3->phcode = alternative;

   if (alternative == 1)
    deleted = true; // NULL phoneme, discard
   else {
    if (ph->type == phVOWEL) {
     plist3->synthflags |= SFLAG_SYLLABLE;
     if (ph2->type != phVOWEL)
      plist3->stresslevel = 0; // change from non-vowel to vowel,
make sure it's unstressed
    } else
     plist3->synthflags &= ~SFLAG_SYLLABLE;

    // re-interpret the changed phoneme
    // But it doesn't obey a second ChangePhoneme()
    InterpretPhoneme(tr, 0x100, plist3, &phdata, &worddata);
   }
  }

  if ((ph->type == phVOWEL) && (deleted == false)) {
   PHONEME_LIST *p;

   // Check for consecutive unstressed syllables, even across
word boundaries.
   // Do this after changing phonemes according to stress level.
   if (plist3->stresslevel <= 1) {
    // an unstressed vowel
    unstress_count++;
```

```
    if (tr->langopts.stress_flags & 0x08) {
     // change sequences of consecutive unstressed vowels in
unstressed words to diminished stress (TEST)
     for (p = plist3+1; p->type != phPAUSE; p++) {
      if (p->type == phVOWEL) {
       if (p->stresslevel <= 1) {
        if (plist3->wordstress < 4)
         plist3->stresslevel = 0;
        if (p->wordstress < 4)
         p->stresslevel = 0;
       }
       break;
      }
     }
    } else {
     if ((unstress_count > 1) && ((unstress_count & 1) == 0)) {
      // in a sequence of unstressed syllables, reduce alternate
syllables to 'diminished'
      // stress.  But not for the last phoneme of a stressed word
      if ((tr->langopts.stress_flags & S_NO_DIM) || ((word_stress
> 3) && ((plist3+1)->sourceix != 0))) {
       // An unstressed final vowel of a stressed word
       unstress_count = 1; // try again for next syllable
      } else
       plist3->stresslevel = 0; // change stress to 'diminished'
     }
    }
   } else
    unstress_count = 0;
  }

  if ((plist3+1)->synthflags & SFLAG_LENGTHEN) {
   static char types_double[] = { phFRICATIVE, phVFRICATIVE,
phNASAL, phLIQUID, 0 };
   if ((j > 0) && (strchr(types_double, next->type))) {
    // lengthen this consonant by doubling it
```

```
    // BUT, can't insert a phoneme at position plist3[0] because
it crashes PrevPh()
    insert_ph = next->code;
    (plist3+1)->synthflags ^= SFLAG_LENGTHEN;
   }
  }

  if ((plist3+1)->sourceix != 0) {
   int x;

   if (tr->langopts.vowel_pause && (ph->type != phPAUSE)) {

    if ((ph->type != phVOWEL) && (tr->langopts.vowel_pause &
0x200)) {
     // add a pause after a word which ends in a consonant
     insert_ph = phonPAUSE_NOLINK;
    }

    if (next->type == phVOWEL) {
     if ((x = tr->langopts.vowel_pause & 0x0c) != 0) {
      // break before a word which starts with a vowel
      if (x == 0xc)
       insert_ph = phonPAUSE_NOLINK;
      else
       insert_ph = phonPAUSE_VSHORT;
     }

     if ((ph->type == phVOWEL) && ((x = tr->langopts.vowel_pause
& 0x03) != 0)) {
      // adjacent vowels over a word boundary
      if (x == 2)
       insert_ph = phonPAUSE_SHORT;
      else
       insert_ph = phonPAUSE_VSHORT;
     }

     if (((plist3+1)->stresslevel >= 4) &&
```

```
(tr->langopts.vowel_pause & 0x100)) {
      // pause before a words which starts with a stressed vowel
      insert_ph = phonPAUSE_SHORT;
     }
    }
   }

   if ((plist3 != plist3_inserted) && (ix > 0)) {
    if ((x = (tr->langopts.word_gap & 0x7)) != 0) {
     if ((x > 1) || ((insert_ph != phonPAUSE_SHORT) && (insert_ph
!= phonPAUSE_NOLINK))) {
      // don't reduce the pause
      insert_ph = pause_phonemes[x];
     }
    }
    if (option_wordgap > 0)
     insert_ph = phonPAUSE_LONG;
   }
  }

  next2 = phoneme_tab[plist3[2].phcode];
  plist3[2].ph = next2;

  if ((insert_ph == 0) && (phdata.pd_param[pd_APPENDPHONEME] !=
0))
   insert_ph = phdata.pd_param[pd_APPENDPHONEME];

  if (deleted == false) {
   phlist[ix].ph = ph;
   phlist[ix].type = ph->type;
   phlist[ix].env = PITCHfall; // default, can be changed in the
"intonation" module
   phlist[ix].synthflags = plist3->synthflags;
   phlist[ix].stresslevel = plist3->stresslevel & 0xf;
   phlist[ix].wordstress = plist3->wordstress;
   phlist[ix].tone_ph = plist3->tone_ph;
   phlist[ix].sourceix = 0;
```

```
   phlist[ix].phcode = ph->code;

   if (plist3->sourceix != 0) {
    phlist[ix].sourceix = plist3->sourceix;
    phlist[ix].newword = PHLIST_START_OF_WORD;

    if (start_sentence) {
     phlist[ix].newword |= PHLIST_START_OF_SENTENCE;
     start_sentence = false;
    }

    if (start_of_clause) {
     phlist[ix].newword |= PHLIST_START_OF_CLAUSE;
     start_of_clause = false;
    }
   } else
    phlist[ix].newword = 0;

   phlist[ix].length = phdata.pd_param[i_SET_LENGTH]*2;
   if ((ph->code == phonPAUSE_LONG) && (option_wordgap > 0) &&
(plist3[1].sourceix != 0)) {
    phlist[ix].ph = phoneme_tab[phonPAUSE_SHORT];
    phlist[ix].length = option_wordgap*14; // 10mS per unit at
the default speed
   }

   if (ph->type == phVOWEL || ph->type == phLIQUID || ph->type ==
phNASAL || ph->type == phVSTOP || ph->type == phVFRICATIVE ||
(ph->phflags & phPREVOICE)) {
    phlist[ix].length = 128; // length_mod
    phlist[ix].env = PITCHfall;
   }

   phlist[ix].prepause = 0;
   phlist[ix].amp = 20; // default, will be changed later
   phlist[ix].pitch1 = 255;
   phlist[ix].pitch2 = 255;
```

```
   ix++;
  }
 }

 phlist[ix].newword = PHLIST_END_OF_CLAUSE;

 phlist[ix].phcode = phonPAUSE;
 phlist[ix].type = phPAUSE; // terminate with 2 Pause phonemes
 phlist[ix].length = post_pause; // length of the pause, depends
on the punctuation
 phlist[ix].sourceix = end_sourceix;
 phlist[ix].synthflags = 0;
 phlist[ix++].ph = phoneme_tab[phonPAUSE];

 phlist[ix].phcode = phonPAUSE;
 phlist[ix].type = phPAUSE;
 phlist[ix].length = 0;
 phlist[ix].sourceix = 0;
 phlist[ix].synthflags = 0;
 phlist[ix++].ph = phoneme_tab[phonPAUSE_SHORT];

 n_phoneme_list = ix;
}
```

Chapter 49

./src/libespeak-ng/compiledict.c

```
#include "config.h"

#include <ctype.h>
#include <errno.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wctype.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "compiledict.h"
#include "dictionary.h"
#include "readclause.h"

#include "error.h"
#include "speech.h"
#include "phoneme.h"
```

```
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

static FILE *f_log = NULL;

extern char word_phonemes[N_WORD_PHONEMES];    // a word
translated into phoneme codes

static int linenum;
static int error_count;
static bool text_mode = false;
static int debug_flag = 0;
static int error_need_dictionary = 0;

// A hash chain is a linked-list of hash chain entry objects:
//     struct hash_chain_entry {
//         hash_chain_entry *next_entry;
//         // dict_line output from compile_line:
//         uint8_t length;
//         char contents[length];
//     };
static char *hash_chains[N_HASH_DICT];

static char letterGroupsDefined[N_LETTER_GROUPS];

MNEM_TAB mnem_rules[] = {
 { "unpr",     DOLLAR_UNPR },
 { "noprefix", DOLLAR_NOPREFIX },  // rule fails if a prefix has
been removed
 { "list",     DOLLAR_LIST },    // a pronunciation is given in
the *_list file

 { "w_alt1", 0x11 },
 { "w_alt2", 0x12 },
 { "w_alt3", 0x13 },
 { "w_alt4", 0x14 },
```

```
 { "w_alt5", 0x15 },
 { "w_alt6", 0x16 },
 { "w_alt",  0x11 }, // note: put longer names before their sub-
strings

 { "p_alt1", 0x21 },
 { "p_alt2", 0x22 },
 { "p_alt3", 0x23 },
 { "p_alt4", 0x24 },
 { "p_alt5", 0x25 },
 { "p_alt6", 0x26 },
 { "p_alt",  0x21 },

 { NULL, -1 }
};

MNEM_TAB mnem_flags[] = {
 // these in the first group put a value in bits0-3 of
dictionary_flags
 { "$1",   0x41 }, // stress on 1st syllable
 { "$2",   0x42 }, // stress on 2nd syllable
 { "$3",   0x43 },
 { "$4",   0x44 },
 { "$5",   0x45 },
 { "$6",   0x46 },
 { "$7",   0x47 },
 { "$u",   0x48 }, // reduce to unstressed
 { "$u1",  0x49 },
 { "$u2",  0x4a },
 { "$u3",  0x4b },
 { "$u+",  0x4c }, // reduce to unstressed, but stress at end of
clause
 { "$u1+", 0x4d },
 { "$u2+", 0x4e },
 { "$u3+", 0x4f },

 // these set the corresponding numbered bit if dictionary_flags
```

```
 { "$pause",          8 }, // ensure pause before this word
 { "$strend",         9 }, // full stress if at end of clause
 { "$strend2",       10 }, // full stress if at end of clause, or
only followed by unstressed
 { "$unstressend",   11 }, // reduce stress at end of clause
 { "$accent_before", 12 }, // used with accent names, say this
accent name before the letter name
 { "$abbrev",        13 }, // use this pronuciation rather than
split into letters

 // language specific
 { "$double",        14 }, // IT double the initial consonant of
next word
 { "$alt",           15 }, // use alternative pronunciation
 { "$alt1",          15 }, // synonym for $alt
 { "$alt2",          16 },
 { "$alt3",          17 },
 { "$alt4",          18 },
 { "$alt5",          19 },
 { "$alt6",          20 },
 { "$alt7",          21 },

 { "$combine",       23 }, // Combine with the next word

 { "$dot",           24 }, // ignore '.' after this word
(abbreviation)
 { "$hasdot",        25 }, // use this pronunciation if there is
a dot after the word

 { "$max3",          27 }, // limit to 3 repetitions
 { "$brk",           28 }, // a shorter $pause
 { "$text",          29 }, // word translates to replcement text,
not phonemes

 // flags in dictionary word 2
 { "$verbf",      0x20 }, // verb follows
 { "$verbsf",     0x21 }, // verb follows, allow -s suffix
```

```
 { "$nounf",      0x22 }, // noun follows
 { "$pastf",      0x23 }, // past tense follows
 { "$verb",       0x24 }, // use this pronunciation when its a
verb
 { "$noun",       0x25 }, // use this pronunciation when its a
noun
 { "$past",       0x26 }, // use this pronunciation when its past
tense
 { "$verbextend", 0x28 }, // extend influence of 'verb follows'
 { "$capital",    0x29 }, // use this pronunciation if initial
letter is upper case
 { "$allcaps",    0x2a }, // use this pronunciation if initial
letter is upper case
 { "$accent",     0x2b }, // character name is base-character
name + accent name
 { "$sentence",   0x2d }, // only if this clause is a sentence
(i.e. terminator is {. ? !} not {, ; :}
 { "$only",       0x2e }, // only match on this word without
suffix
 { "$onlys",      0x2f }, // only match with none, or with 's'
suffix
 { "$stem",       0x30 }, // must have a suffix
 { "$atend",      0x31 }, // use this pronunciation if at end of
clause
 { "$atstart",    0x32 }, // use this pronunciation at start of
clause
 { "$native",     0x33 }, // not if we've switched translators

 // doesn't set dictionary_flags
 { "$?",           100 }, // conditional rule, followed by byte
giving the condition number

 { "$textmode",    200 },
 { "$phonememode", 201 },

 { NULL, -1 }
};
```

```
#define LEN_GROUP_NAME  12

typedef struct {
 char name[LEN_GROUP_NAME+1];
 unsigned int start;
 unsigned int length;
 int group3_ix;
} RGROUP;

void print_dictionary_flags(unsigned int *flags, char *buf, int
buf_len)
{
 int stress;
 int ix;
 const char *name;
 int len;
 int total = 0;

 buf[0] = 0;
 if ((stress = flags[0] & 0xf) != 0) {
  sprintf(buf, "%s", LookupMnemName(mnem_flags, stress + 0x40));
  total = strlen(buf);
  buf += total;
 }

 for (ix = 8; ix < 64; ix++) {
  if (((ix < 30) && (flags[0] & (1 << ix))) || ((ix >= 0x20) &&
(flags[1] & (1 << (ix-0x20))))) {
   name = LookupMnemName(mnem_flags, ix);
   len = strlen(name) + 1;
   total += len;
   if (total >= buf_len)
    continue;
   sprintf(buf, " %s", name);
   buf += len;
  }
```

```
 }
}

char *DecodeRule(const char *group_chars, int group_length, char
*rule, int control)
{
 // Convert compiled match template to ascii

 unsigned char rb;
 unsigned char c;
 char *p;
 char *p_end;
 int ix;
 int match_type;
 bool finished = false;
 int value;
 int linenum = 0;
 int flags;
 int suffix_char;
 int condition_num = 0;
 bool at_start = false;
 const char *name;
 char buf[200];
 char buf_pre[200];
 char suffix[20];
 static char output[80];

 static char symbols[] = {
  ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
  '&', '%', '+', '#', 'S', 'D', 'Z', 'A', 'L', '!',
  ' ', '@', '?', 'J', 'N', 'K', 'V', '?', 'T', 'X',
  '?', 'W'
 };

 static char symbols_lg[] = { 'A', 'B', 'C', 'H', 'F', 'G', 'Y'
};
```

```
match_type = 0;
buf_pre[0] = 0;

for (ix = 0; ix < group_length; ix++)
 buf[ix] = group_chars[ix];
buf[ix] = 0;

p = &buf[strlen(buf)];
while (!finished) {
 rb = *rule++;

 if (rb <= RULE_LINENUM) {
  switch (rb)
  {
  case 0:
  case RULE_PHONEMES:
   finished = true;
   break;
  case RULE_PRE_ATSTART:
   at_start = true;
   // fallthrough:
  case RULE_PRE:
   match_type = RULE_PRE;
   *p = 0;
   p = buf_pre;
   break;
  case RULE_POST:
   match_type = RULE_POST;
   *p = 0;
   strcat(buf, " (");
   p = &buf[strlen(buf)];
   break;
  case RULE_PH_COMMON:
   break;
  case RULE_CONDITION:
   // conditional rule, next byte gives condition number
   condition_num = *rule++;
```

```
  break;
 case RULE_LINENUM:
  value = (rule[1] & 0xff) - 1;
  linenum = (rule[0] & 0xff) - 1 + (value * 255);
  rule += 2;
  break;
 }
 continue;
}

if (rb == RULE_DOLLAR) {
 value = *rule++ & 0xff;
 if ((value != 0x01) || (control & FLAG_UNPRON_TEST)) {
  // TODO write the string backwards if in RULE_PRE
  p[0] = '$';
  name = LookupMnemName(mnem_rules, value);
  strcpy(&p[1], name);
  p += (strlen(name)+1);
 }
 c = ' ';
} else if (rb == RULE_ENDING) {
 static const char *flag_chars = "eipvdfq tba ";
 flags = ((rule[0] & 0x7f)<< 8) + (rule[1] & 0x7f);
 suffix_char = 'S';
 if (flags & (SUFX_P >> 8))
  suffix_char = 'P';
 sprintf(suffix, "%c%d", suffix_char, rule[2] & 0x7f);
 rule += 3;
 for (ix = 0; ix < 9; ix++) {
  if (flags & 1)
   sprintf(&suffix[strlen(suffix)], "%c", flag_chars[ix]);
  flags = (flags >> 1);
 }
 strcpy(p, suffix);
 p += strlen(suffix);
 c = ' ';
} else if (rb == RULE_LETTERGP)
```

```
  c = symbols_lg[*rule++ - 'A'];
 else if (rb == RULE_LETTERGP2) {
  value = *rule++ - 'A';
  if (value < 0)
   value += 256;
  p[0] = 'L';
  p[1] = (value / 10) + '0';
  c = (value % 10) + '0';

  if (match_type == RULE_PRE) {
   p[0] = c;
   c = 'L';
  }
  p += 2;
 } else if (rb <= RULE_LAST_RULE)
  c = symbols[rb];
 else if (rb == RULE_SPACE)
  c = '_';
 else
  c = rb;
 *p++ = c;
}

p = output;
p_end = p + sizeof(output) - 1;

if (linenum > 0) {
 sprintf(p, "%5d:\t", linenum);
 p += 7;
}
if (condition_num > 0) {
 sprintf(p, "?%d ", condition_num);
 p = &p[strlen(p)];
}
if (((ix = strlen(buf_pre)) > 0) || at_start) {
 if (at_start)
  *p++ = '_';
```

```
  while ((--ix >= 0) && (p < p_end-3))
   *p++ = buf_pre[ix];
  *p++ = ')';
  *p++ = ' ';
 }

 buf[p_end - p] = 0; // prevent overflow in output[]
 strcat(p, buf);
 ix = strlen(output);
 while (ix < 8)
  output[ix++] = ' ';
 output[ix] = 0;
 return output;
}

typedef enum
{
 LINE_PARSER_WORD = 0,
 LINE_PARSER_END_OF_WORD = 1,
 LINE_PARSER_MULTIPLE_WORDS = 2,
 LINE_PARSER_END_OF_WORDS = 3,
 LINE_PARSER_PRONUNCIATION = 4,
 LINE_PARSER_END_OF_PRONUNCIATION = 5,
} LINE_PARSER_STATES;

static int compile_line(char *linebuf, char *dict_line, int
n_dict_line, int *hash)
{
 // Compile a line in the language_list file
 unsigned char c;
 char *p;
 char *word;
 char *phonetic;
 char *phonetic_end;
 unsigned int ix;
 LINE_PARSER_STATES step;
 unsigned int n_flag_codes = 0;
```

```
 int flagnum;
 int flag_offset;
 int length;
 int multiple_words = 0;
 bool multiple_numeric_hyphen = false;
 char *multiple_string = NULL;
 char *multiple_string_end = NULL;

 int len_word;
 int len_phonetic;
 bool text_not_phonemes = false; // this word specifies
replacement text, not phonemes
 unsigned int wc;
 bool all_upper_case;

 char *mnemptr;
 unsigned char flag_codes[100];
 char encoded_ph[200];
 char bad_phoneme_str[4];
 int bad_phoneme;
 static char nullstring[] = { 0 };

 phonetic = word = nullstring;

 p = linebuf;

 step = LINE_PARSER_WORD;

 c = *p;
 while (c != '\n' && c != '\0') {
  c = *p;

  if ((c == '?') && (step == 0)) {
   // conditional rule, allow only if the numbered condition is
set for the voice
   flag_offset = 100;
```

```
 p++;
 if (*p == '!') {
  // allow only if the numbered condition is NOT set
  flag_offset = 132;
  p++;
 }

 ix = 0;
 if (IsDigit09(*p)) {
  ix += (*p-'0');
  p++;
 }
 if (IsDigit09(*p)) {
  ix = ix*10 + (*p-'0');
  p++;
 }
 flag_codes[n_flag_codes++] = ix + flag_offset;
 c = *p;
}

if ((c == '$') && isalnum(p[1])) {
 // read keyword parameter
 mnemptr = p;
 while (!isspace2(c = *p)) p++;
 *p = 0;

 flagnum = LookupMnem(mnem_flags, mnemptr);
 if (flagnum > 0) {
  if (flagnum == 200)
   text_mode = true;
  else if (flagnum == 201)
   text_mode = false;
  else if (flagnum == BITNUM_FLAG_TEXTMODE)
   text_not_phonemes = true;
  else
   flag_codes[n_flag_codes++] = flagnum;
 } else {
```

```
    fprintf(f_log, "%5d: Unknown keyword: %s\n", linenum,
mnemptr);
    error_count++;
   }
  }

  if ((c == '/') && (p[1] == '/') && (multiple_words == 0))
   c = '\n'; // "//" treat comment as end of line

  switch (step)
  {
  case LINE_PARSER_WORD:
   if (c == '(') {
    multiple_words = 1;
    word = p+1;
    step = LINE_PARSER_END_OF_WORD;
   } else if (!isspace2(c)) {
    word = p;
    step = LINE_PARSER_END_OF_WORD;
   }
   break;
  case LINE_PARSER_END_OF_WORD:
   if ((c == '-') && multiple_words) {
    if (IsDigit09(word[0]))
     multiple_numeric_hyphen = true;
    flag_codes[n_flag_codes++] = BITNUM_FLAG_HYPHENATED;
    c = ' ';
   }
   if (isspace2(c)) {
    p[0] = 0; // terminate english word

    if (multiple_words) {
     multiple_string = multiple_string_end = p+1;
     step = LINE_PARSER_MULTIPLE_WORDS;
    } else
     step = LINE_PARSER_END_OF_WORDS;
   } else if (c == ')') {
```

```
   if (multiple_words) {
    p[0] = 0;
    multiple_words = 0;
    step = LINE_PARSER_END_OF_WORDS;
   } else if (word[0] != '_') {
    fprintf(f_log, "%5d: Missing '('\n", linenum);
    error_count++;
    step = LINE_PARSER_END_OF_WORDS;
   }
  }
  break;
 case LINE_PARSER_MULTIPLE_WORDS:
  if (isspace2(c))
   multiple_words++;
  else if (c == ')') {
   p[0] = ' '; // terminate extra string
   multiple_string_end = p+1;
   step = LINE_PARSER_END_OF_WORDS;
  }
  break;
 case LINE_PARSER_END_OF_WORDS:
  if (!isspace2(c)) {
   phonetic = p;
   step = LINE_PARSER_PRONUNCIATION;
  }
  break;
 case LINE_PARSER_PRONUNCIATION:
  if (isspace2(c)) {
   phonetic_end = p;
   p[0] = 0; // terminate phonetic
   step = LINE_PARSER_END_OF_PRONUNCIATION;
  }
  break;
 case LINE_PARSER_END_OF_PRONUNCIATION:
  if (!isspace2(c)) {
   *phonetic_end = ' ';
   step = LINE_PARSER_PRONUNCIATION;
```

```
   }
   break;
  }
  p++;
 }

 if (word[0] == 0)
  return 0; // blank line

 if (text_mode)
  text_not_phonemes = true;

 if (text_not_phonemes) {
  if (word[0] == '_') {
   // This is a special word, used by eSpeak.  Translate this
into phonemes now
   strcat(phonetic, " "); // need a space to indicate word-
boundary

   // PROBLEM  vowel reductions are not applied to the translated
phonemes
   // condition rules are not applied
   TranslateWord(translator, phonetic, NULL, NULL);
   text_not_phonemes = false;
   strncpy0(encoded_ph, word_phonemes, N_WORD_BYTES-4);

   if ((word_phonemes[0] == 0) && (error_need_dictionary < 3)) {
    // the dictionary was not loaded, we need a second attempt
    error_need_dictionary++;
    fprintf(f_log, "%5d: Need to compile dictionary again\n",
linenum);
   }
  } else
   // this is replacement text, so don't encode as phonemes.
Restrict the length of the replacement word
   strncpy0(encoded_ph, phonetic, N_WORD_BYTES-4);
 } else {
```

```
  EncodePhonemes(phonetic, encoded_ph, &bad_phoneme);
  if (strchr(encoded_ph, phonSWITCH) != 0)
   flag_codes[n_flag_codes++] = BITNUM_FLAG_ONLY_S;  // don't
match on suffixes (except 's') when switching languages

  // check for errors in the phonemes codes
  if (bad_phoneme != 0) {
   // unrecognised phoneme, report error
   bad_phoneme_str[utf8_out(bad_phoneme, bad_phoneme_str)] = 0;
   fprintf(f_log, "%5d: Bad phoneme [%s] (U+%x) in: %s  %s\n",
linenum, bad_phoneme_str, bad_phoneme, word, phonetic);
   error_count++;
  }
 }

 if (text_not_phonemes != translator->langopts.textmode)
  flag_codes[n_flag_codes++] = BITNUM_FLAG_TEXTMODE;

 if (sscanf(word, "U+%x", &wc) == 1) {
  // Character code
  ix = utf8_out(wc, word);
  word[ix] = 0;
 } else if (word[0] != '_') {
  // convert to lower case, and note if the word is all-capitals
  int c2;

  all_upper_case = true;
  for (p = word;;) {
   // this assumes that the lower case char is the same length as
the upper case char
   // OK, except for Turkish "I", but use towlower() rather than
towlower2()
   ix = utf8_in(&c2, p);
   if (c2 == 0)
    break;
   if (iswupper(c2))
    utf8_out(towlower2(c2, translator), p);
```

```
   else
    all_upper_case = false;
   p += ix;
  }
  if (all_upper_case)
   flag_codes[n_flag_codes++] = BITNUM_FLAG_ALLCAPS;
 }

 len_word = strlen(word);

 if (translator->transpose_min > 0)
  len_word = TransposeAlphabet(translator, word);

 len_phonetic = strlen(encoded_ph);

 dict_line[1] = len_word; // bit 6 indicates whether the word has
been compressed
 len_word &= 0x3f;

 memcpy(&dict_line[2], word, len_word);

 if (len_phonetic == 0) {
  // no phonemes specified. set bit 7
  dict_line[1] |= 0x80;
  length = len_word + 2;
 } else {
  length = len_word + len_phonetic + 3;
  if (length < n_dict_line) {
   strcpy(&dict_line[(len_word)+2], encoded_ph);
  } else {
   fprintf(f_log, "%5d: Dictionary line length would overflow the
data buffer: %d\n", linenum, length);
   error_count++;
   // no phonemes specified. set bit 7
   dict_line[1] |= 0x80;
   length = len_word + 2;
  }
```

```
 }

 for (ix = 0; ix < n_flag_codes; ix++)
  dict_line[ix+length] = flag_codes[ix];
 length += n_flag_codes;

 if ((multiple_string != NULL) && (multiple_words > 0)) {
  if (multiple_words > 10) {
   fprintf(f_log, "%5d: Two many parts in a multi-word entry:
%d\n", linenum, multiple_words);
   error_count++;
  } else {
   dict_line[length++] = 80 + multiple_words;
   ix = multiple_string_end - multiple_string;
   if (multiple_numeric_hyphen)
    dict_line[length++] = ' ';   // ???
   memcpy(&dict_line[length], multiple_string, ix);
   length += ix;
  }
 }

 return length;
}

static void compile_dictlist_start(void)
{
 // initialise dictionary list
 int ix;
 char *p;
 char *p2;

 for (ix = 0; ix < N_HASH_DICT; ix++) {
  p = hash_chains[ix];
  while (p != NULL) {
   memcpy(&p2, p, sizeof(char *));
   free(p);
   p = p2;
```

```
  }
  hash_chains[ix] = NULL;
 }
}

static void compile_dictlist_end(FILE *f_out)
{
 // Write out the compiled dictionary list
 int hash;
 int length;
 char *p;

 for (hash = 0; hash < N_HASH_DICT; hash++) {
  p = hash_chains[hash];

  while (p != NULL) {
   length = *(uint8_t *)(p+sizeof(char *));
   fwrite(p+sizeof(char *), length, 1, f_out);
   memcpy(&p, p, sizeof(char *));
  }
  fputc(0, f_out);
 }
}

static int compile_dictlist_file(const char *path, const char
*filename)
{
 int length;
 int hash;
 char *p;
 int count = 0;
 FILE *f_in;
 char buf[200];
 char fname[sizeof(path_home)+45];
 char dict_line[256]; // length is uint8_t, so an entry can't
take up more than 256 bytes
```

```
 text_mode = false;

 // try with and without '.txt' extension
 sprintf(fname, "%s%s.txt", path, filename);
 if ((f_in = fopen(fname, "r")) == NULL) {
  sprintf(fname, "%s%s", path, filename);
  if ((f_in = fopen(fname, "r")) == NULL)
   return -1;
 }

 if (f_log != NULL)
  fprintf(f_log, "Compiling: '%s'\n", fname);

 linenum = 0;

 while (fgets(buf, sizeof(buf), f_in) != NULL) {
  linenum++;

  length = compile_line(buf, dict_line, sizeof(dict_line),
&hash);
  if (length == 0)  continue; // blank line

  p = (char *)malloc(length+sizeof(char *));
  if (p == NULL) {
   if (f_log != NULL) {
    fprintf(f_log, "Can't allocate memory\n");
    error_count++;
   }
   break;
  }

  memcpy(p, &hash_chains[hash], sizeof(char *));
  hash_chains[hash] = p;
  // NOTE: dict_line[0] is the entry length (0-255)
  memcpy(p+sizeof(char *), dict_line, length);
  count++;
 }
```

```
 if (f_log != NULL)
  fprintf(f_log, "\t%d entries\n", count);
 fclose(f_in);
 return 0;
}

static char rule_cond[80];
static char rule_pre[80];
static char rule_post[80];
static char rule_match[80];
static char rule_phonemes[80];
static char group_name[LEN_GROUP_NAME+1];
static int group3_ix;

#define N_RULES 3000 // max rules for each group

static int isHexDigit(int c)
{
 if ((c >= '0') && (c <= '9'))
  return c - '0';
 if ((c >= 'a') && (c <= 'f'))
  return c - 'a' + 10;
 if ((c >= 'A') && (c <= 'F'))
  return c - 'A' + 10;
 return -1;
}

static void copy_rule_string(char *string, int *state_out)
{
 // state 0: conditional, 1=pre, 2=match, 3=post, 4=phonemes
 static char *outbuf[5] = { rule_cond, rule_pre, rule_match,
rule_post, rule_phonemes };
 static int next_state[5] = { 2, 2, 4, 4, 4 };
 char *output;
 char *p;
 int ix;
```

```
 int len;
 char c;
 int c2, c3;
 int sxflags;
 int value;
 bool literal;
 bool hexdigit_input = false;
 int state = *state_out;
 MNEM_TAB *mr;

 if (string[0] == 0) return;

 output = outbuf[state];
 if (state == 4) {
  // append to any previous phoneme string, i.e. allow spaces in
the phoneme string
  len = strlen(rule_phonemes);
  if (len > 0)
   rule_phonemes[len++] = ' ';
  output = &rule_phonemes[len];
 }
 sxflags = 0x808000; // to ensure non-zero bytes

 for (p = string, ix = 0;;) {
  literal = false;
  c = *p++;
  if ((c == '0') && (p[0] == 'x') && (isHexDigit(p[1]) >= 0) &&
(isHexDigit(p[2]) >= 0)) {
   hexdigit_input = true;
   c = p[1];
   p += 2;
  }
  if (c == '\\') {
   c = *p++; // treat next character literally
   if ((c >= '0') && (c <= '3') && (p[0] >= '0') && (p[0] <= '7')
&& (p[1] >= '0') && (p[1] <= '7')) {
    // character code given by 3 digit octal value;
```

```
    c = (c-'0')*64 + (p[0]-'0')*8 + (p[1]-'0');
    p += 2;
   }
   literal = true;
  }
  if (hexdigit_input) {
   if (((c2 = isHexDigit(c)) >= 0) && ((c3 = isHexDigit(p[0])) >=
0)) {
    c = c2 * 16 + c3;
    literal = true;
    p++;
   } else
    hexdigit_input = false;
  }
  if ((state == 1) || (state == 3)) {
   // replace special characters (note: 'E' is reserved for a
replaced silent 'e')
   if (literal == false) {
    static const char lettergp_letters[9] = { LETTERGP_A,
LETTERGP_B, LETTERGP_C, 0, 0, LETTERGP_F, LETTERGP_G, LETTERGP_H,
LETTERGP_Y };
    switch (c)
    {
    case '_':
     c = RULE_SPACE;
     break;

    case 'Y':
     c = 'I';
     // fallthrough:
    case 'A': // vowel
    case 'B':
    case 'C':
    case 'H':
    case 'F':
    case 'G':
     if (state == 1) {
```

```
      // pre-rule, put the number before the RULE_LETTERGP;
      output[ix++] = lettergp_letters[c-'A'] + 'A';
      c = RULE_LETTERGP;
     } else {
      output[ix++] = RULE_LETTERGP;
      c = lettergp_letters[c-'A'] + 'A';
     }
     break;
    case 'D':
     c = RULE_DIGIT;
     break;
    case 'K':
     c = RULE_NOTVOWEL;
     break;
    case 'N':
     c = RULE_NO_SUFFIX;
     break;
    case 'V':
     c = RULE_IFVERB;
     break;
    case 'Z':
     c = RULE_NONALPHA;
     break;
    case '+':
     c = RULE_INC_SCORE;
     break;
    case '<': // Can't use - as opposite for + because it is used
literally as part of word
     c = RULE_DEC_SCORE;
     break;
    case '@':
     c = RULE_SYLLABLE;
     break;
    case '&':
     c = RULE_STRESSED;
     break;
    case '%':
```

```
     c = RULE_DOUBLE;
     break;
    case '#':
     c = RULE_DEL_FWD;
     break;
    case '!':
     c = RULE_CAPITAL;
     break;
    case 'T':
     output[ix++] = RULE_DOLLAR;
     c = 0x11;
     break;
    case 'W':
     c = RULE_SPELLING;
     break;
    case 'X':
     c = RULE_NOVOWELS;
     break;
    case 'J':
     c = RULE_SKIPCHARS;
     break;
    case 'L':
     // expect two digits
     c = *p++ - '0';
     value = *p++ - '0';
     c = c * 10 + value;
     if ((value < 0) || (value > 9)) {
      c = 0;
      fprintf(f_log, "%5d: Expected 2 digits after 'L'\n",
linenum);
      error_count++;
     } else if ((c <= 0) || (c >= N_LETTER_GROUPS) ||
(letterGroupsDefined[(int)c] == 0)) {
      fprintf(f_log, "%5d: Letter group L%.2d not defined\n",
linenum, c);
      error_count++;
     }
```

```
    c += 'A';
    if (state == 1) {
     // pre-rule, put the group number before the RULE_LETTERGP
command
     output[ix++] = c;
     c = RULE_LETTERGP2;
    } else
     output[ix++] = RULE_LETTERGP2;
    break;
   case '$':
    value = 0;
    mr = mnem_rules;
    while (mr->mnem != NULL) {
     len = strlen(mr->mnem);
     if (memcmp(p, mr->mnem, len) == 0) {
      value = mr->value;
      p += len;
      break;
     }
     mr++;
    }

    if (state == 1) {
     // pre-rule, put the number before the RULE_DOLLAR
     output[ix++] = value;
     c = RULE_DOLLAR;
    } else {
     output[ix++] = RULE_DOLLAR;
     c = value;
    }

    if (value == 0) {
     fprintf(f_log, "%5d: $ command not recognized\n", linenum);
     error_count++;
    }
    break;
   case 'P': // Prefix
```

```
 sxflags |= SUFX_P;
 // fallthrough
case 'S': // Suffix
 output[ix++] = RULE_ENDING;
 value = 0;
 while (!isspace2(c = *p++) && (c != 0)) {
  switch (c)
  {
  case 'e':
   sxflags |= SUFX_E;
   break;
  case 'i':
   sxflags |= SUFX_I;
   break;
  case 'p': // obsolete, replaced by 'P' above
   sxflags |= SUFX_P;
   break;
  case 'v':
   sxflags |= SUFX_V;
   break;
  case 'd':
   sxflags |= SUFX_D;
   break;
  case 'f':
   sxflags |= SUFX_F;
   break;
  case 'q':
   sxflags |= SUFX_Q;
   break;
  case 't':
   sxflags |= SUFX_T;
   break;
  case 'b':
   sxflags |= SUFX_B;
   break;
  case 'a':
   sxflags |= SUFX_A;
```

```
      break;
     case 'm':
      sxflags |= SUFX_M;
      break;
     default:
      if (IsDigit09(c))
       value = (value*10) + (c - '0');
      break;
     }
    }
    p--;
    output[ix++] = sxflags >> 16;
    output[ix++] = sxflags >> 8;
    c = value | 0x80;
    break;
   }
  }
 }
 output[ix++] = c;
 if (c == 0) break;
}

}

static char *compile_rule(char *input)
{
 int ix;
 unsigned char c;
 int wc;
 char *p;
 char *prule;
 int len;
 int len_name;
 int start;
 int state = 2;
 bool finish = false;
 char buf[80];
```

```
 char output[150];
 int bad_phoneme;
 char bad_phoneme_str[4];

 buf[0] = 0;
 rule_cond[0] = 0;
 rule_pre[0] = 0;
 rule_post[0] = 0;
 rule_match[0] = 0;
 rule_phonemes[0] = 0;

 p = buf;

 for (ix = 0; finish == false; ix++) {
  switch (c = input[ix])
  {
  case ')': // end of prefix section
   *p = 0;
   state = 1;
   copy_rule_string(buf, &state);
   p = buf;
   break;
  case '(': // start of suffix section
   *p = 0;
   state = 2;
   copy_rule_string(buf, &state);
   state = 3;
   p = buf;
   if (input[ix+1] == ' ') {
    fprintf(f_log, "%5d: Syntax error. Space after (, or negative
score for previous rule\n", linenum);
    error_count++;
   }
   break;
  case '\n': // end of line
  case '\r':
  case 0:    // end of line
```

```
  *p = 0;
  copy_rule_string(buf, &state);
  finish = true;
  break;
 case '\t': // end of section section
 case ' ':
  *p = 0;
  copy_rule_string(buf, &state);
  p = buf;
  break;
 case '?':
  if (state == 2)
   state = 0;
  else
   *p++ = c;
  break;
 default:
  *p++ = c;
  break;
 }
}

if (strcmp(rule_match, "$group") == 0)
 strcpy(rule_match, group_name);

if (rule_match[0] == 0) {
 if (rule_post[0] != 0) {
  fprintf(f_log, "%5d: Syntax error\n", linenum);
  error_count++;
 }
 return NULL;
}

EncodePhonemes(rule_phonemes, buf, &bad_phoneme);
if (bad_phoneme != 0) {
 bad_phoneme_str[utf8_out(bad_phoneme, bad_phoneme_str)] = 0;
 fprintf(f_log, "%5d: Bad phoneme [%s] (U+%x) in: %s\n",
```

```
linenum, bad_phoneme_str, bad_phoneme, input);
  error_count++;
 }
 strcpy(output, buf);
 len = strlen(buf)+1;

 len_name = strlen(group_name);
 if ((len_name > 0) && (memcmp(rule_match, group_name, len_name)
!= 0)) {
  utf8_in(&wc, rule_match);
  if ((group_name[0] == '9') && IsDigit(wc)) {
   // numeric group, rule_match starts with a digit, so OK
  } else {
   fprintf(f_log, "%5d: Wrong initial letters '%s' for group
'%s'\n", linenum, rule_match, group_name);
   error_count++;
  }
 }
 strcpy(&output[len], rule_match);
 len += strlen(rule_match);

 if (debug_flag) {
  output[len] = RULE_LINENUM;
  output[len+1] = (linenum % 255) + 1;
  output[len+2] = (linenum / 255) + 1;
  len += 3;
 }

 if (rule_cond[0] != 0) {
  if (rule_cond[0] == '!') {
   // allow the rule only if the condition number is NOT set for
the voice
   ix = atoi(&rule_cond[1]) + 32;
  } else {
   // allow the rule only if the condition number is set for the
voice
   ix = atoi(rule_cond);
```

```
  }

  if ((ix > 0) && (ix < 255)) {
   output[len++] = RULE_CONDITION;
   output[len++] = ix;
  } else {
   fprintf(f_log, "%5d: bad condition number ?%d\n", linenum,
ix);
   error_count++;
  }
 }
 if (rule_pre[0] != 0) {
  start = 0;
  if (rule_pre[0] == RULE_SPACE) {
   // omit '_' at the beginning of the pre-string and imply it by
using RULE_PRE_ATSTART
   c = RULE_PRE_ATSTART;
   start = 1;
  } else
   c = RULE_PRE;
  output[len++] = c;

  // output PRE string in reverse order
  for (ix = strlen(rule_pre)-1; ix >= start; ix--)
   output[len++] = rule_pre[ix];
 }

 if (rule_post[0] != 0) {
  sprintf(&output[len], "%c%s", RULE_POST, rule_post);
  len += (strlen(rule_post)+1);
 }
 output[len++] = 0;
 if ((prule = (char *)malloc(len)) != NULL)
  memcpy(prule, output, len);
 return prule;
}
```

```
static int __cdecl string_sorter(char **a, char **b)
{
 char *pa, *pb;
 int ix;

 if ((ix = strcmp(pa = *a, pb = *b)) != 0)
  return ix;
 pa += (strlen(pa)+1);
 pb += (strlen(pb)+1);
 return strcmp(pa, pb);
}

static int __cdecl rgroup_sorter(RGROUP *a, RGROUP *b)
{
 // Sort long names before short names
 int ix;
 ix = strlen(b->name) - strlen(a->name);
 if (ix != 0) return ix;
 ix = strcmp(a->name, b->name);
 if (ix != 0) return ix;
 return a->start-b->start;
}

static void output_rule_group(FILE *f_out, int n_rules, char
**rules, char *name)
{
 int ix;
 int len1;
 int len2;
 int len_name;
 char *p;
 char *p2, *p3;
 const char *common;

 short nextchar_count[256];
 memset(nextchar_count, 0, sizeof(nextchar_count));
```

```
 len_name = strlen(name);

 // sort the rules in this group by their phoneme string
 common = "";
 qsort((void *)rules, n_rules, sizeof(char *), (int(__cdecl
*)(const void *, const void *))string_sorter);

 if (strcmp(name, "9") == 0)
  len_name = 0; //  don't remove characters from numeric match
strings

 for (ix = 0; ix < n_rules; ix++) {
  p = rules[ix];
  len1 = strlen(p) + 1; // phoneme string
  p3 = &p[len1];
  p2 = p3 + len_name; // remove group name from start of match
string
  len2 = strlen(p2);

  nextchar_count[(unsigned char)(p2[0])]++; // the next byte
after the group name

  if ((common[0] != 0) && (strcmp(p, common) == 0)) {
   fwrite(p2, len2, 1, f_out);
   fputc(0, f_out); // no phoneme string, it's the same as
previous rule
  } else {
   if ((ix < n_rules-1) && (strcmp(p, rules[ix+1]) == 0)) {
    common = rules[ix]; // phoneme string is same as next, set as
common
    fputc(RULE_PH_COMMON, f_out);
   }

   fwrite(p2, len2, 1, f_out);
   fputc(RULE_PHONEMES, f_out);
   fwrite(p, len1, 1, f_out);
  }
```

```
 }
}

static int compile_lettergroup(char *input, FILE *f_out)
{
 char *p;
 char *p_start;
 int group;
 int ix;
 int n_items;
 int length;
 int max_length = 0;

 #define N_LETTERGP_ITEMS 200
 char *items[N_LETTERGP_ITEMS];
 char item_length[N_LETTERGP_ITEMS];

 p = input;
 if (!IsDigit09(p[0]) || !IsDigit09(p[1])) {
  fprintf(f_log, "%5d: Expected 2 digits after '.L'\n", linenum);
  error_count++;
  return 1;
 }

 group = atoi(&p[0]);
 if (group >= N_LETTER_GROUPS) {
  fprintf(f_log, "%5d: lettergroup out of range (01-%.2d)\n",
linenum, N_LETTER_GROUPS-1);
  error_count++;
  return 1;
 }

 while (!isspace2(*p)) p++;

 fputc(RULE_GROUP_START, f_out);
 fputc(RULE_LETTERGP2, f_out);
 fputc(group + 'A', f_out);
```

```
 if (letterGroupsDefined[group] != 0) {
  fprintf(f_log, "%5d: lettergroup L%.2d is already defined\n",
linenum, group);
  error_count++;
 }
 letterGroupsDefined[group] = 1;

 n_items = 0;
 while (n_items < N_LETTERGP_ITEMS) {
  while (isspace2(*p)) p++;
  if (*p == 0)
   break;

  items[n_items] = p_start = p;
  while ((*p & 0xff) > ' ') {
   if (*p == '_') *p = ' '; // allow '_' for word break
   p++;
  }
  *p++ = 0;
  length = p - p_start;
  if (length > max_length)
   max_length = length;
  item_length[n_items++] = length;
 }

 // write out the items, longest first
 while (max_length > 1) {
  for (ix = 0; ix < n_items; ix++) {
   if (item_length[ix] == max_length)
    fwrite(items[ix], 1, max_length, f_out);
  }
  max_length--;
 }

 fputc(RULE_GROUP_END, f_out);

 return 0;
```

```
}

static void free_rules(char **rules, int n_rules)
{
 for (int i = 0; i < n_rules; ++i) {
  free(*rules);
  *rules++ = NULL;
 }
}

static espeak_ng_STATUS compile_dictrules(FILE *f_in, FILE
*f_out, char *fname_temp, espeak_ng_ERROR_CONTEXT *context)
{
 char *prule;
 unsigned char *p;
 int ix;
 int c;
 int gp;
 FILE *f_temp;
 int n_rules = 0;
 int count = 0;
 int different;
 int wc;
 int err_n_rules = 0;
 const char *prev_rgroup_name;
 unsigned int char_code;
 int compile_mode = 0;
 char *buf;
 char buf1[500];
 char *rules[N_RULES];

 int n_rgroups = 0;
 int n_groups3 = 0;
 RGROUP rgroup[N_RULE_GROUP2];

 linenum = 0;
 group_name[0] = 0;
```

```
 if ((f_temp = fopen(fname_temp, "wb")) == NULL)
  return create_file_error_context(context, errno, fname_temp);

 for (;;) {
  linenum++;
  buf = fgets(buf1, sizeof(buf1), f_in);
  if (buf != NULL) {
   if ((p = (unsigned char *)strstr(buf, "//")) != NULL)
    *p = 0;

   if (buf[0] == '\r') buf++; // ignore extra \r in \r\n
  }

  if ((buf == NULL) || (buf[0] == '.')) {
   // next .group or end of file, write out the previous group

   if (n_rules > 0) {
    strcpy(rgroup[n_rgroups].name, group_name);
    rgroup[n_rgroups].group3_ix = group3_ix;
    rgroup[n_rgroups].start = ftell(f_temp);
    output_rule_group(f_temp, n_rules, rules, group_name);
    rgroup[n_rgroups].length = ftell(f_temp) -
rgroup[n_rgroups].start;
    n_rgroups++;

    count += n_rules;
    free_rules(rules, n_rules);
   }
   n_rules = 0;
   err_n_rules = 0;

   if (compile_mode == 2) {
    // end of the character replacements section
    fwrite(&n_rules, 1, 4, f_out); // write a zero word to
terminate the replacemenmt list
    fputc(RULE_GROUP_END, f_out);
```

```
    compile_mode = 0;
   }

   if (buf == NULL) break; // end of file

   if (memcmp(buf, ".L", 2) == 0) {
    compile_lettergroup(&buf[2], f_out);
    continue;
   }

   if (memcmp(buf, ".replace", 8) == 0) {
    compile_mode = 2;
    fputc(RULE_GROUP_START, f_out);
    fputc(RULE_REPLACEMENTS, f_out);

    // advance to next word boundary
    while ((ftell(f_out) & 3) != 0)
     fputc(0, f_out);
   }

   if (memcmp(buf, ".group", 6) == 0) {
    compile_mode = 1;

    p = (unsigned char *)&buf[6];
    while ((p[0] == ' ') || (p[0] == '\t')) p++; // Note: Windows
isspace(0xe1) gives TRUE !
    ix = 0;
    while ((*p > ' ') && (ix < LEN_GROUP_NAME))
     group_name[ix++] = *p++;
    group_name[ix] = 0;
    group3_ix = 0;

    if (sscanf(group_name, "0x%x", &char_code) == 1) {
     // group character is given as a character code (max 16
bits)
     p = (unsigned char *)group_name;
```

```
     if (char_code > 0x100)
      *p++ = (char_code >> 8);
     *p++ = char_code;
     *p = 0;
    } else {
     if (translator->letter_bits_offset > 0) {
      utf8_in(&wc, group_name);
      if (((ix = (wc - translator->letter_bits_offset)) >= 0) &&
(ix < 128))
       group3_ix = ix+1; // not zero
     }
    }

    if ((group3_ix == 0) && (strlen(group_name) > 2)) {
     if (utf8_in(&c, group_name) < 2) {
      fprintf(f_log, "%5d: Group name longer than 2 bytes
(UTF8)", linenum);
      error_count++;
     }

     group_name[2] = 0;
    }
   }

   continue;
  }

  switch (compile_mode)
  {
  case 1: //  .group
   prule = compile_rule(buf);
   if (prule != NULL) {
    if (n_rules < N_RULES)
     rules[n_rules++] = prule;
    else {
     if (err_n_rules == 0) {
      fprintf(stderr, "\nExceeded limit of rules (%d) in group
```

```
'%s'\n", N_RULES, group_name);
      error_count++;
      err_n_rules = 1;
     }
    }

   }
   break;
  case 2: //  .replace
   p = (unsigned char *)buf;

   while (isspace2(*p)) p++;
   if ((unsigned char)(*p) > 0x20) {
    while ((unsigned char)(*p) > 0x20) { // not space or zero-
byte
     fputc(*p, f_out);
     p++;
    }
    fputc(0, f_out);

    while (isspace2(*p)) p++;
    while ((unsigned char)(*p) > 0x20) {
     fputc(*p, f_out);
     p++;
    }
    fputc(0, f_out);
   }
   break;
  }
 }
 fclose(f_temp);

 qsort((void *)rgroup, n_rgroups, sizeof(rgroup[0]), (int(__cdecl
*)(const void *, const void *))rgroup_sorter);

 if ((f_temp = fopen(fname_temp, "rb")) == NULL) {
  free_rules(rules, n_rules);
```

```
  return create_file_error_context(context, errno, fname_temp);
 }

 prev_rgroup_name = "\n";

 for (gp = 0; gp < n_rgroups; gp++) {
  fseek(f_temp, rgroup[gp].start, SEEK_SET);

  if ((different = strcmp(rgroup[gp].name, prev_rgroup_name)) !=
0) {
   // not the same as the previous group
   if (gp > 0)
    fputc(RULE_GROUP_END, f_out);
   fputc(RULE_GROUP_START, f_out);

   if (rgroup[gp].group3_ix != 0) {
    n_groups3++;
    fputc(1, f_out);
    fputc(rgroup[gp].group3_ix, f_out);
   } else
    fprintf(f_out, "%s", prev_rgroup_name = rgroup[gp].name);
   fputc(0, f_out);
  }

  for (ix = rgroup[gp].length; ix > 0; ix--) {
   c = fgetc(f_temp);
   fputc(c, f_out);
  }
 }
 fputc(RULE_GROUP_END, f_out);
 fputc(0, f_out);

 fclose(f_temp);
 remove(fname_temp);

 fprintf(f_log, "\t%d rules, %d groups (%d)\n\n", count,
n_rgroups, n_groups3);
```

```
 free_rules(rules, n_rules);
 return ENS_OK;
}

#pragma GCC visibility push(default)
ESPEAK_NG_API espeak_ng_STATUS espeak_ng_CompileDictionary(const
char *dsource, const char *dict_name, FILE *log, int flags,
espeak_ng_ERROR_CONTEXT *context)
{
 if (!log) log = stderr;
 if (!dict_name) dict_name = dictionary_name;

 // fname:  space to write the filename in case of error
 // flags: bit 0:  include source line number information, for
debug purposes.

 FILE *f_in;
 FILE *f_out;
 int offset_rules = 0;
 int value;
 char fname_in[sizeof(path_home)+45];
 char fname_out[sizeof(path_home)+15];
 char fname_temp[sizeof(path_home)+15];
 char path[sizeof(path_home)+40];       // path_dsource+20

 error_count = 0;
 error_need_dictionary = 0;
 memset(letterGroupsDefined, 0, sizeof(letterGroupsDefined));

 debug_flag = flags & 1;

 if (dsource == NULL)
  dsource = "";

 f_log = log;
 if (f_log == NULL)
  f_log = stderr;
```

```
 // try with and without '.txt' extension
 sprintf(path, "%s%s_", dsource, dict_name);
 sprintf(fname_in, "%srules.txt", path);
 if ((f_in = fopen(fname_in, "r")) == NULL) {
  sprintf(fname_in, "%srules", path);
  if ((f_in = fopen(fname_in, "r")) == NULL)
   return create_file_error_context(context, errno, fname_in);
 }

 sprintf(fname_out, "%s%c%s_dict", path_home, PATHSEP,
dict_name);
 if ((f_out = fopen(fname_out, "wb+")) == NULL) {
  int error = errno;
  fclose(f_in);
  return create_file_error_context(context, error, fname_out);
 }
 sprintf(fname_temp, "%s%ctemp", path_home, PATHSEP);

 value = N_HASH_DICT;
 Write4Bytes(f_out, value);
 Write4Bytes(f_out, offset_rules);

 compile_dictlist_start();

 fprintf(f_log, "Using phonemetable: '%s'\n",
phoneme_tab_list[phoneme_tab_number].name);
 compile_dictlist_file(path, "roots");
 if (translator->langopts.listx) {
  compile_dictlist_file(path, "list");
  compile_dictlist_file(path, "listx");
 } else {
  compile_dictlist_file(path, "listx");
  compile_dictlist_file(path, "list");
 }
 compile_dictlist_file(path, "emoji");
 compile_dictlist_file(path, "extra");
```

```
 compile_dictlist_end(f_out);
 offset_rules = ftell(f_out);

 fprintf(f_log, "Compiling: '%s'\n", fname_in);

 espeak_ng_STATUS status = compile_dictrules(f_in, f_out,
fname_temp, context);
 fclose(f_in);

 fseek(f_out, 4, SEEK_SET);
 Write4Bytes(f_out, offset_rules);
 fclose(f_out);
 fflush(f_log);

 if (status != ENS_OK)
  return status;

 LoadDictionary(translator, dict_name, 0);

 return error_count > 0 ? ENS_COMPILE_ERROR : ENS_OK;
}
#pragma GCC visibility pop
```

Chapter 50

./src/libespeak-ng/espeak_api.c

```
#include "config.h"

#include <stdint.h>
#include <stdlib.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "compiledict.h"

#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"
#include "event.h"

static espeak_ERROR status_to_espeak_error(espeak_ng_STATUS
status)
{
 switch (status)
 {
```

```
 case ENS_OK:                        return EE_OK;
 case ENS_SPEECH_STOPPED:            return EE_OK;
 case ENS_VOICE_NOT_FOUND:           return EE_NOT_FOUND;
 case ENS_MBROLA_NOT_FOUND:          return EE_NOT_FOUND;
 case ENS_MBROLA_VOICE_NOT_FOUND: return EE_NOT_FOUND;
 case ENS_FIFO_BUFFER_FULL:          return EE_BUFFER_FULL;
 default:                            return EE_INTERNAL_ERROR;
 }
}

#pragma GCC visibility push(default)

ESPEAK_API int espeak_Initialize(espeak_AUDIO_OUTPUT output_type,
int buf_length, const char *path, int options)
{
 espeak_ng_InitializePath(path);
 espeak_ng_ERROR_CONTEXT context = NULL;
 espeak_ng_STATUS result = espeak_ng_Initialize(&context);
 if (result != ENS_OK) {
  espeak_ng_PrintStatusCodeMessage(result, stderr, context);
  espeak_ng_ClearErrorContext(&context);
  if ((options & espeakINITIALIZE_DONT_EXIT) == 0)
   exit(1);
 }

 switch (output_type)
 {
 case AUDIO_OUTPUT_PLAYBACK:
  espeak_ng_InitializeOutput(ENOUTPUT_MODE_SPEAK_AUDIO,
buf_length, NULL);
  break;
 case AUDIO_OUTPUT_RETRIEVAL:
  espeak_ng_InitializeOutput(0, buf_length, NULL);
  break;
 case AUDIO_OUTPUT_SYNCHRONOUS:
  espeak_ng_InitializeOutput(ENOUTPUT_MODE_SYNCHRONOUS,
buf_length, NULL);
```

```
  break;
 case AUDIO_OUTPUT_SYNCH_PLAYBACK:
  espeak_ng_InitializeOutput(ENOUTPUT_MODE_SYNCHRONOUS |
ENOUTPUT_MODE_SPEAK_AUDIO, buf_length, NULL);
  break;
 }

 option_phoneme_events = (options &
(espeakINITIALIZE_PHONEME_EVENTS |
espeakINITIALIZE_PHONEME_IPA));

 return espeak_ng_GetSampleRate();
}

ESPEAK_API espeak_ERROR espeak_Synth(const void *text, size_t
size,
                                     unsigned int position,
                                     espeak_POSITION_TYPE
position_type,
                                     unsigned int end_position,
unsigned int flags,
                                     unsigned int
*unique_identifier, void *user_data)
{
 return status_to_espeak_error(espeak_ng_Synthesize(text, size,
position, position_type, end_position, flags, unique_identifier,
user_data));
}

ESPEAK_API espeak_ERROR espeak_Synth_Mark(const void *text,
size_t size,
                                          const char *index_mark,
                                          unsigned int
end_position,
                                          unsigned int flags,
                                          unsigned int
*unique_identifier,
```

```
                                          void *user_data)
{
 return status_to_espeak_error(espeak_ng_SynthesizeMark(text,
size, index_mark, end_position, flags, unique_identifier,
user_data));
}

ESPEAK_API espeak_ERROR espeak_Key(const char *key_name)
{
 return status_to_espeak_error(espeak_ng_SpeakKeyName(key_name));
}

ESPEAK_API espeak_ERROR espeak_Char(wchar_t character)
{
 return
status_to_espeak_error(espeak_ng_SpeakCharacter(character));
}

ESPEAK_API espeak_ERROR espeak_SetParameter(espeak_PARAMETER
parameter, int value, int relative)
{
 return status_to_espeak_error(espeak_ng_SetParameter(parameter,
value, relative));
}

ESPEAK_API espeak_ERROR espeak_SetPunctuationList(const wchar_t
*punctlist)
{
 return
status_to_espeak_error(espeak_ng_SetPunctuationList(punctlist));
}

ESPEAK_API espeak_ERROR espeak_SetVoiceByName(const char *name)
{
 return status_to_espeak_error(espeak_ng_SetVoiceByName(name));
}
```

```
ESPEAK_API espeak_ERROR espeak_SetVoiceByFile(const char
*filename)
{
 return
status_to_espeak_error(espeak_ng_SetVoiceByFile(filename));
}

ESPEAK_API espeak_ERROR espeak_SetVoiceByProperties(espeak_VOICE
*voice_selector)
{
 return status_to_espeak_error(espeak_ng_SetVoiceByProperties(voi
ce_selector));
}

ESPEAK_API espeak_ERROR espeak_Cancel(void)
{
 return status_to_espeak_error(espeak_ng_Cancel());
}

ESPEAK_API espeak_ERROR espeak_Synchronize(void)
{
 return status_to_espeak_error(espeak_ng_Synchronize());
}

ESPEAK_API espeak_ERROR espeak_Terminate(void)
{
 return status_to_espeak_error(espeak_ng_Terminate());
}

ESPEAK_API void espeak_CompileDictionary(const char *path, FILE
*log, int flags)
{
 espeak_ng_ERROR_CONTEXT context = NULL;
 espeak_ng_STATUS result = espeak_ng_CompileDictionary(path,
dictionary_name, log, flags, &context);
 if (result != ENS_OK) {
  espeak_ng_PrintStatusCodeMessage(result, stderr, context);
```

```
  espeak_ng_ClearErrorContext(&context);
 }
}

#pragma GCC visibility pop
```

Chapter 51

./src/libespeak-ng/voices.c

```
#include "config.h"

#include <ctype.h>
#include <wctype.h>
#include <errno.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

#if defined(_WIN32) || defined(_WIN64)
#include <windows.h>
#else
#include <dirent.h>
#endif

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "dictionary.h"
```

```
#include "readclause.h"
#include "synthdata.h"
#include "wavegen.h"

#include "speech.h"
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

MNEM_TAB genders[] = {
 { "male", ENGENDER_MALE },
 { "female", ENGENDER_FEMALE },
 { NULL, ENGENDER_MALE }
};

int tone_points[12] = { 600, 170, 1200, 135, 2000, 110, 3000,
110, -1, 0 };

// limit the rate of change for each formant number
static int formant_rate_22050[9] = { 240, 170, 170, 170, 170,
170, 170, 170, 170 }; // values for 22kHz sample rate
int formant_rate[9]; // values adjusted for actual sample rate

#define DEFAULT_LANGUAGE_PRIORITY  5
#define N_VOICES_LIST  300
static int n_voices_list = 0;
static espeak_VOICE *voices_list[N_VOICES_LIST];

espeak_VOICE current_voice_selected;

enum {
 V_NAME = 1,
 V_LANGUAGE,
 V_GENDER,
 V_TRANSLATOR,
 V_PHONEMES,
```

```
V_DICTIONARY,
V_VARIANTS,

V_MAINTAINER,
V_STATUS,

// these affect voice quality, are independent of language
V_FORMANT,
V_PITCH,
V_ECHO,
V_FLUTTER,
V_ROUGHNESS,
V_CLARITY,
V_TONE,
V_VOICING,
V_BREATH,
V_BREATHW,

// these override defaults set by the translator
V_WORDGAP,
V_INTONATION,
V_TUNES,
V_STRESSLENGTH,
V_STRESSAMP,
V_STRESSADD,
V_DICTRULES,
V_STRESSRULE,
V_STRESSOPT,
V_NUMBERS,
V_OPTION,

V_MBROLA,
V_KLATT,
V_FAST,
V_SPEED,
V_DICTMIN,
V_ALPHABET2,
```

```
 // these need a phoneme table to have been specified
 V_REPLACE,
 V_CONSONANTS
};

static MNEM_TAB options_tab[] = {
 { "reduce_t", LOPT_REDUCE_T },
 { "bracket",  LOPT_BRACKET_PAUSE },
 { NULL,   -1 }
};

static MNEM_TAB keyword_tab[] = {
 { "name",         V_NAME },
 { "language",     V_LANGUAGE },
 { "gender",       V_GENDER },

 { "maintainer",   V_MAINTAINER },
 { "status",       V_STATUS },

 { "variants",     V_VARIANTS },
 { "formant",      V_FORMANT },
 { "pitch",        V_PITCH },
 { "phonemes",     V_PHONEMES },
 { "translator",   V_TRANSLATOR },
 { "dictionary",   V_DICTIONARY },
 { "stressLength", V_STRESSLENGTH },
 { "stressAmp",    V_STRESSAMP },
 { "stressAdd",    V_STRESSADD },
 { "intonation",   V_INTONATION },
 { "tunes",        V_TUNES },
 { "dictrules",    V_DICTRULES },
 { "stressRule",   V_STRESSRULE },
 { "stressopt",    V_STRESSOPT },
 { "replace",      V_REPLACE },
 { "words",        V_WORDGAP },
 { "echo",         V_ECHO },
```

```
 { "flutter",      V_FLUTTER },
 { "roughness",    V_ROUGHNESS },
 { "clarity",      V_CLARITY },
 { "tone",         V_TONE },
 { "voicing",      V_VOICING },
 { "breath",       V_BREATH },
 { "breathw",      V_BREATHW },
 { "numbers",      V_NUMBERS },
 { "option",       V_OPTION },
 { "mbrola",       V_MBROLA },
 { "consonants",   V_CONSONANTS },
 { "klatt",        V_KLATT },
 { "fast_test2",   V_FAST },
 { "speed",        V_SPEED },
 { "dict_min",     V_DICTMIN },

 // these just set a value in langopts.param[]
 { "l_dieresis",       0x100+LOPT_DIERESES },
 { "l_prefix",         0x100+LOPT_PREFIXES },
 { "l_regressive_v",   0x100+LOPT_REGRESSIVE_VOICING },
 { "l_unpronouncable", 0x100+LOPT_UNPRONOUNCABLE },
 { "l_sonorant_min",   0x100+LOPT_SONORANT_MIN },
 { "l_length_mods",    0x100+LOPT_LENGTH_MODS },
 { "apostrophe",       0x100+LOPT_APOSTROPHE },

 { NULL, 0 }
};

#define N_VOICE_VARIANTS   12
const char variants_either[N_VOICE_VARIANTS] = { 1, 2, 12, 3, 13,
4, 14, 5, 11, 0 };
const char variants_male[N_VOICE_VARIANTS] = { 1, 2, 3, 4, 5, 6,
0 };
const char variants_female[N_VOICE_VARIANTS] = { 11, 12, 13, 14,
0 };
const char *variant_lists[3] = { variants_either, variants_male,
variants_female };
```

```
static voice_t voicedata;
voice_t *voice = &voicedata;

static char *fgets_strip(char *buf, int size, FILE *f_in)
{
 // strip trailing spaces, and truncate lines at // comment
 int len;
 char *p;

 if (fgets(buf, size, f_in) == NULL)
  return NULL;

 if (buf[0] == '#') {
  buf[0] = 0;
  return buf;
 }

 len = strlen(buf);
 while ((--len > 0) && isspace(buf[len]))
  buf[len] = 0;

 if ((p = strstr(buf, "//")) != NULL)
  *p = 0;

 return buf;
}

static int LookupTune(const char *name)
{
 int ix;

 for (ix = 0; ix < n_tunes; ix++) {
  if (strcmp(name, tunes[ix].name) == 0)
   return ix;
 }
 return -1;
```

```
}

static void SetToneAdjust(voice_t *voice, int *tone_pts)
{
 int ix;
 int pt;
 int y;
 int freq1 = 0;
 int freq2;
 int height1 = tone_pts[1];
 int height2;
 double rate;

 for (pt = 0; pt < 12; pt += 2) {
  if (tone_pts[pt] == -1) {
   tone_pts[pt] = N_TONE_ADJUST*8;
   if (pt > 0)
    tone_pts[pt+1] = tone_pts[pt-1];
  }
  freq2 = tone_pts[pt] / 8; // 8Hz steps
  height2 = tone_pts[pt+1];
  if ((freq2 - freq1) > 0) {
   rate = (double)(height2-height1)/(freq2-freq1);

   for (ix = freq1; ix < freq2; ix++) {
    y = height1 + (int)(rate * (ix-freq1));
    if (y > 255)
     y = 255;
    voice->tone_adjust[ix] = y;
   }
  }
  freq1 = freq2;
  height1 = height2;
 }
}

void ReadTonePoints(char *string, int *tone_pts)
```

```
{
 // tone_pts[] is int[12]
 int ix;

 for (ix = 0; ix < 12; ix++)
  tone_pts[ix] = -1;

 sscanf(string, "%d %d %d %d %d %d %d %d %d %d",
        &tone_pts[0], &tone_pts[1], &tone_pts[2], &tone_pts[3],
        &tone_pts[4], &tone_pts[5], &tone_pts[6], &tone_pts[7],
        &tone_pts[8], &tone_pts[9]);
}

static espeak_VOICE *ReadVoiceFile(FILE *f_in, const char *fname,
int is_language_file)
{
 // Read a Voice file, allocate a VOICE_DATA and set data from
the
 // file's  language, gender, name  lines

 char linebuf[120];
 char vname[80];
 char vgender[80];
 char vlanguage[80];
 char languages[300]; // allow space for several alternate
language names and priorities

 unsigned int len;
 int langix = 0;
 int n_languages = 0;
 char *p;
 espeak_VOICE *voice_data;
 int priority;
 int age;
 int n_variants = 4; // default, number of variants of this voice
before using another voice
 int gender;
```

```
vname[0] = 0;
vgender[0] = 0;
age = 0;

while (fgets_strip(linebuf, sizeof(linebuf), f_in) != NULL) {
 // isolate the attribute name
 for (p = linebuf; (*p != 0) && !iswspace(*p); p++) ;
 *p++ = 0;

 if (linebuf[0] == 0) continue;

 switch (LookupMnem(keyword_tab, linebuf))
 {
 case V_NAME:
  while (isspace(*p)) p++;
  strncpy0(vname, p, sizeof(vname));
  break;
 case V_LANGUAGE:
  priority = DEFAULT_LANGUAGE_PRIORITY;
  vlanguage[0] = 0;

  sscanf(p, "%s %d", vlanguage, &priority);
  len = strlen(vlanguage) + 2;
  // check for space in languages[]
  if (len < (sizeof(languages)-langix-1)) {
   languages[langix] = priority;

   strcpy(&languages[langix+1], vlanguage);
   langix += len;
   n_languages++;
  }
  break;
 case V_GENDER:
  sscanf(p, "%s %d", vgender, &age);
  if (is_language_file)
   fprintf(stderr, "Error (%s): gender attribute specified on a
```

```
language file\n", fname);
   break;
  case V_VARIANTS:
   sscanf(p, "%d", &n_variants);
  }
 }
 languages[langix++] = 0;

 gender = LookupMnem(genders, vgender);

 if (n_languages == 0)
  return NULL; // no language lines in the voice file

 p = (char *)calloc(sizeof(espeak_VOICE) + langix + strlen(fname)
+ strlen(vname) + 3, 1);
 voice_data = (espeak_VOICE *)p;
 p = &p[sizeof(espeak_VOICE)];

 memcpy(p, languages, langix);
 voice_data->languages = p;

 strcpy(&p[langix], fname);
 voice_data->identifier = &p[langix];
 voice_data->name = &p[langix];

 if (vname[0] != 0) {
  langix += strlen(fname)+1;
  strcpy(&p[langix], vname);
  voice_data->name = &p[langix];
 }

 voice_data->age = age;
 voice_data->gender = gender;
 voice_data->variant = 0;
 voice_data->xx1 = n_variants;
 return voice_data;
}
```

```
void VoiceReset(int tone_only)
{
 // Set voice to the default values

 int pk;
 static unsigned char default_heights[N_PEAKS] = { 130, 128, 120,
116, 100, 100, 128, 128, 128 }; // changed for v.1.47
 static unsigned char default_widths[N_PEAKS] = { 140, 128, 128,
160, 171, 171, 128, 128, 128 };

 static int breath_widths[N_PEAKS] = { 0, 200, 200, 400, 400,
400, 600, 600, 600 };

 // default is:  pitch 80,118
 voice->pitch_base = 0x47000;
 voice->pitch_range = 4104;

 voice->formant_factor = 256;

 voice->speed_percent = 100;
 voice->echo_delay = 0;
 voice->echo_amp = 0;
 voice->flutter = 64;
 voice->n_harmonic_peaks = 5;
 voice->peak_shape = 0;
 voice->voicing = 64;
 voice->consonant_amp = 90; // change from 100 to 90 for v.1.47
 voice->consonant_ampv = 100;
 voice->samplerate = samplerate_native;
 memset(voice->klattv, 0, sizeof(voice->klattv));

 speed.fast_settings[0] = espeakRATE_MAXIMUM;
 speed.fast_settings[1] = 800;
 speed.fast_settings[2] = espeakRATE_NORMAL;

 voice->roughness = 2;
```

```
 InitBreath();
 for (pk = 0; pk < N_PEAKS; pk++) {
  voice->freq[pk] = 256;
  voice->height[pk] = default_heights[pk]*2;
  voice->width[pk] = default_widths[pk]*2;
  voice->breath[pk] = 0;
  voice->breathw[pk] = breath_widths[pk]; // default breath
formant widths
  voice->freqadd[pk] = 0;

  // adjust formant smoothing depending on sample rate
  formant_rate[pk] = (formant_rate_22050[pk] * 22050)/samplerate;
 }

 // This table provides the opportunity for tone control.
 // Adjustment of harmonic amplitudes, steps of 8Hz
 // value of 128 means no change
 SetToneAdjust(voice, tone_points);

 // default values of speed factors
 voice->speedf1 = 256;
 voice->speedf2 = 238;
 voice->speedf3 = 232;

 if (tone_only == 0) {
  n_replace_phonemes = 0;
  LoadMbrolaTable(NULL, NULL, 0);
 }
}

static void VoiceFormant(char *p)
{
 // Set parameters for a formant
 int ix;
 int formant;
 int freq = 100;
```

```
 int height = 100;
 int width = 100;
 int freqadd = 0;

 ix = sscanf(p, "%d %d %d %d %d", &formant, &freq, &height,
&width, &freqadd);
 if (ix < 2)
  return;

 if ((formant < 0) || (formant > 8))
  return;

 if (freq >= 0)
  voice->freq[formant] = (int)(freq * 2.56001);
 if (height >= 0)
  voice->height[formant] = (int)(height * 2.56001);
 if (width >= 0)
  voice->width[formant] = (int)(width * 2.56001);
 voice->freqadd[formant] = freqadd;
}

static void PhonemeReplacement(char *p)
{
 int n;
 int phon;
 int flags = 0;
 char phon_string1[12];
 char phon_string2[12];

 strcpy(phon_string2, "NULL");
 n = sscanf(p, "%d %s %s", &flags, phon_string1, phon_string2);
 if ((n < 2) || (n_replace_phonemes >= N_REPLACE_PHONEMES))
  return;

 if ((phon = LookupPhonemeString(phon_string1)) == 0)
  return; // not recognised
```

```
 replace_phonemes[n_replace_phonemes].old_ph = phon;
 replace_phonemes[n_replace_phonemes].new_ph =
LookupPhonemeString(phon_string2);
 replace_phonemes[n_replace_phonemes++].type = flags;
}

static int Read8Numbers(char *data_in, int *data)
{
 // Read 8 integer numbers
 memset(data, 0, 8+sizeof(int));
 return sscanf(data_in, "%d %d %d %d %d %d %d %d",
               &data[0], &data[1], &data[2], &data[3], &data[4],
&data[5], &data[6], &data[7]);
}

voice_t *LoadVoice(const char *vname, int control)
{
 // control, bit 0  1= no_default
 //          bit 1  1 = change tone only, not language
 //          bit 2  1 = don't report error on LoadDictionary
 //          bit 4  1 = vname = full path

 FILE *f_voice = NULL;
 char *p;
 int key;
 int ix;
 int n;
 int value;
 int value2;
 int langix = 0;
 int tone_only = control & 2;
 bool language_set = false;
 bool phonemes_set = false;
 int stress_amps_set = 0;
 int stress_lengths_set = 0;
 int stress_add_set = 0;
 int conditional_rules = 0;
```

```
 LANGUAGE_OPTIONS *langopts = NULL;

 Translator *new_translator = NULL;

 char voicename[40];
 char language_name[40];
 char translator_name[40];
 char new_dictionary[40];
 char phonemes_name[40];
 char option_name[40];
 const char *language_type;
 char buf[sizeof(path_home)+30];
 char path_voices[sizeof(path_home)+12];

 int dict_min = 0;
 int stress_amps[8];
 int stress_lengths[8];
 int stress_add[8];
 char names[8][40];
 char name1[40];
 char name2[80];

 int pitch1;
 int pitch2;

 static char voice_identifier[40]; // file name for
current_voice_selected
 static char voice_name[40];       // voice name for
current_voice_selected
 static char voice_languages[100]; // list of languages and
priorities for current_voice_selected

 strncpy0(voicename, vname, sizeof(voicename));
 if (control & 0x10) {
  strcpy(buf, vname);
  if (GetFileLength(buf) <= 0)
   return NULL;
```

```
 } else {
  if (voicename[0] == 0)
   strcpy(voicename, ESPEAKNG_DEFAULT_VOICE);

  sprintf(path_voices, "%s%cvoices%c", path_home, PATHSEP,
PATHSEP);
  sprintf(buf, "%s%s", path_voices, voicename); // look in the
main voices directory

  if (GetFileLength(buf) <= 0) {
   sprintf(path_voices, "%s%clang%c", path_home, PATHSEP,
PATHSEP);
   sprintf(buf, "%s%s", path_voices, voicename); // look in the
main languages directory
  }
 }

 f_voice = fopen(buf, "r");

 language_type = "en"; // default
 if (f_voice == NULL) {
  if (control & 3)
   return NULL; // can't open file

  if (SelectPhonemeTableName(voicename) >= 0)
   language_type = voicename;
 }

 if (!tone_only && (translator != NULL)) {
  DeleteTranslator(translator);
  translator = NULL;
 }

 strcpy(translator_name, language_type);
 strcpy(new_dictionary, language_type);
 strcpy(phonemes_name, language_type);
```

```
 if (!tone_only) {
  voice = &voicedata;
  strncpy0(voice_identifier, vname, sizeof(voice_identifier));
  voice_name[0] = 0;
  voice_languages[0] = 0;

  current_voice_selected.identifier = voice_identifier;
  current_voice_selected.name = voice_name;
  current_voice_selected.languages = voice_languages;
 } else {
  // append the variant file name to the voice identifier
  if ((p = strchr(voice_identifier, '+')) != NULL)
   *p = 0;    // remove previous variant name
  sprintf(buf, "+%s", &vname[3]);    // omit  !v/  from the
variant filename
  strcat(voice_identifier, buf);
  langopts = &translator->langopts;
 }
 VoiceReset(tone_only);

 if (!tone_only)
  SelectPhonemeTableName(phonemes_name); // set up phoneme_tab

 while ((f_voice != NULL) && (fgets_strip(buf, sizeof(buf),
f_voice) != NULL)) {
  // isolate the attribute name
  for (p = buf; (*p != 0) && !isspace(*p); p++) ;
  *p++ = 0;

  if (buf[0] == 0) continue;

  key = LookupMnem(keyword_tab, buf);

  switch (key)
  {
  case V_LANGUAGE:
  {
```

```
unsigned int len;
int priority;

if (tone_only)
 break;

priority = DEFAULT_LANGUAGE_PRIORITY;
language_name[0] = 0;

sscanf(p, "%s %d", language_name, &priority);
if (strcmp(language_name, "variant") == 0)
 break;

len = strlen(language_name) + 2;
// check for space in languages[]
if (len < (sizeof(voice_languages)-langix-1)) {
 voice_languages[langix] = priority;

 strcpy(&voice_languages[langix+1], language_name);
 langix += len;
}

// only act on the first language line
if (language_set == false) {
 language_type = strtok(language_name, "-");
 language_set = true;
 strcpy(translator_name, language_type);
 strcpy(new_dictionary, language_type);
 strcpy(phonemes_name, language_type);
 SelectPhonemeTableName(phonemes_name);

 if (new_translator != NULL)
  DeleteTranslator(new_translator);

 new_translator = SelectTranslator(translator_name);
 langopts = &new_translator->langopts;
 strncpy0(voice->language_name, language_name,
```

```
sizeof(voice->language_name));
   }
  }
   break;
  case V_NAME:
   if (tone_only == 0) {
    while (isspace(*p)) p++;
    strncpy0(voice_name, p, sizeof(voice_name));
   }
   break;
  case V_GENDER:
  {
   int age = 0;
   char vgender[80];
   sscanf(p, "%s %d", vgender, &age);
   current_voice_selected.gender = LookupMnem(genders, vgender);
   current_voice_selected.age = age;
  }
   break;
  case V_TRANSLATOR:
   if (tone_only) break;

   sscanf(p, "%s", translator_name);

   if (new_translator != NULL)
    DeleteTranslator(new_translator);

   new_translator = SelectTranslator(translator_name);
   langopts = &new_translator->langopts;
   break;
  case V_DICTIONARY: // dictionary
   sscanf(p, "%s", new_dictionary);
   break;
  case V_PHONEMES: // phoneme table
   sscanf(p, "%s", phonemes_name);
   break;
  case V_FORMANT:
```

```
   VoiceFormant(p);
   break;
  case V_PITCH:
   // default is  pitch 82 118
   if (sscanf(p, "%d %d", &pitch1, &pitch2) == 2) {
    voice->pitch_base = (pitch1 - 9) << 12;
    voice->pitch_range = (pitch2 - pitch1) * 108;
    double factor = (double)(pitch1 - 82)/82;
    voice->formant_factor = (int)((1+factor/4) * 256); // nominal
formant shift for a different voice pitch
   }
   break;
  case V_STRESSLENGTH: // stressLength
   stress_lengths_set = Read8Numbers(p, stress_lengths);
   break;
  case V_STRESSAMP: // stressAmp
   stress_amps_set = Read8Numbers(p, stress_amps);
   break;
  case V_STRESSADD: // stressAdd
   stress_add_set = Read8Numbers(p, stress_add);
   break;
  case V_INTONATION: // intonation
   sscanf(p, "%d", &option_tone_flags);
   if ((option_tone_flags & 0xff) != 0) {
    if (langopts)
     langopts->intonation_group = option_tone_flags & 0xff;
    else
     fprintf(stderr, "Cannot set intonation: language not set, or
is invalid.\n");
   }
   break;
  case V_TUNES:
   n = sscanf(p, "%s %s %s %s %s %s", names[0], names[1],
names[2], names[3], names[4], names[5]);
   if (langopts) {
    langopts->intonation_group = 0;
    for (ix = 0; ix < n; ix++) {
```

```
      if (strcmp(names[ix], "NULL") == 0)
       continue;

      if ((value = LookupTune(names[ix])) < 0)
       fprintf(stderr, "Unknown tune '%s'\n", names[ix]);
      else
       langopts->tunes[ix] = value;
     }
    } else
     fprintf(stderr, "Cannot set tunes: language not set, or is
invalid.\n");
    break;
   case V_DICTRULES: // conditional dictionary rules and list
entries
   case V_NUMBERS:
   case V_STRESSOPT:
    if (langopts) {
     // expect a list of numbers
     while (*p != 0) {
      while (isspace(*p)) p++;
      if ((n = atoi(p)) > 0) {
       p++;
       if (n < 32) {
        if (key == V_DICTRULES)
         conditional_rules |= (1 << n);
        else if (key == V_NUMBERS)
         langopts->numbers |= (1 << n);
        else if (key == V_STRESSOPT)
         langopts->stress_flags |= (1 << n);
       } else {
        if ((key == V_NUMBERS) && (n < 64))
         langopts->numbers2 |= (1 << (n-32));
        else
         fprintf(stderr, "Bad option number %d\n", n);
       }
      }
      while (isalnum(*p)) p++;
```

```
    }
    ProcessLanguageOptions(langopts);
   } else
    fprintf(stderr, "Cannot set stressopt: language not set, or
is invalid.\n");
   break;
  case V_REPLACE:
   if (phonemes_set == false) {
    // must set up a phoneme table before we can lookup phoneme
mnemonics
    SelectPhonemeTableName(phonemes_name);
    phonemes_set = true;
   }
   PhonemeReplacement(p);
   break;
  case V_WORDGAP: // words
   if (langopts)
    sscanf(p, "%d %d", &langopts->word_gap,
&langopts->vowel_pause);
   else
    fprintf(stderr, "Cannot set wordgap: language not set, or is
invalid.\n");
   break;
  case V_STRESSRULE:
   if (langopts)
    sscanf(p, "%d %d %d %d", &langopts->stress_rule,
           &langopts->stress_flags,
           &langopts->unstressed_wd1,
           &langopts->unstressed_wd2);
   else
    fprintf(stderr, "Cannot set stressrule: language not set, or
is invalid.\n");
   break;
  case V_OPTION:
   if (langopts) {
    value2 = 0;
    if (((sscanf(p, "%s %d %d", option_name, &value, &value2) >=
```

```
2) && ((ix = LookupMnem(options_tab, option_name)) >= 0)) ||
        ((sscanf(p, "%d %d %d", &ix, &value, &value2) >= 2) &&
(ix < N_LOPTS))) {
     langopts->param[ix] = value;
     langopts->param2[ix] = value2;
    } else
     fprintf(stderr, "Bad voice option: %s %s\n", buf, p);
   } else
    fprintf(stderr, "Cannot set option: language not set, or is
invalid.\n");
   break;
  case V_ECHO:
   // echo.  suggest: 135mS  11%
   value = 0;
   voice->echo_amp = 0;
   sscanf(p, "%d %d", &voice->echo_delay, &voice->echo_amp);
   break;
  case V_FLUTTER: // flutter
   if (sscanf(p, "%d", &value) == 1)
    voice->flutter = value * 32;
   break;
  case V_ROUGHNESS: // roughness
   if (sscanf(p, "%d", &value) == 1)
    voice->roughness = value;
   break;
  case V_CLARITY: // formantshape
   if (sscanf(p, "%d", &value) == 1) {
    if (value > 4) {
     voice->peak_shape = 1; // squarer formant peaks
     value = 4;
    }
    voice->n_harmonic_peaks = 1+value;
   }
   break;
  case V_TONE:
  {
   int tone_data[12];
```

```
   ReadTonePoints(p, tone_data);
   SetToneAdjust(voice, tone_data);
  }
   break;
  case V_VOICING:
   if (sscanf(p, "%d", &value) == 1)
    voice->voicing = (value * 64)/100;
   break;
  case V_BREATH:
   voice->breath[0] = Read8Numbers(p, &voice->breath[1]);
   for (ix = 1; ix < 8; ix++) {
    if (ix % 2)
     voice->breath[ix] = -voice->breath[ix];
   }
   break;
  case V_BREATHW:
   voice->breathw[0] = Read8Numbers(p, &voice->breathw[1]);
   break;
  case V_CONSONANTS:
   value = sscanf(p, "%d %d", &voice->consonant_amp,
&voice->consonant_ampv);
   break;
  case V_SPEED:
   sscanf(p, "%d", &voice->speed_percent);
   break;
  case V_MBROLA:
  {
   int srate = 16000;

   name2[0] = 0;
   sscanf(p, "%s %s %d", name1, name2, &srate);
   espeak_ng_STATUS status = LoadMbrolaTable(name1, name2,
&srate);
   if (status != ENS_OK)
    espeak_ng_PrintStatusCodeMessage(status, stderr, NULL);
   else
    voice->samplerate = srate;
```

```
  }
   break;
  case V_KLATT:
   voice->klattv[0] = 1; // default source: IMPULSIVE
   Read8Numbers(p, voice->klattv);
   voice->klattv[KLATT_Kopen] -= 40;
   break;
  case V_FAST:
   Read8Numbers(p, speed.fast_settings);
   SetSpeed(3);
   break;
  case V_DICTMIN:
   sscanf(p, "%d", &dict_min);
   break;
  case V_MAINTAINER:
  case V_STATUS:
   break;
  default:
   if ((key & 0xff00) == 0x100) {
    if (langopts)
     sscanf(p, "%d", &langopts->param[key &0xff]);
    else
     fprintf(stderr, "Cannot set voice attribute: language not
set, or is invalid.\n");
   } else
    fprintf(stderr, "Bad voice attribute: %s\n", buf);
   break;
  }
 }
 if (f_voice != NULL)
  fclose(f_voice);

 if ((new_translator == NULL) && (!tone_only)) {
  // not set by language attribute
  new_translator = SelectTranslator(translator_name);
 }
```

```
 SetSpeed(3); // for speed_percent

 for (ix = 0; ix < N_PEAKS; ix++) {
  voice->freq2[ix] = voice->freq[ix];
  voice->height2[ix] = voice->height[ix];
  voice->width2[ix] = voice->width[ix];
 }

 if (tone_only)
  new_translator = translator;
 else {
  if ((ix = SelectPhonemeTableName(phonemes_name)) < 0) {
   fprintf(stderr, "Unknown phoneme table: '%s'\n",
phonemes_name);
   ix = 0;
  }
  voice->phoneme_tab_ix = ix;
  new_translator->phoneme_tab_ix = ix;
  new_translator->dict_min_size = dict_min;
  LoadDictionary(new_translator, new_dictionary, control & 4);
  if (dictionary_name[0] == 0) {
   DeleteTranslator(new_translator);
   return NULL; // no dictionary loaded
  }

  new_translator->dict_condition = conditional_rules;

  voice_languages[langix] = 0;
 }

 langopts = &new_translator->langopts;

 if ((value = langopts->param[LOPT_LENGTH_MODS]) != 0)
  SetLengthMods(new_translator, value);

 voice->width[0] = (voice->width[0] * 105)/100;
```

```
 if (!tone_only)
  translator = new_translator;

 // relative lengths of different stress syllables
 for (ix = 0; ix < stress_lengths_set; ix++)
  translator->stress_lengths[ix] = stress_lengths[ix];
 for (ix = 0; ix < stress_add_set; ix++)
  translator->stress_lengths[ix] += stress_add[ix];
 for (ix = 0; ix < stress_amps_set; ix++) {
  translator->stress_amps[ix] = stress_amps[ix];
  translator->stress_amps_r[ix] = stress_amps[ix] -1;
 }

 return voice;
}

static char *ExtractVoiceVariantName(char *vname, int
variant_num, int add_dir)
{
 // Remove any voice variant suffix (name or number) from a voice
name
 // Returns the voice variant name

 char *p;
 static char variant_name[40];
 char variant_prefix[5];

 variant_name[0] = 0;
 sprintf(variant_prefix, "!v%c", PATHSEP);
 if (add_dir == 0)
  variant_prefix[0] = 0;

 if (vname != NULL) {
  if ((p = strchr(vname, '+')) != NULL) {
   // The voice name has a +variant suffix
   variant_num = 0;
   *p++ = 0; // delete the suffix from the voice name
```

```
   if (IsDigit09(*p))
    variant_num = atoi(p); // variant number
   else {
    // voice variant name, not number
    sprintf(variant_name, "%s%s", variant_prefix, p);
   }
  }
 }

 if (variant_num > 0) {
  if (variant_num < 10)
   sprintf(variant_name, "%sm%d", variant_prefix, variant_num);
// male
  else
   sprintf(variant_name, "%sf%d", variant_prefix,
variant_num-10); // female
 }

 return variant_name;
}

voice_t *LoadVoiceVariant(const char *vname, int variant_num)
{
 // Load a voice file.
 // Also apply a voice variant if specified by "variant", or by
"+number" or "+name" in the "vname"

 voice_t *v;
 char *variant_name;
 char buf[60];

 strncpy0(buf, vname, sizeof(buf));
 variant_name = ExtractVoiceVariantName(buf, variant_num, 1);

 if ((v = LoadVoice(buf, 0)) == NULL)
  return NULL;
```

```
 if (variant_name[0] != 0)
  v = LoadVoice(variant_name, 2);
 return v;
}

static int __cdecl VoiceNameSorter(const void *p1, const void
*p2)
{
 int ix;
 espeak_VOICE *v1 = *(espeak_VOICE **)p1;
 espeak_VOICE *v2 = *(espeak_VOICE **)p2;

 if ((ix = strcmp(&v1->languages[1], &v2->languages[1])) != 0) //
primary language name
  return ix;
 if ((ix = v1->languages[0] - v2->languages[0]) != 0) // priority
number
  return ix;
 return strcmp(v1->name, v2->name);
}

static int __cdecl VoiceScoreSorter(const void *p1, const void
*p2)
{
 int ix;
 espeak_VOICE *v1 = *(espeak_VOICE **)p1;
 espeak_VOICE *v2 = *(espeak_VOICE **)p2;

 if ((ix = v2->score - v1->score) != 0)
  return ix;
 return strcmp(v1->name, v2->name);
}

static int ScoreVoice(espeak_VOICE *voice_spec, const char
*spec_language, int spec_n_parts, int spec_lang_len, espeak_VOICE
*voice)
{
```

```
 int ix;
 const char *p;
 int c1, c2;
 int language_priority;
 int n_parts;
 int matching;
 int matching_parts;
 int score = 0;
 int x;
 int ratio;
 int required_age;
 int diff;

 p = voice->languages; // list of languages+dialects for which
this voice is suitable

 if (spec_n_parts < 0) {
  // match on the subdirectory
  if (memcmp(voice->identifier, spec_language, spec_lang_len) ==
0)
   return 100;
  return 0;
 }

 if (spec_n_parts == 0)
  score = 100;
 else {
  if ((*p == 0) && (strcmp(spec_language, "variants") == 0)) {
   // match on a voice with no languages if the required language
is "variants"
   score = 100;
  }

  // compare the required language with each of the languages of
this voice
  while (*p != 0) {
   language_priority = *p++;
```

```
   matching = 1;
   matching_parts = 0;
   n_parts = 1;

   for (ix = 0;; ix++) {
    if ((ix >= spec_lang_len) || ((c1 = spec_language[ix]) ==
'-'))
     c1 = 0;
    if ((c2 = p[ix]) == '-')
     c2 = 0;

    if (c1 != c2)
     matching = 0;

    if (p[ix] == '-') {
     n_parts++;
     if (matching)
      matching_parts++;
    }
    if (p[ix] == 0)
     break;
   }
   p += (ix+1);
   matching_parts += matching; // number of parts which match

   if (matching_parts == 0)
    continue; // no matching parts for this language

   x = 5;
   // reduce the score if not all parts of the required language
match
   if ((diff = (spec_n_parts - matching_parts)) > 0)
    x -= diff;

   // reduce score if the language is more specific than required
   if ((diff = (n_parts - matching_parts)) > 0)
```

```
    x -= diff;

   x = x*100 - (language_priority * 2);

   if (x > score)
    score = x;
  }
 }
 if (score == 0)
  return 0;

 if (voice_spec->name != NULL) {
  if (strcmp(voice_spec->name, voice->name) == 0) {
   // match on voice name
   score += 500;
  } else if (strcmp(voice_spec->name, voice->identifier) == 0)
   score += 400;
 }

 if (((voice_spec->gender == ENGENDER_MALE) ||
(voice_spec->gender == ENGENDER_FEMALE)) &&
     ((voice->gender == ENGENDER_MALE) || (voice->gender ==
ENGENDER_FEMALE))) {
  if (voice_spec->gender == voice->gender)
   score += 50;
  else
   score -= 50;
 }

 if ((voice_spec->age <= 12) && (voice->gender ==
ENGENDER_FEMALE) && (voice->age > 12))
  score += 5; // give some preference for non-child female voice
if a child is requested

 if (voice->age != 0) {
  if (voice_spec->age == 0)
   required_age = 30;
```

```
  else
   required_age = voice_spec->age;

  ratio = (required_age*100)/voice->age;
  if (ratio < 100)
   ratio = 10000/ratio;
  ratio = (ratio - 100)/10; // 0=exact match, 10=out by factor of
2
  x = 5 - ratio;
  if (x > 0) x = 0;

  score = score + x;

  if (voice_spec->age > 0)
   score += 10; // required age specified, favour voices with a
specified age (near it)
 }
 if (score < 1)
  score = 1;
 return score;
}

static int SetVoiceScores(espeak_VOICE *voice_select,
espeak_VOICE **voices, int control)
{
 // control: bit0=1  include mbrola voices
 int ix;
 int score;
 int nv; // number of candidates
 int n_parts = 0;
 int lang_len = 0;
 espeak_VOICE *vp;
 char language[80];
 char buf[sizeof(path_home)+80];

 // count number of parts in the specified language
 if ((voice_select->languages != NULL) &&
```

```
(voice_select->languages[0] != 0)) {
  n_parts = 1;
  lang_len = strlen(voice_select->languages);
  for (ix = 0; (ix <= lang_len) && ((unsigned)ix <
sizeof(language)); ix++) {
   if ((language[ix] = tolower(voice_select->languages[ix])) ==
'-')
    n_parts++;
  }
 }

 if ((n_parts == 1) && (control & 1)) {
  if (strcmp(language, "mbrola") == 0) {
   language[2] = 0; // truncate to "mb"
   lang_len = 2;
  }

  sprintf(buf, "%s/voices/%s", path_home, language);
  if (GetFileLength(buf) == -EISDIR) {
   // A subdirectory name has been specified.  List all the
voices in that subdirectory
   language[lang_len++] = PATHSEP;
   language[lang_len] = 0;
   n_parts = -1;
  }
 }

 // select those voices which match the specified language
 nv = 0;
 for (ix = 0; ix < n_voices_list; ix++) {
  vp = voices_list[ix];

  if (((control & 1) == 0) && (memcmp(vp->identifier, "mb/", 3)
== 0))
   continue;

  if ((score = ScoreVoice(voice_select, language, n_parts,
```

```
lang_len, voices_list[ix])) > 0) {
   voices[nv++] = vp;
   vp->score = score;
  }
 }
 voices[nv] = NULL; // list terminator

 if (nv == 0)
  return 0;

 // sort the selected voices by their score
 qsort(voices, nv, sizeof(espeak_VOICE *), (int(__cdecl *)(const
void *, const void *))VoiceScoreSorter);

 return nv;
}

espeak_VOICE *SelectVoiceByName(espeak_VOICE **voices, const char
*name2)
{
 int ix;
 int match_fname = -1;
 int match_fname2 = -1;
 int match_name = -1;
 const char *id; // this is the filename within espeak-ng-
data/voices
 char *variant_name;
 int last_part_len;
 char last_part[41];
 char name[40];

 if (voices == NULL) {
  if (n_voices_list == 0)
   espeak_ListVoices(NULL); // create the voices list
  voices = voices_list;
 }
```

```
 strncpy0(name, name2, sizeof(name));
 if ((variant_name = strchr(name, '+')) != NULL) {
  *variant_name = 0;
  variant_name++;
 }

 sprintf(last_part, "%c%s", PATHSEP, name);
 last_part_len = strlen(last_part);

 for (ix = 0; voices[ix] != NULL; ix++) {
  if (strcasecmp(name, voices[ix]->name) == 0) {
   match_name = ix; // found matching voice name
   break;
  } else {
   id = voices[ix]->identifier;
   if (strcasecmp(name, id) == 0)
    match_fname = ix; // matching identifier, use this if no
matching name
   else if (strcasecmp(last_part, &id[strlen(id)-last_part_len])
== 0)
    match_fname2 = ix;
  }
 }

 if (match_name < 0) {
  match_name = match_fname; // no matching name, try matching
filename
  if (match_name < 0)
   match_name = match_fname2; // try matching just the last part
of the filename
 }

 if (match_name < 0)
  return NULL;

 return voices[match_name];
}
```

```
char const *SelectVoice(espeak_VOICE *voice_select, int *found)
{
 // Returns a path within espeak-voices, with a possible +variant
suffix
 // variant is an output-only parameter

 int nv; // number of candidates
 int ix, ix2;
 int j;
 int n_variants;
 int variant_number;
 int gender;
 int skip;
 int aged = 1;
 char *variant_name;
 const char *p, *p_start;
 espeak_VOICE *vp = NULL;
 espeak_VOICE *vp2;
 espeak_VOICE voice_select2;
 espeak_VOICE *voices[N_VOICES_LIST]; // list of candidates
 espeak_VOICE *voices2[N_VOICES_LIST+N_VOICE_VARIANTS];
 static espeak_VOICE voice_variants[N_VOICE_VARIANTS];
 static char voice_id[50];

 memcpy(&voice_select2, voice_select, sizeof(voice_select2));

 if (n_voices_list == 0)
  espeak_ListVoices(NULL); // create the voices list

 if ((voice_select2.languages == NULL) ||
(voice_select2.languages[0] == 0)) {
  // no language is specified. Get language from the named voice
  static char buf[60];

  if (voice_select2.name == NULL) {
   if ((voice_select2.name = voice_select2.identifier) == NULL)
```

```
    voice_select2.name = ESPEAKNG_DEFAULT_VOICE;
  }

  strncpy0(buf, voice_select2.name, sizeof(buf));
  variant_name = ExtractVoiceVariantName(buf, 0, 0);

  vp = SelectVoiceByName(voices_list, buf);
  if (vp != NULL) {
   voice_select2.languages = &(vp->languages[1]);

   if ((voice_select2.gender == ENGENDER_UNKNOWN) &&
(voice_select2.age == 0) && (voice_select2.variant == 0)) {
    if (variant_name[0] != 0) {
     sprintf(voice_id, "%s+%s", vp->identifier, variant_name);
     return voice_id;
    }

    return vp->identifier;
   }
  }
 }

 // select and sort voices for the required language
 nv = SetVoiceScores(&voice_select2, voices, 0);

 if (nv == 0) {
  // no matching voice, choose the default
  *found = 0;
  if ((voices[0] = SelectVoiceByName(voices_list,
ESPEAKNG_DEFAULT_VOICE)) != NULL)
   nv = 1;
 }

 gender = 0;
 if ((voice_select2.gender == ENGENDER_FEMALE) ||
((voice_select2.age > 0) && (voice_select2.age < 13)))
  gender = ENGENDER_FEMALE;
```

```
 else if (voice_select2.gender == ENGENDER_MALE)
  gender = ENGENDER_MALE;

 #define AGE_OLD 60
 if (voice_select2.age < AGE_OLD)
  aged = 0;

 p = p_start = variant_lists[gender];
 if (aged == 0)
  p++; // the first voice in the variants list is older

 // add variants for the top voices
 n_variants = 0;
 for (ix = 0, ix2 = 0; ix < nv; ix++) {
  vp = voices[ix];
  // is the main voice the required gender?
  skip = 0;

  if ((gender != ENGENDER_UNKNOWN) && (vp->gender != gender))
   skip = 1;
  if ((ix2 == 0) && aged && (vp->age < AGE_OLD))
   skip = 1;

  if (skip == 0)
   voices2[ix2++] = vp;

  for (j = 0; (j < vp->xx1) && (n_variants < N_VOICE_VARIANTS);)
{
   if ((variant_number = *p) == 0) {
    p = p_start;
    continue;
   }

   vp2 = &voice_variants[n_variants++]; // allocate space for
voice variant
   memcpy(vp2, vp, sizeof(espeak_VOICE)); // copy from the
original voice
```

```
   vp2->variant = variant_number;
   voices2[ix2++] = vp2;
   p++;
   j++;
  }
 }
 // add any more variants to the end of the list
 while ((vp != NULL) && ((variant_number = *p++) != 0) &&
(n_variants < N_VOICE_VARIANTS)) {
  vp2 = &voice_variants[n_variants++]; // allocate space for
voice variant
  memcpy(vp2, vp, sizeof(espeak_VOICE)); // copy from the
original voice
  vp2->variant = variant_number;
  voices2[ix2++] = vp2;
 }

 // index the sorted list by the required variant number
 if (ix2 == 0)
  return NULL;
 vp = voices2[voice_select2.variant % ix2];

 if (vp->variant != 0) {
  variant_name = ExtractVoiceVariantName(NULL, vp->variant, 0);
  sprintf(voice_id, "%s+%s", vp->identifier, variant_name);
  return voice_id;
 }

 return vp->identifier;
}

static void GetVoices(const char *path, int len_path_voices, int
is_language_file)
{
 FILE *f_voice;
 espeak_VOICE *voice_data;
 int ftype;
```

```
 char fname[sizeof(path_home)+100];

#ifdef PLATFORM_WINDOWS
 WIN32_FIND_DATAA FindFileData;
 HANDLE hFind = INVALID_HANDLE_VALUE;

 #undef UNICODE // we need FindFirstFileA() which takes an 8-bit
c-string
 sprintf(fname, "%s\\*", path);
 hFind = FindFirstFileA(fname, &FindFileData);
 if (hFind == INVALID_HANDLE_VALUE)
  return;

 do {
  if (n_voices_list >= (N_VOICES_LIST-2)) {
   fprintf(stderr, "Warning: maximum number %d of (N_VOICES_LIST
= %d - 1) reached\n", n_voices_list + 1, N_VOICES_LIST);
   break; // voices list is full
  }

  if (FindFileData.cFileName[0] != '.') {
   sprintf(fname, "%s%c%s", path, PATHSEP,
FindFileData.cFileName);
   ftype = GetFileLength(fname);

   if (ftype == -EISDIR) {
    // a sub-directory
    GetVoices(fname, len_path_voices, is_language_file);
   } else if (ftype > 0) {
    // a regular file, add it to the voices list
    if ((f_voice = fopen(fname, "r")) == NULL)
     continue;

    // pass voice file name within the voices directory
    voice_data = ReadVoiceFile(f_voice, fname+len_path_voices,
is_language_file);
    fclose(f_voice);
```

```
    if (voice_data != NULL)
     voices_list[n_voices_list++] = voice_data;
   }
  }
 } while (FindNextFileA(hFind, &FindFileData) != 0);
 FindClose(hFind);
#else
 DIR *dir;
 struct dirent *ent;

 if ((dir = opendir((char *)path)) == NULL) // note: (char *) is
needed for WINCE
  return;

 while ((ent = readdir(dir)) != NULL) {
  if (n_voices_list >= (N_VOICES_LIST-2)) {
   fprintf(stderr, "Warning: maximum number %d of (N_VOICES_LIST
= %d - 1) reached\n", n_voices_list + 1, N_VOICES_LIST);
   break; // voices list is full
  }

  if (ent->d_name[0] == '.')
   continue;

  sprintf(fname, "%s%c%s", path, PATHSEP, ent->d_name);

  ftype = GetFileLength(fname);

  if (ftype == -EISDIR) {
   // a sub-directory
   GetVoices(fname, len_path_voices, is_language_file);
  } else if (ftype > 0) {
   // a regular file, add it to the voices list
   if ((f_voice = fopen(fname, "r")) == NULL)
    continue;
```

```
   // pass voice file name within the voices directory
   voice_data = ReadVoiceFile(f_voice, fname+len_path_voices,
is_language_file);
   fclose(f_voice);

   if (voice_data != NULL)
    voices_list[n_voices_list++] = voice_data;
  }
 }
 closedir(dir);
#endif
}

#pragma GCC visibility push(default)

ESPEAK_NG_API espeak_ng_STATUS espeak_ng_SetVoiceByFile(const
char *filename)
{
 int ix;
 espeak_VOICE voice_selector;
 char *variant_name;
 static char buf[60];

 strncpy0(buf, filename, sizeof(buf));

 variant_name = ExtractVoiceVariantName(buf, 0, 1);

 for (ix = 0;; ix++) {
  // convert voice name to lower case  (ascii)
  if ((buf[ix] = tolower(buf[ix])) == 0)
   break;
 }

 memset(&voice_selector, 0, sizeof(voice_selector));
 voice_selector.name = (char *)filename; // include variant name
in voice stack ??
```

```
 // first check for a voice with this filename
 // This may avoid the need to call espeak_ListVoices().

 if (LoadVoice(buf, 0x10) != NULL) {
  if (variant_name[0] != 0)
   LoadVoice(variant_name, 2);

  DoVoiceChange(voice);
  voice_selector.languages = voice->language_name;
  SetVoiceStack(&voice_selector, variant_name);
  return ENS_OK;
 }

 return ENS_VOICE_NOT_FOUND;
}

ESPEAK_NG_API espeak_ng_STATUS espeak_ng_SetVoiceByName(const
char *name)
{
 espeak_VOICE *v;
 int ix;
 espeak_VOICE voice_selector;
 char *variant_name;
 static char buf[60];

 strncpy0(buf, name, sizeof(buf));

 variant_name = ExtractVoiceVariantName(buf, 0, 1);

 for (ix = 0;; ix++) {
  // convert voice name to lower case  (ascii)
  if ((buf[ix] = tolower(buf[ix])) == 0)
   break;
 }

 memset(&voice_selector, 0, sizeof(voice_selector));
 voice_selector.name = (char *)name; // include variant name in
```

```
voice stack ??

 // first check for a voice with this filename
 // This may avoid the need to call espeak_ListVoices().

 if (LoadVoice(buf, 1) != NULL) {
  if (variant_name[0] != 0)
   LoadVoice(variant_name, 2);

  DoVoiceChange(voice);
  voice_selector.languages = voice->language_name;
  SetVoiceStack(&voice_selector, variant_name);
  return ENS_OK;
 }

 if (n_voices_list == 0)
  espeak_ListVoices(NULL); // create the voices list

 if ((v = SelectVoiceByName(voices_list, buf)) != NULL) {
  if (LoadVoice(v->identifier, 0) != NULL) {
   if (variant_name[0] != 0)
    LoadVoice(variant_name, 2);
   DoVoiceChange(voice);
   voice_selector.languages = voice->language_name;
   SetVoiceStack(&voice_selector, variant_name);
   return ENS_OK;
  }
 }
 return ENS_VOICE_NOT_FOUND;
}

ESPEAK_NG_API espeak_ng_STATUS
espeak_ng_SetVoiceByProperties(espeak_VOICE *voice_selector)
{
 const char *voice_id;
 int voice_found;
```

```
 voice_id = SelectVoice(voice_selector, &voice_found);
 if (voice_found == 0)
  return ENS_VOICE_NOT_FOUND;

 LoadVoiceVariant(voice_id, 0);
 DoVoiceChange(voice);
 SetVoiceStack(voice_selector, "");

 return ENS_OK;
}

#pragma GCC visibility pop

void FreeVoiceList()
{
 int ix;
 for (ix = 0; ix < n_voices_list; ix++) {
  if (voices_list[ix] != NULL) {
   free(voices_list[ix]);
   voices_list[ix] = NULL;
  }
 }
 n_voices_list = 0;
}

#pragma GCC visibility push(default)

ESPEAK_API const espeak_VOICE **espeak_ListVoices(espeak_VOICE
*voice_spec)
{
 char path_voices[sizeof(path_home)+12];

 int ix;
 int j;
 espeak_VOICE *v;
 static espeak_VOICE **voices = NULL;
```

```
 // free previous voice list data
 FreeVoiceList();

 sprintf(path_voices, "%s%cvoices", path_home, PATHSEP);
 GetVoices(path_voices, strlen(path_voices)+1, 0);

 sprintf(path_voices, "%s%clang", path_home, PATHSEP);
 GetVoices(path_voices, strlen(path_voices)+1, 1);

 voices_list[n_voices_list] = NULL; // voices list terminator
 espeak_VOICE **new_voices = (espeak_VOICE **)realloc(voices,
sizeof(espeak_VOICE *)*(n_voices_list+1));
 if (new_voices == NULL)
  return (const espeak_VOICE **)voices;
 voices = new_voices;

 // sort the voices list
 qsort(voices_list, n_voices_list, sizeof(espeak_VOICE *),
       (int(__cdecl *)(const void *, const void
*))VoiceNameSorter);

 if (voice_spec) {
  // select the voices which match the voice_spec, and sort them
by preference
  SetVoiceScores(voice_spec, voices, 1);
 } else {
  // list all: omit variant and mbrola voices
  j = 0;
  for (ix = 0; (v = voices_list[ix]) != NULL; ix++) {
   if ((v->languages[0] != 0) && (strcmp(&v->languages[1],
"variant") != 0)
       && (memcmp(v->identifier, "mb/", 3) != 0))
    voices[j++] = v;
  }
  voices[j] = NULL;
 }
 return (const espeak_VOICE **)voices;
```

```
}

ESPEAK_API espeak_VOICE *espeak_GetCurrentVoice(void)
{
 return &current_voice_selected;
}

#pragma GCC visibility pop
```

Chapter 52

./src/libespeak-ng/event.c

```
// This source file is only used for asynchronious modes

#include "config.h"

#include <assert.h>
#include <errno.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <unistd.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>

#include "event.h"

// my_mutex: protects my_thread_is_talking,
static pthread_mutex_t my_mutex;
static pthread_cond_t my_cond_start_is_required;
```

```
static bool my_start_is_required = false;
static pthread_cond_t my_cond_stop_is_required;
static bool my_stop_is_required = false;
static pthread_cond_t my_cond_stop_is_acknowledged;
static bool my_stop_is_acknowledged = false;
static bool my_terminate_is_required = 0;
// my_thread: polls the audio duration and compares it to the
duration of the first event.
static pthread_t my_thread;
static bool thread_inited;

static t_espeak_callback *my_callback = NULL;
static bool my_event_is_running = false;

enum {
 MIN_TIMEOUT_IN_MS = 10,
 ACTIVITY_TIMEOUT = 50, // in ms, check that the stream is active
 MAX_ACTIVITY_CHECK = 6
};

typedef struct t_node {
 void *data;
 struct t_node *next;
} node;

static node *head = NULL;
static node *tail = NULL;
static int node_counter = 0;
static espeak_ng_STATUS push(void *data);
static void *pop(void);
static void init(void);
static void *polling_thread(void *);

void event_set_callback(t_espeak_callback *SynthCallback)
{
 my_callback = SynthCallback;
}
```

```
void event_init(void)
{
 my_event_is_running = false;

 // security
 pthread_mutex_init(&my_mutex, (const pthread_mutexattr_t
*)NULL);
 init();

 assert(-1 != pthread_cond_init(&my_cond_start_is_required,
NULL));
 assert(-1 != pthread_cond_init(&my_cond_stop_is_required,
NULL));
 assert(-1 != pthread_cond_init(&my_cond_stop_is_acknowledged,
NULL));

 pthread_attr_t a_attrib;

 if (pthread_attr_init(&a_attrib) == 0
     && pthread_attr_setdetachstate(&a_attrib,
PTHREAD_CREATE_JOINABLE) == 0) {
  thread_inited = (0 == pthread_create(&my_thread,
                                       &a_attrib,
                                       polling_thread,
                                       (void *)NULL));
 }
 assert(thread_inited);
 pthread_attr_destroy(&a_attrib);
}

static espeak_EVENT *event_copy(espeak_EVENT *event)
{
 if (event == NULL)
  return NULL;

 espeak_EVENT *a_event = (espeak_EVENT
```

```
*)malloc(sizeof(espeak_EVENT));
 if (a_event) {
  memcpy(a_event, event, sizeof(espeak_EVENT));

  switch (event->type)
  {
  case espeakEVENT_MARK:
  case espeakEVENT_PLAY:
   if (event->id.name)
    a_event->id.name = strdup(event->id.name);
   break;

  default:
   break;
  }
 }

 return a_event;
}

// Call the user supplied callback
//
// Note: the current sequence is:
//
// * First call with: event->type = espeakEVENT_SENTENCE
// * 0, 1 or several calls: event->type = espeakEVENT_WORD
// * Last call: event->type = espeakEVENT_MSG_TERMINATED
//

static void event_notify(espeak_EVENT *event)
{
 static unsigned int a_old_uid = 0;

 espeak_EVENT events[2];
 memcpy(&events[0], event, sizeof(espeak_EVENT));     // the
event parameter in the callback function should be an array of
eventd
```

```
 memcpy(&events[1], event, sizeof(espeak_EVENT));
 events[1].type = espeakEVENT_LIST_TERMINATED;           // ...
terminated by an event type=0

 if (event && my_callback) {
  switch (event->type)
  {
  case espeakEVENT_SENTENCE:
   my_callback(NULL, 0, events);
   a_old_uid = event->unique_identifier;
   break;
  case espeakEVENT_MSG_TERMINATED:
  case espeakEVENT_MARK:
  case espeakEVENT_WORD:
  case espeakEVENT_END:
  case espeakEVENT_PHONEME:
  {
   if (a_old_uid != event->unique_identifier) {
    espeak_EVENT_TYPE a_new_type = events[0].type;
    events[0].type = espeakEVENT_SENTENCE;
    my_callback(NULL, 0, events);
    events[0].type = a_new_type;
    usleep(50000);
   }
   my_callback(NULL, 0, events);
   a_old_uid = event->unique_identifier;
  }
   break;
  case espeakEVENT_LIST_TERMINATED:
  case espeakEVENT_PLAY:
  default:
   break;
  }
 }
}

static int event_delete(espeak_EVENT *event)
```

```
{
 if (event == NULL)
  return 0;

 switch (event->type)
 {
 case espeakEVENT_MSG_TERMINATED:
  event_notify(event);
  break;
 case espeakEVENT_MARK:
 case espeakEVENT_PLAY:
  if (event->id.name)
   free((void *)(event->id.name));
  break;
 default:
  break;
 }

 free(event);
 return 1;
}

espeak_ng_STATUS event_declare(espeak_EVENT *event)
{
 if (!event)
  return EINVAL;

 espeak_ng_STATUS status;
 if ((status = pthread_mutex_lock(&my_mutex)) != ENS_OK) {
  my_start_is_required = true;
  return status;
 }

 espeak_EVENT *a_event = event_copy(event);
 if ((status = push(a_event)) != ENS_OK) {
  event_delete(a_event);
  pthread_mutex_unlock(&my_mutex);
```

```
 } else {
  my_start_is_required = true;
  pthread_cond_signal(&my_cond_start_is_required);
  status = pthread_mutex_unlock(&my_mutex);
 }

 return status;
}

espeak_ng_STATUS event_clear_all()
{
 espeak_ng_STATUS status;
 if ((status = pthread_mutex_lock(&my_mutex)) != ENS_OK)
  return status;

 int a_event_is_running = 0;
 if (my_event_is_running) {
  my_stop_is_required = true;
  pthread_cond_signal(&my_cond_stop_is_required);
  a_event_is_running = 1;
 } else
  init(); // clear pending events

 if (a_event_is_running) {
  while (my_stop_is_acknowledged == false) {
   while ((pthread_cond_wait(&my_cond_stop_is_acknowledged,
&my_mutex) == -1) && errno == EINTR)
    continue; // Restart when interrupted by handler
  }
 }

 if ((status = pthread_mutex_unlock(&my_mutex)) != ENS_OK)
  return status;

 return ENS_OK;
}
```

```
static void *polling_thread(void *p)
{
 (void)p; // unused

 while (!my_terminate_is_required) {
  bool a_stop_is_required = false;

  (void)pthread_mutex_lock(&my_mutex);
  my_event_is_running = false;

  while (my_start_is_required == false &&
my_terminate_is_required == false) {
   while ((pthread_cond_wait(&my_cond_start_is_required,
&my_mutex) == -1) && errno == EINTR)
    continue; // Restart when interrupted by handler
  }

  my_event_is_running = true;
  a_stop_is_required = false;
  my_start_is_required = false;

  pthread_mutex_unlock(&my_mutex);

  // In this loop, my_event_is_running = true
  while (head && (a_stop_is_required == false) &&
(my_terminate_is_required == false)) {
   espeak_EVENT *event = (espeak_EVENT *)(head->data);
   assert(event);

   if (my_callback) {
    event_notify(event);
    // the user_data (and the type) are cleaned to be sure
    // that MSG_TERMINATED is called twice (at delete time too).
    event->type = espeakEVENT_LIST_TERMINATED;
    event->user_data = NULL;
   }
```

```
    (void)pthread_mutex_lock(&my_mutex);
    event_delete((espeak_EVENT *)pop());
    a_stop_is_required = my_stop_is_required;
    if (a_stop_is_required == true)
     my_stop_is_required = false;

    (void)pthread_mutex_unlock(&my_mutex);
   }

   (void)pthread_mutex_lock(&my_mutex);

   my_event_is_running = false;

   if (a_stop_is_required == false) {
    a_stop_is_required = my_stop_is_required;
    if (a_stop_is_required == true)
     my_stop_is_required = false;
   }

   (void)pthread_mutex_unlock(&my_mutex);

   if (a_stop_is_required == true || my_terminate_is_required ==
true) {
    // no mutex required since the stop command is synchronous
    // and waiting for my_cond_stop_is_acknowledged
    init();

    // acknowledge the stop request
    (void)pthread_mutex_lock(&my_mutex);
    my_stop_is_acknowledged = true;
    (void)pthread_cond_signal(&my_cond_stop_is_acknowledged);
    (void)pthread_mutex_unlock(&my_mutex);
   }
  }

 return NULL;
}
```

```
enum { MAX_NODE_COUNTER = 1000 };

static espeak_ng_STATUS push(void *the_data)
{
 assert((!head && !tail) || (head && tail));

 if (the_data == NULL)
  return EINVAL;

 if (node_counter >= MAX_NODE_COUNTER)
  return ENS_EVENT_BUFFER_FULL;

 node *n = (node *)malloc(sizeof(node));
 if (n == NULL)
  return ENOMEM;

 if (head == NULL) {
  head = n;
  tail = n;
 } else {
  tail->next = n;
  tail = n;
 }

 tail->next = NULL;
 tail->data = the_data;

 node_counter++;

 return ENS_OK;
}

static void *pop()
{
 void *the_data = NULL;
```

```
 assert((!head && !tail) || (head && tail));

 if (head != NULL) {
  node *n = head;
  the_data = n->data;
  head = n->next;
  free(n);
  node_counter--;
 }

 if (head == NULL)
  tail = NULL;

 return the_data;
}

static void init()
{
 while (event_delete((espeak_EVENT *)pop()))
  ;

 node_counter = 0;
}

void event_terminate()
{
 if (thread_inited) {
  my_terminate_is_required = true;
  pthread_cond_signal(&my_cond_start_is_required);
  pthread_cond_signal(&my_cond_stop_is_required);
  pthread_join(my_thread, NULL);
  my_terminate_is_required = false;

  pthread_mutex_destroy(&my_mutex);
  pthread_cond_destroy(&my_cond_start_is_required);
  pthread_cond_destroy(&my_cond_stop_is_required);
  pthread_cond_destroy(&my_cond_stop_is_acknowledged);
```

```
  init(); // purge event
  thread_inited = 0;
 }
}

enum { ONE_BILLION = 1000000000 };

void clock_gettime2(struct timespec *ts)
{
 struct timeval tv;

 if (!ts)
  return;

 assert(gettimeofday(&tv, NULL) != -1);
 ts->tv_sec = tv.tv_sec;
 ts->tv_nsec = tv.tv_usec*1000;
}

void add_time_in_ms(struct timespec *ts, int time_in_ms)
{
 if (!ts)
  return;

 uint64_t t_ns = (uint64_t)ts->tv_nsec + 1000000 *
(uint64_t)time_in_ms;
 while (t_ns >= ONE_BILLION) {
  ts->tv_sec += 1;
  t_ns -= ONE_BILLION;
 }
 ts->tv_nsec = (long int)t_ns;
}
```

Chapter 53

./src/libespeak-ng/ssml.c

```
#include "config.h"

#include <ctype.h>
#include <errno.h>
#include <locale.h>
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <wchar.h>
#include <wctype.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>
#include <ucd/ucd.h>

#include "readclause.h"

#include "error.h"
```

```
#include "speech.h"
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"
#include "dictionary.h"
#include "ssml.h"

static MNEM_TAB ssmltags[] = {
 { "speak",     SSML_SPEAK },
 { "voice",     SSML_VOICE },
 { "prosody",   SSML_PROSODY },
 { "say-as",    SSML_SAYAS },
 { "mark",      SSML_MARK },
 { "s",         SSML_SENTENCE },
 { "p",         SSML_PARAGRAPH },
 { "phoneme",   SSML_PHONEME },
 { "sub",       SSML_SUB },
 { "tts:style", SSML_STYLE },
 { "audio",     SSML_AUDIO },
 { "emphasis",  SSML_EMPHASIS },
 { "break",     SSML_BREAK },
 { "metadata",  SSML_IGNORE_TEXT },

 { "br",     HTML_BREAK },
 { "li",     HTML_BREAK },
 { "dd",     HTML_BREAK },
 { "img",    HTML_BREAK },
 { "td",     HTML_BREAK },
 { "h1",     SSML_PARAGRAPH },
 { "h2",     SSML_PARAGRAPH },
 { "h3",     SSML_PARAGRAPH },
 { "h4",     SSML_PARAGRAPH },
 { "hr",     SSML_PARAGRAPH },
 { "script", SSML_IGNORE_TEXT },
 { "style",  SSML_IGNORE_TEXT },
 { "font",   HTML_NOSPACE },
```

```
 { "b",      HTML_NOSPACE },
 { "i",      HTML_NOSPACE },
 { "strong", HTML_NOSPACE },
 { "em",     HTML_NOSPACE },
 { "code",   HTML_NOSPACE },

 { NULL, 0 }
};

static int attrcmp(const wchar_t *string1, const char *string2)
{
 int ix;

 if (string1 == NULL)
  return 1;

 for (ix = 0; (string1[ix] == string2[ix]) && (string1[ix] != 0);
ix++)
  ;
 if (((string1[ix] == '"') || (string1[ix] == '\'')) &&
(string2[ix] == 0))
  return 0;
 return 1;
}

static int attrlookup(const wchar_t *string1, const MNEM_TAB
*mtab)
{
 int ix;

 for (ix = 0; mtab[ix].mnem != NULL; ix++) {
  if (attrcmp(string1, mtab[ix].mnem) == 0)
   return mtab[ix].value;
 }
 return mtab[ix].value;
}
```

```
static int attrnumber(const wchar_t *pw, int default_value, int
type)
{
 int value = 0;

 if ((pw == NULL) || !IsDigit09(*pw))
  return default_value;

 while (IsDigit09(*pw))
  value = value*10 + *pw++ - '0';
 if ((type == 1) && (ucd_tolower(*pw) == 's')) {
  // time: seconds rather than ms
  value *= 1000;
 }
 return value;
}

static int attrcopy_utf8(char *buf, const wchar_t *pw, int len)
{
 // Convert attribute string into utf8, write to buf, and return
its utf8 length
 unsigned int c;
 int ix = 0;
 int n;
 int prev_c = 0;

 if (pw != NULL) {
  while ((ix < (len-4)) && ((c = *pw++) != 0)) {
   if ((c == '"') && (prev_c != '\\'))
    break; // " indicates end of attribute, unless preceded by
backstroke
   n = utf8_out(c, &buf[ix]);
   ix += n;
   prev_c = c;
  }
 }
 buf[ix] = 0;
```

```
 return ix;
}

static int attr_prosody_value(int param_type, const wchar_t *pw,
int *value_out)
{
 int sign = 0;
 wchar_t *tail;
 double value;

 while (iswspace(*pw)) pw++;
 if (*pw == '+') {
  pw++;
  sign = 1;
 }
 if (*pw == '-') {
  pw++;
  sign = -1;
 }
 value = (double)wcstod(pw, &tail);
 if (tail == pw) {
  // failed to find a number, return 100%
  *value_out = 100;
  return 2;
 }

 if (*tail == '%') {
  if (sign != 0)
   value = 100 + (sign * value);
  *value_out = (int)value;
  return 2; // percentage
 }

 if ((tail[0] == 's') && (tail[1] == 't')) {
  double x;
  // convert from semitones to a  frequency percentage
  x = pow((double)2.0, (double)((value*sign)/12)) * 100;
```

```
  *value_out = (int)x;
  return 2; // percentage
 }

 if (param_type == espeakRATE) {
  if (sign == 0)
   *value_out = (int)(value * 100);
  else
   *value_out = 100 + (int)(sign * value * 100);
  return 2; // percentage
 }

 return sign;   // -1, 0, or 1
}

static const char *VoiceFromStack(SSML_STACK *ssml_stack, int
n_ssml_stack, espeak_VOICE *base_voice, char
base_voice_variant_name[40])
{
 // Use the voice properties from the SSML stack to choose a
voice, and switch
 // to that voice if it's not the current voice

 int ix;
 const char *p;
 SSML_STACK *sp;
 const char *v_id;
 int voice_name_specified;
 int voice_found;
 espeak_VOICE voice_select;
 static char voice_name[40];
 char language[40];
 char buf[80];

 strcpy(voice_name, ssml_stack[0].voice_name);
 strcpy(language, ssml_stack[0].language);
 voice_select.age = ssml_stack[0].voice_age;
```

```
 voice_select.gender = ssml_stack[0].voice_gender;
 voice_select.variant = ssml_stack[0].voice_variant_number;
 voice_select.identifier = NULL;

 for (ix = 0; ix < n_ssml_stack; ix++) {
  sp = &ssml_stack[ix];
  voice_name_specified = 0;

  if ((sp->voice_name[0] != 0) && (SelectVoiceByName(NULL,
sp->voice_name) != NULL)) {
   voice_name_specified = 1;
   strcpy(voice_name, sp->voice_name);
   language[0] = 0;
   voice_select.gender = ENGENDER_UNKNOWN;
   voice_select.age = 0;
   voice_select.variant = 0;
  }
  if (sp->language[0] != 0) {
   strcpy(language, sp->language);

   // is this language provided by the base voice?
   p = base_voice->languages;
   while (*p++ != 0) {
    if (strcmp(p, language) == 0) {
     // yes, change the language to the main language of the base
voice
     strcpy(language, &base_voice->languages[1]);
     break;
    }
    p += (strlen(p) + 1);
   }

   if (voice_name_specified == 0)
    voice_name[0] = 0; // forget a previous voice name if a
language is specified
  }
  if (sp->voice_gender != ENGENDER_UNKNOWN)
```

```
   voice_select.gender = sp->voice_gender;

  if (sp->voice_age != 0)
   voice_select.age = sp->voice_age;
  if (sp->voice_variant_number != 0)
   voice_select.variant = sp->voice_variant_number;
 }

 voice_select.name = voice_name;
 voice_select.languages = language;
 v_id = SelectVoice(&voice_select, &voice_found);
 if (v_id == NULL)
  return "default";

 if ((strchr(v_id, '+') == NULL) && ((voice_select.gender ==
ENGENDER_UNKNOWN) || (voice_select.gender == base_voice->gender))
&& (base_voice_variant_name[0] != 0)) {
  // a voice variant has not been selected, use the original
voice variant
  sprintf(buf, "%s+%s", v_id, base_voice_variant_name);
  strncpy0(voice_name, buf, sizeof(voice_name));
  return voice_name;
 }
 return v_id;
}

static wchar_t *GetSsmlAttribute(wchar_t *pw, const char *name)
{
 // Gets the value string for an attribute.
 // Returns NULL if the attribute is not present

 int ix;
 static wchar_t empty[1] = { 0 };

 while (*pw != 0) {
  if (iswspace(pw[-1])) {
   ix = 0;
```

```
   while (*pw == name[ix]) {
    pw++;
    ix++;
   }
   if (name[ix] == 0) {
    // found the attribute, now get the value
    while (iswspace(*pw)) pw++;
    if (*pw == '=') pw++;
    while (iswspace(*pw)) pw++;
    if ((*pw == '"') || (*pw == '\'')) // allow single-quotes ?
     return pw+1;
    else
     return empty;
   }
  }
  pw++;
 }
 return NULL;
}

static int GetVoiceAttributes(wchar_t *pw, int tag_type,
SSML_STACK *ssml_sp, SSML_STACK *ssml_stack, int n_ssml_stack,
char current_voice_id[40], espeak_VOICE *base_voice, char
*base_voice_variant_name)
{
 // Determines whether voice attribute are specified in this tag,
and if so, whether this means
 // a voice change.
 // If it's a closing tag, delete the top frame of the stack and
determine whether this implies
 // a voice change.
 // Returns  CLAUSE_TYPE_VOICE_CHANGE if there is a voice change

 wchar_t *lang;
 wchar_t *gender;
 wchar_t *name;
 wchar_t *age;
```

```
wchar_t *variant;
int value;
const char *new_voice_id;

static const MNEM_TAB mnem_gender[] = {
 { "male", ENGENDER_MALE },
 { "female", ENGENDER_FEMALE },
 { "neutral", ENGENDER_NEUTRAL },
 { NULL, ENGENDER_UNKNOWN }
};

if (tag_type & SSML_CLOSE) {
 // delete a stack frame
 if (n_ssml_stack > 1)
  n_ssml_stack--;
} else {
 // add a stack frame if any voice details are specified
 lang = GetSsmlAttribute(pw, "xml:lang");

 if (tag_type != SSML_VOICE) {
  // only expect an xml:lang attribute
  name = NULL;
  variant = NULL;
  age = NULL;
  gender = NULL;
 } else {
  name = GetSsmlAttribute(pw, "name");
  variant = GetSsmlAttribute(pw, "variant");
  age = GetSsmlAttribute(pw, "age");
  gender = GetSsmlAttribute(pw, "gender");
 }

 if ((tag_type != SSML_VOICE) && (lang == NULL))
  return 0; // <s> or <p> without language spec, nothing to do

 ssml_sp = &ssml_stack[n_ssml_stack++];
```

```
  attrcopy_utf8(ssml_sp->language, lang,
sizeof(ssml_sp->language));
  attrcopy_utf8(ssml_sp->voice_name, name,
sizeof(ssml_sp->voice_name));
  if ((value = attrnumber(variant, 1, 0)) > 0)
   value--; // variant='0' and variant='1' the same
  ssml_sp->voice_variant_number = value;
  ssml_sp->voice_age = attrnumber(age, 0, 0);
  ssml_sp->voice_gender = attrlookup(gender, mnem_gender);
  ssml_sp->tag_type = tag_type;
 }

 new_voice_id = VoiceFromStack(ssml_stack, n_ssml_stack,
base_voice, base_voice_variant_name);
 if (strcmp(new_voice_id, current_voice_id) != 0) {
  // add an embedded command to change the voice
  strcpy(current_voice_id, new_voice_id);
  return CLAUSE_TYPE_VOICE_CHANGE;
 }

 return 0;
}

static void ProcessParamStack(char *outbuf, int *outix, int
n_param_stack, PARAM_STACK *param_stack, int *speech_parameters)
{
 // Set the speech parameters from the parameter stack
 int param;
 int ix;
 int value;
 char buf[20];
 int new_parameters[N_SPEECH_PARAM];
 static char cmd_letter[N_SPEECH_PARAM] = { 0, 'S', 'A', 'P',
'R', 0, 'C', 0, 0, 0, 0, 0, 'F' }; // embedded command letters

 for (param = 0; param < N_SPEECH_PARAM; param++)
  new_parameters[param] = -1;
```

```
 for (ix = 0; ix < n_param_stack; ix++) {
  for (param = 0; param < N_SPEECH_PARAM; param++) {
   if (param_stack[ix].parameter[param] >= 0)
    new_parameters[param] = param_stack[ix].parameter[param];
  }
 }

 for (param = 0; param < N_SPEECH_PARAM; param++) {
  if ((value = new_parameters[param]) !=
speech_parameters[param]) {
   buf[0] = 0;

   switch (param)
   {
   case espeakPUNCTUATION:
    option_punctuation = value-1;
    break;
   case espeakCAPITALS:
    option_capitals = value;
    break;
   case espeakRATE:
   case espeakVOLUME:
   case espeakPITCH:
   case espeakRANGE:
   case espeakEMPHASIS:
    sprintf(buf, "%c%d%c", CTRL_EMBEDDED, value,
cmd_letter[param]);
    break;
   }

   speech_parameters[param] = new_parameters[param];
   strcpy(&outbuf[*outix], buf);
   *outix += strlen(buf);
  }
 }
}
```

```
static PARAM_STACK *PushParamStack(int tag_type, int
*n_param_stack, PARAM_STACK *param_stack)
{
 int ix;
 PARAM_STACK *sp;

 sp = &param_stack[*n_param_stack];
 if (*n_param_stack < (N_PARAM_STACK-1))
  (*n_param_stack)++;

 sp->type = tag_type;
 for (ix = 0; ix < N_SPEECH_PARAM; ix++)
  sp->parameter[ix] = -1;
 return sp;
}

static void PopParamStack(int tag_type, char *outbuf, int *outix,
int *n_param_stack, PARAM_STACK *param_stack, int
*speech_parameters)
{
 // unwind the stack up to and including the previous tag of this
type
 int ix;
 int top = 0;

 if (tag_type >= SSML_CLOSE)
  tag_type -= SSML_CLOSE;

 for (ix = 0; ix < *n_param_stack; ix++) {
  if (param_stack[ix].type == tag_type)
   top = ix;
 }
 if (top > 0)
  *n_param_stack = top;
 ProcessParamStack(outbuf, outix, *n_param_stack, param_stack,
speech_parameters);
```

```
}

static int ReplaceKeyName(char *outbuf, int index, int *outix)
{
 // Replace some key-names by single characters, so they can be
pronounced in different languages
 static MNEM_TAB keynames[] = {
  { "space ",        0xe020 },
  { "tab ",          0xe009 },
  { "underscore ",   0xe05f },
  { "double-quote ", '"' },
  { NULL,            0 }
 };

 int ix;
 int letter;
 char *p;

 p = &outbuf[index];

 if ((letter = LookupMnem(keynames, p)) != 0) {
  ix = utf8_out(letter, p);
  *outix = index + ix;
  return letter;
 }
 return 0;
}

static void SetProsodyParameter(int param_type, wchar_t *attr1,
PARAM_STACK *sp, PARAM_STACK *param_stack, int
*speech_parameters)
{
 int value;
 int sign;

 static const MNEM_TAB mnem_volume[] = {
  { "default", 100 },
```

```
 { "silent",    0 },
 { "x-soft",   30 },
 { "soft",     65 },
 { "medium",  100 },
 { "loud",    150 },
 { "x-loud",  230 },
 { NULL,       -1 }
};

static const MNEM_TAB mnem_rate[] = {
 { "default", 100 },
 { "x-slow",   60 },
 { "slow",     80 },
 { "medium",  100 },
 { "fast",    125 },
 { "x-fast",  160 },
 { NULL,       -1 }
};

static const MNEM_TAB mnem_pitch[] = {
 { "default", 100 },
 { "x-low",    70 },
 { "low",      85 },
 { "medium",  100 },
 { "high",    110 },
 { "x-high",  120 },
 { NULL,       -1 }
};

static const MNEM_TAB mnem_range[] = {
 { "default", 100 },
 { "x-low",    20 },
 { "low",      50 },
 { "medium",  100 },
 { "high",    140 },
 { "x-high",  180 },
 { NULL,       -1 }
```

```
 };

 static const MNEM_TAB *mnem_tabs[5] = {
  NULL, mnem_rate, mnem_volume, mnem_pitch, mnem_range
 };

 if ((value = attrlookup(attr1, mnem_tabs[param_type])) >= 0) {
  // mnemonic specifies a value as a percentage of the base
pitch/range/rate/volume
  sp->parameter[param_type] =
(param_stack[0].parameter[param_type] * value)/100;
 } else {
  sign = attr_prosody_value(param_type, attr1, &value);

  if (sign == 0)
   sp->parameter[param_type] = value; // absolute value in Hz
  else if (sign == 2) {
   // change specified as percentage or in semitones
   sp->parameter[param_type] = (speech_parameters[param_type] *
value)/100;
  } else {
   // change specified as plus or minus Hz
   sp->parameter[param_type] = speech_parameters[param_type] +
(value*sign);
  }
 }
}

int ProcessSsmlTag(wchar_t *xml_buf, char *outbuf, int *outix,
int n_outbuf, bool self_closing, const char *xmlbase, bool
*audio_text, char *current_voice_id, espeak_VOICE *base_voice,
char *base_voice_variant_name, bool *ignore_text, bool
*clear_skipping_text, int *sayas_mode, int *sayas_start,
SSML_STACK *ssml_stack, int *n_ssml_stack, int *n_param_stack,
int *speech_parameters)
{
 // xml_buf is the tag and attributes with a zero terminator in
```

```
place of the original '>'
 // returns a clause terminator value.

 unsigned int ix;
 int index;
 int c;
 int tag_type;
 int value;
 int value2;
 int value3;
 int voice_change_flag;
 wchar_t *px;
 wchar_t *attr1;
 wchar_t *attr2;
 wchar_t *attr3;
 int terminator;
 char *uri;
 int param_type;
 char tag_name[40];
 char buf[80];
 PARAM_STACK *sp;
 SSML_STACK *ssml_sp;

 // these tags have no effect if they are self-closing, eg.
<voice />
 static char ignore_if_self_closing[] = { 0, 1, 1, 1, 1, 0, 0, 0,
0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0 };

 static const MNEM_TAB mnem_phoneme_alphabet[] = {
  { "espeak", 1 },
  { NULL,    -1 }
 };

 static const MNEM_TAB mnem_punct[] = {
  { "none", 1 },
  { "all",  2 },
  { "some", 3 },
```

```
 { NULL,  -1 }
};

static const MNEM_TAB mnem_capitals[] = {
 { "no",         0 },
 { "icon",       1 },
 { "spelling",  2 },
 { "pitch",    20 },  // this is the amount by which to raise
the pitch
 { NULL,       -1 }
};

static const MNEM_TAB mnem_interpret_as[] = {
 { "characters", SAYAS_CHARS },
 { "tts:char",   SAYAS_SINGLE_CHARS },
 { "tts:key",    SAYAS_KEY },
 { "tts:digits", SAYAS_DIGITS },
 { "telephone",  SAYAS_DIGITS1 },
 { NULL,         -1 }
};

static const MNEM_TAB mnem_sayas_format[] = {
 { "glyphs", 1 },
 { NULL,    -1 }
};

static const MNEM_TAB mnem_break[] = {
 { "none",     0 },
 { "x-weak",   1 },
 { "weak",     2 },
 { "medium",   3 },
 { "strong",   4 },
 { "x-strong", 5 },
 { NULL,      -1 }
};

static const MNEM_TAB mnem_emphasis[] = {
```

```
  { "none",     1 },
  { "reduced",  2 },
  { "moderate", 3 },
  { "strong",   4 },
  { "x-strong", 5 },
  { NULL,      -1 }
 };

 static const char *prosody_attr[5] = {
  NULL, "rate", "volume", "pitch", "range"
 };

 for (ix = 0; ix < (sizeof(tag_name)-1); ix++) {
  if (((c = xml_buf[ix]) == 0) || iswspace(c))
   break;
  tag_name[ix] = tolower((char)c);
 }
 tag_name[ix] = 0;

 px = &xml_buf[ix]; // the tag's attributes

 if (tag_name[0] == '/') {
  // closing tag
  if ((tag_type = LookupMnem(ssmltags, &tag_name[1])) !=
HTML_NOSPACE)
   outbuf[(*outix)++] = ' ';
  tag_type += SSML_CLOSE;
 } else {
  if ((tag_type = LookupMnem(ssmltags, tag_name)) !=
HTML_NOSPACE) {
   // separate SSML tags from the previous word (but not HMTL
tags such as <b> <font> which can occur inside a word)
   outbuf[(*outix)++] = ' ';
  }

  if (self_closing && ignore_if_self_closing[tag_type])
   return 0;
```

```
 }

 voice_change_flag = 0;
 ssml_sp = &ssml_stack[*n_ssml_stack-1];

 switch (tag_type)
 {
 case SSML_STYLE:
  sp = PushParamStack(tag_type, n_param_stack, (PARAM_STACK *)
param_stack);
  attr1 = GetSsmlAttribute(px, "field");
  attr2 = GetSsmlAttribute(px, "mode");

  if (attrcmp(attr1, "punctuation") == 0) {
   value = attrlookup(attr2, mnem_punct);
   sp->parameter[espeakPUNCTUATION] = value;
  } else if (attrcmp(attr1, "capital_letters") == 0) {
   value = attrlookup(attr2, mnem_capitals);
   sp->parameter[espeakCAPITALS] = value;
  }
  ProcessParamStack(outbuf, outix, *n_param_stack, param_stack,
speech_parameters);
  break;
 case SSML_PROSODY:
  sp = PushParamStack(tag_type, n_param_stack, (PARAM_STACK *)
param_stack);

  // look for attributes:  rate, volume, pitch, range
  for (param_type = espeakRATE; param_type <= espeakRANGE;
param_type++) {
   if ((attr1 = GetSsmlAttribute(px, prosody_attr[param_type]))
!= NULL)
    SetProsodyParameter(param_type, attr1, sp, param_stack,
speech_parameters);
  }

  ProcessParamStack(outbuf, outix, *n_param_stack, param_stack,
```

```
speech_parameters);
  break;
 case SSML_EMPHASIS:
  sp = PushParamStack(tag_type, n_param_stack, (PARAM_STACK *)
param_stack);
  value = 3; // default is "moderate"
  if ((attr1 = GetSsmlAttribute(px, "level")) != NULL)
   value = attrlookup(attr1, mnem_emphasis);

  if (translator->langopts.tone_language == 1) {
   static unsigned char emphasis_to_pitch_range[] = { 50, 50, 40,
70, 90, 100 };
   static unsigned char emphasis_to_volume[] = { 100, 100, 70,
110, 135, 150 };
   // tone language (eg.Chinese) do emphasis by increasing the
pitch range.
   sp->parameter[espeakRANGE] = emphasis_to_pitch_range[value];
   sp->parameter[espeakVOLUME] = emphasis_to_volume[value];
  } else {
   static unsigned char emphasis_to_volume2[] = { 100, 100, 75,
100, 120, 150 };
   sp->parameter[espeakVOLUME] = emphasis_to_volume2[value];
   sp->parameter[espeakEMPHASIS] = value;
  }
  ProcessParamStack(outbuf, outix, *n_param_stack, param_stack,
speech_parameters);
  break;
 case SSML_STYLE + SSML_CLOSE:
 case SSML_PROSODY + SSML_CLOSE:
 case SSML_EMPHASIS + SSML_CLOSE:
  PopParamStack(tag_type, outbuf, outix, n_param_stack,
(PARAM_STACK *) param_stack, (int *) speech_parameters);
  break;
 case SSML_PHONEME:
  attr1 = GetSsmlAttribute(px, "alphabet");
  attr2 = GetSsmlAttribute(px, "ph");
  value = attrlookup(attr1, mnem_phoneme_alphabet);
```

```
  if (value == 1) { // alphabet="espeak"
   outbuf[(*outix)++] = '[';
   *outix += attrcopy_utf8(&outbuf[*outix], attr2,
n_outbuf-*outix);
   outbuf[(*outix)++] = ']';
  }
  break;
 case SSML_SAYAS:
  attr1 = GetSsmlAttribute(px, "interpret-as");
  attr2 = GetSsmlAttribute(px, "format");
  attr3 = GetSsmlAttribute(px, "detail");
  value = attrlookup(attr1, mnem_interpret_as);
  value2 = attrlookup(attr2, mnem_sayas_format);
  if (value2 == 1)
   value = SAYAS_GLYPHS;

  value3 = attrnumber(attr3, 0, 0);

  if (value == SAYAS_DIGITS) {
   if (value3 <= 1)
    value = SAYAS_DIGITS1;
   else
    value = SAYAS_DIGITS + value3;
  }

  sprintf(buf, "%c%dY", CTRL_EMBEDDED, value);
  strcpy(&outbuf[*outix], buf);
  *outix += strlen(buf);

  *sayas_start = *outix;
  *sayas_mode = value; // punctuation doesn't end clause during
SAY-AS
  break;
 case SSML_SAYAS + SSML_CLOSE:
  if (*sayas_mode == SAYAS_KEY) {
   outbuf[*outix] = 0;
   ReplaceKeyName(outbuf, *sayas_start, outix);
```

```
  }

  outbuf[(*outix)++] = CTRL_EMBEDDED;
  outbuf[(*outix)++] = 'Y';
  *sayas_mode = 0;
  break;
 case SSML_SUB:
  if ((attr1 = GetSsmlAttribute(px, "alias")) != NULL) {
   // use the alias  rather than the text
   *ignore_text = true;
   *outix += attrcopy_utf8(&outbuf[*outix], attr1,
n_outbuf-*outix);
  }
  break;
 case SSML_IGNORE_TEXT:
  *ignore_text = true;
  break;
 case SSML_SUB + SSML_CLOSE:
 case SSML_IGNORE_TEXT + SSML_CLOSE:
  *ignore_text = false;
  break;
 case SSML_MARK:
  if ((attr1 = GetSsmlAttribute(px, "name")) != NULL) {
   // add name to circular buffer of marker names
   attrcopy_utf8(buf, attr1, sizeof(buf));

   if (strcmp(skip_marker, buf) == 0) {
    // This is the marker we are waiting for before starting to
speak
    *clear_skipping_text = true;
    skip_marker[0] = 0;
    return CLAUSE_NONE;
   }

   if ((index = AddNameData(buf, 0)) >= 0) {
    sprintf(buf, "%c%dM", CTRL_EMBEDDED, index);
    strcpy(&outbuf[*outix], buf);
```

```
    *outix += strlen(buf);
   }
  }
  break;
 case SSML_AUDIO:
  sp = PushParamStack(tag_type, n_param_stack, (PARAM_STACK
*)param_stack);

  if ((attr1 = GetSsmlAttribute(px, "src")) != NULL) {
   char fname[256];
   attrcopy_utf8(buf, attr1, sizeof(buf));

   if (uri_callback == NULL) {
    if ((xmlbase != NULL) && (buf[0] != '/')) {
     sprintf(fname, "%s/%s", xmlbase, buf);
     index = LoadSoundFile2(fname);
    } else
     index = LoadSoundFile2(buf);
    if (index >= 0) {
     sprintf(buf, "%c%dI", CTRL_EMBEDDED, index);
     strcpy(&outbuf[*outix], buf);
     *outix += strlen(buf);
     sp->parameter[espeakSILENCE] = 1;
    }
   } else {
    if ((index = AddNameData(buf, 0)) >= 0) {
     uri = &namedata[index];
     if (uri_callback(1, uri, xmlbase) == 0) {
      sprintf(buf, "%c%dU", CTRL_EMBEDDED, index);
      strcpy(&outbuf[*outix], buf);
      *outix += strlen(buf);
      sp->parameter[espeakSILENCE] = 1;
     }
    }
   }
  }
  ProcessParamStack(outbuf, outix, *n_param_stack, param_stack,
```

```
speech_parameters);

  if (self_closing)
   PopParamStack(tag_type, outbuf, outix, n_param_stack,
(PARAM_STACK *) param_stack, (int *) speech_parameters);
  else
   *audio_text = true;
  return CLAUSE_NONE;
 case SSML_AUDIO + SSML_CLOSE:
  PopParamStack(tag_type, outbuf, outix, n_param_stack,
(PARAM_STACK *) param_stack, (int *) speech_parameters);
  *audio_text = false;
  return CLAUSE_NONE;
 case SSML_BREAK:
  value = 21;
  terminator = CLAUSE_NONE;

  if ((attr1 = GetSsmlAttribute(px, "strength")) != NULL) {
   static int break_value[6] = { 0, 7, 14, 21, 40, 80 }; // *10mS
   value = attrlookup(attr1, mnem_break);
   if (value < 3) {
    // adjust prepause on the following word
    sprintf(&outbuf[*outix], "%c%dB", CTRL_EMBEDDED, value);
    *outix += 3;
    terminator = 0;
   }
   value = break_value[value];
  }
  if ((attr2 = GetSsmlAttribute(px, "time")) != NULL) {
   value2 = attrnumber(attr2, 0, 1);   // pause in mS

   // compensate for speaking speed to keep constant pause
length, see function PauseLength()
   // 'value' here is x 10mS
   value = (value2 * 256) / (speed.clause_pause_factor * 10);
   if (value < 200)
    value = (value2 * 256) / (speed.pause_factor * 10);
```

```
   if (terminator == 0)
    terminator = CLAUSE_NONE;
  }
  if (terminator) {
   if (value > 0xfff) {
    // scale down the value and set a scaling indicator bit
    value = value / 32;
    if (value > 0xfff)
     value = 0xfff;
    terminator |= CLAUSE_PAUSE_LONG;
   }
   return terminator + value;
  }
  break;
 case SSML_SPEAK:
  if ((attr1 = GetSsmlAttribute(px, "xml:base")) != NULL) {
   attrcopy_utf8(buf, attr1, sizeof(buf));
   if ((index = AddNameData(buf, 0)) >= 0)
    xmlbase = &namedata[index];
  }
  if (GetVoiceAttributes(px, tag_type, ssml_sp, ssml_stack,
*n_ssml_stack, current_voice_id, base_voice,
base_voice_variant_name) == 0)
   return 0; // no voice change
  return CLAUSE_VOICE;
 case SSML_VOICE:
  if (GetVoiceAttributes(px, tag_type, ssml_sp, ssml_stack,
*n_ssml_stack, current_voice_id, base_voice,
base_voice_variant_name) == 0)
   return 0; // no voice change
  return CLAUSE_VOICE;
 case SSML_SPEAK + SSML_CLOSE:
  // unwind stack until the previous <voice> or <speak> tag
  while ((*n_ssml_stack > 1) &&
(ssml_stack[*n_ssml_stack-1].tag_type != SSML_SPEAK))
   (*n_ssml_stack)--;
```

```
  return CLAUSE_PERIOD + GetVoiceAttributes(px, tag_type,
ssml_sp, ssml_stack, *n_ssml_stack, current_voice_id, base_voice,
base_voice_variant_name);
 case SSML_VOICE + SSML_CLOSE:
  // unwind stack until the previous <voice> or <speak> tag
  while ((*n_ssml_stack > 1) &&
(ssml_stack[*n_ssml_stack-1].tag_type != SSML_VOICE))
   (*n_ssml_stack)--;

  terminator = 0; // ??  Sentence intonation, but no pause ??
  return terminator + GetVoiceAttributes(px, tag_type, ssml_sp,
ssml_stack, *n_ssml_stack, current_voice_id, base_voice,
base_voice_variant_name);
 case HTML_BREAK:
 case HTML_BREAK + SSML_CLOSE:
  return CLAUSE_COLON;
 case SSML_SENTENCE:
  if (ssml_sp->tag_type == SSML_SENTENCE) {
   // new sentence implies end-of-sentence
   voice_change_flag = GetVoiceAttributes(px,
SSML_SENTENCE+SSML_CLOSE, ssml_sp, ssml_stack, *n_ssml_stack,
current_voice_id, base_voice, base_voice_variant_name);
  }
  voice_change_flag |= GetVoiceAttributes(px, tag_type, ssml_sp,
ssml_stack, *n_ssml_stack, current_voice_id, base_voice,
base_voice_variant_name);
  return CLAUSE_PARAGRAPH + voice_change_flag;
 case SSML_PARAGRAPH:
  if (ssml_sp->tag_type == SSML_SENTENCE) {
   // new paragraph implies end-of-sentence or end-of-paragraph
   voice_change_flag = GetVoiceAttributes(px,
SSML_SENTENCE+SSML_CLOSE, ssml_sp, ssml_stack, *n_ssml_stack,
current_voice_id, base_voice, base_voice_variant_name);
  }
  if (ssml_sp->tag_type == SSML_PARAGRAPH) {
   // new paragraph implies end-of-sentence or end-of-paragraph
   voice_change_flag |= GetVoiceAttributes(px,
```

```
SSML_PARAGRAPH+SSML_CLOSE, ssml_sp, ssml_stack, *n_ssml_stack,
current_voice_id, base_voice, base_voice_variant_name);
  }
  voice_change_flag |= GetVoiceAttributes(px, tag_type, ssml_sp,
ssml_stack, *n_ssml_stack, current_voice_id, base_voice,
base_voice_variant_name);
  return CLAUSE_PARAGRAPH + voice_change_flag;
 case SSML_SENTENCE + SSML_CLOSE:
  if (ssml_sp->tag_type == SSML_SENTENCE) {
   // end of a sentence which specified a language
   voice_change_flag = GetVoiceAttributes(px, tag_type, ssml_sp,
ssml_stack, *n_ssml_stack, current_voice_id, base_voice,
base_voice_variant_name);
  }
  return CLAUSE_PERIOD + voice_change_flag;
 case SSML_PARAGRAPH + SSML_CLOSE:
  if ((ssml_sp->tag_type == SSML_SENTENCE) || (ssml_sp->tag_type
== SSML_PARAGRAPH)) {
   // End of a paragraph which specified a language.
   // (End-of-paragraph also implies end-of-sentence)
   return GetVoiceAttributes(px, tag_type, ssml_sp, ssml_stack,
*n_ssml_stack, current_voice_id, base_voice,
base_voice_variant_name) + CLAUSE_PARAGRAPH;
  }
  return CLAUSE_PARAGRAPH;
 }
 return 0;
}
```

Chapter 54

./src/libespeak-ng/spect.c

```
#include "config.h"

#include <errno.h>
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <endian.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>

#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "spect.h"
#include "ieee80.h"

static int frame_width;

static int default_freq[N_PEAKS] =
```

```
{ 200, 500, 1200, 3000, 3500, 4000, 6900, 7800, 9000 };
static int default_width[N_PEAKS] =
{ 750, 500, 550, 550, 600, 700, 700, 700, 700 };
static int default_klt_bw[N_PEAKS] =
{ 89, 90, 140, 260, 260, 260, 500, 500, 500 };

static double read_double(FILE *stream)
{
 unsigned char bytes[10];
 fread(bytes, sizeof(char), 10, stream);
 return ConvertFromIeeeExtended((char *)bytes);
}

float polint(float xa[], float ya[], int n, float x)
{
 // General polinomial interpolation routine, xa[1...n] ya[1...n]
 int i, m, ns = 1;
 float den, dif, dift, ho, hp, w;
 float y; // result
 float c[9], d[9];

 dif = fabs(x-xa[1]);

 for (i = 1; i <= n; i++) {
  if ((dift = fabs(x-xa[i])) < dif) {
   ns = i;
   dif = dift;
  }
  c[i] = ya[i];
  d[i] = ya[i];
 }
 y = ya[ns--];
 for (m = 1; m < n; m++) {
  for (i = 1; i <= n-m; i++) {
   ho = xa[i]-x;
   hp = xa[i+m]-x;
   w = c[i+1]-d[i];
```

```
    if ((den = ho-hp) == 0.0)
     return ya[2]; // two input xa are identical
    den = w/den;
    d[i] = hp*den;
    c[i] = ho*den;
   }
   y += ((2*ns < (n-m) ? c[ns+1] : d[ns--]));
  }
  return y;
}

static SpectFrame *SpectFrameCreate()
{
  int ix;
  SpectFrame *frame;

  frame = malloc(sizeof(SpectFrame));
  if (!frame)
   return NULL;

  frame->keyframe = 0;
  frame->spect = NULL;
  frame->markers = 0;
  frame->pitch = 0;
  frame->nx = 0;
  frame->time = 0;
  frame->length = 0;
  frame->amp_adjust = 100;
  frame->length_adjust = 0;

  for (ix = 0; ix < N_PEAKS; ix++) {
   frame->formants[ix].freq = 0;
   frame->peaks[ix].pkfreq = default_freq[ix];
   frame->peaks[ix].pkheight = 0;
   frame->peaks[ix].pkwidth = default_width[ix];
   frame->peaks[ix].pkright = default_width[ix];
   frame->peaks[ix].klt_bw = default_klt_bw[ix];
```

```
  frame->peaks[ix].klt_ap = 0;
  frame->peaks[ix].klt_bp = default_klt_bw[ix];
 }

 memset(frame->klatt_param, 0, sizeof(frame->klatt_param));
 frame->klatt_param[KLATT_AV] = 59;
 frame->klatt_param[KLATT_Kopen] = 40;

 return frame;
}

static void SpectFrameDestroy(SpectFrame *frame)
{
 if (frame->spect != NULL)
  free(frame->spect);
 free(frame);
}

static espeak_ng_STATUS LoadFrame(SpectFrame *frame, FILE
*stream, int file_format_type)
{
 short ix;
 short x;
 unsigned short *spect_data;

 frame->time = read_double(stream);
 frame->pitch = read_double(stream);
 frame->length = read_double(stream);
 frame->dx = read_double(stream);
 fread(&frame->nx, sizeof(short), 1, stream);
 fread(&frame->markers, sizeof(short), 1, stream);
 fread(&frame->amp_adjust, sizeof(short), 1, stream);
 frame->nx = le16toh(frame->nx);
 frame->markers = le16toh(frame->markers);
 frame->amp_adjust = le16toh(frame->amp_adjust);

 if (file_format_type == 2) {
```

```
 fread(&ix, sizeof(short), 1, stream); // spare
}

for (ix = 0; ix < N_PEAKS; ix++) {
 fread(&frame->formants[ix].freq, sizeof(short), 1, stream);
 fread(&frame->formants[ix].bandw, sizeof(short), 1, stream);
 fread(&frame->peaks[ix].pkfreq, sizeof(short), 1, stream);
 fread(&frame->peaks[ix].pkheight, sizeof(short), 1, stream);
 fread(&frame->peaks[ix].pkwidth, sizeof(short), 1, stream);
 fread(&frame->peaks[ix].pkright, sizeof(short), 1, stream);
 frame->formants[ix].freq = le16toh(frame->formants[ix].freq);
 frame->formants[ix].bandw = le16toh(frame->formants[ix].bandw);
 frame->peaks[ix].pkfreq = le16toh(frame->peaks[ix].pkfreq);
 frame->peaks[ix].pkheight = le16toh(frame->peaks[ix].pkheight);
 frame->peaks[ix].pkwidth = le16toh(frame->peaks[ix].pkwidth);
 frame->peaks[ix].pkright = le16toh(frame->peaks[ix].pkright);
 if (frame->peaks[ix].pkheight > 0)
  frame->keyframe = 1;

 if (file_format_type == 2) {
  fread(&frame->peaks[ix].klt_bw, sizeof(short), 1, stream);
  fread(&frame->peaks[ix].klt_ap, sizeof(short), 1, stream);
  fread(&frame->peaks[ix].klt_bp, sizeof(short), 1, stream);
  frame->peaks[ix].klt_bw = le16toh(frame->peaks[ix].klt_bw);
  frame->peaks[ix].klt_ap = le16toh(frame->peaks[ix].klt_ap);
  frame->peaks[ix].klt_bp = le16toh(frame->peaks[ix].klt_bp);
 }
}

if (file_format_type > 0) {
 for (ix = 0; ix < N_KLATTP2; ix++)
 {
  fread(frame->klatt_param + ix, sizeof(short), 1, stream);
  frame->klatt_param[ix] = le16toh(frame->klatt_param[ix]);
 }
}
```

```
 spect_data = malloc(sizeof(unsigned short) * frame->nx);

 if (spect_data == NULL)
  return ENOMEM;

 frame->max_y = 0;
 for (ix = 0; ix < frame->nx; ix++) {
  fread(&x, sizeof(short), 1, stream);
  x = le16toh(x);
  spect_data[ix] = x;
  if (x > frame->max_y) frame->max_y = x;
 }
 frame->spect = spect_data;

 return ENS_OK;
}

double GetFrameRms(SpectFrame *frame, int seq_amplitude)
{
 int h;
 float total = 0;
 int maxh;
 int height;
 int htab[400];
 wavegen_peaks_t wpeaks[9];

 for (h = 0; h < 9; h++) {
  height = (frame->peaks[h].pkheight * seq_amplitude *
frame->amp_adjust)/10000;
  wpeaks[h].height = height << 8;

  wpeaks[h].freq = frame->peaks[h].pkfreq << 16;
  wpeaks[h].left = frame->peaks[h].pkwidth << 16;
  wpeaks[h].right = frame->peaks[h].pkright << 16;
 }

 maxh = PeaksToHarmspect(wpeaks, 90<<16, htab, 0);
```

```
 for (h = 1; h < maxh; h++)
  total += ((htab[h] * htab[h]) >> 10);
 frame->rms = sqrt(total) / 7.25;
 return frame->rms;
}

#pragma GCC visibility push(default)
SpectSeq *SpectSeqCreate()
{
 SpectSeq *spect = malloc(sizeof(SpectSeq));
 if (!spect)
  return NULL;

 spect->numframes = 0;
 spect->frames = NULL;
 spect->name = NULL;

 spect->grid = 1;
 spect->duration = 0;
 spect->pitch1 = 0;
 spect->pitch2 = 0;
 spect->bass_reduction = 0;

 spect->max_x = 3000;
 spect->max_y = 1;
 spect->file_format = 0;

 return spect;
}

void SpectSeqDestroy(SpectSeq *spect)
{
 int ix;
 if (spect->frames != NULL) {
  for (ix = 0; ix < spect->numframes; ix++) {
   if (spect->frames[ix] != NULL)
    SpectFrameDestroy(spect->frames[ix]);
```

```
  }
  free(spect->frames);
 }
 free(spect->name);
 free(spect);
}
#pragma GCC visibility pop

static float GetFrameLength(SpectSeq *spect, int frame)
{
 int ix;
 float adjust = 0;

 if (frame >= spect->numframes-1) return 0;

 for (ix = frame+1; ix < spect->numframes-1; ix++) {
  if (spect->frames[ix]->keyframe)
   break; // reached next keyframe
  adjust += spect->frames[ix]->length_adjust;
 }
 return (spect->frames[ix]->time - spect->frames[frame]->time) *
1000.0 + adjust;
}

#pragma GCC visibility push(default)
espeak_ng_STATUS LoadSpectSeq(SpectSeq *spect, const char
*filename)
{
 short n, temp;
 int ix;
 uint32_t id1, id2, name_len;
 int set_max_y = 0;
 float time_offset;

 FILE *stream = fopen(filename, "rb");
 if (stream == NULL) {
  fprintf(stderr, "Failed to open: '%s'", filename);
```

```
 return errno;
}

fread(&id1, sizeof(uint32_t), 1, stream);
id1 = le32toh(id1);
fread(&id2, sizeof(uint32_t), 1, stream);
id2 = le32toh(id2);

if ((id1 == FILEID1_SPECTSEQ) && (id2 == FILEID2_SPECTSEQ))
 spect->file_format = 0; // eSpeak formants
else if ((id1 == FILEID1_SPECTSEQ) && (id2 == FILEID2_SPECTSEK))
 spect->file_format = 1; // formants for Klatt synthesizer
else if ((id1 == FILEID1_SPECTSEQ) && (id2 == FILEID2_SPECTSQ2))
 spect->file_format = 2; // formants for Klatt synthesizer
else {
 fprintf(stderr, "Unsupported spectral file format.\n");
 fclose(stream);
 return ENS_UNSUPPORTED_PHON_FORMAT;
}

fread(&name_len, sizeof(uint32_t), 1, stream);
name_len = le32toh(name_len);
if (name_len > 0) {
 if ((spect->name = (char *)malloc(name_len)) == NULL) {
  fclose(stream);
  return ENOMEM;
 }
 fread(spect->name, sizeof(char), name_len, stream);
} else
 spect->name = NULL;

fread(&n, sizeof(short), 1, stream);
fread(&spect->amplitude, sizeof(short), 1, stream);
fread(&spect->max_y, sizeof(short), 1, stream);
fread(&temp, sizeof(short), 1, stream); // unused
n = le16toh(n);
spect->amplitude = le16toh(spect->amplitude);
```

```
 spect->max_y = le16toh(spect->max_y);
 temp = le16toh(temp);

 if (n == 0) {
  fclose(stream);
  return ENS_NO_SPECT_FRAMES;
 }

 if (spect->frames != NULL) {
  for (ix = 0; ix < spect->numframes; ix++) {
   if (spect->frames[ix] != NULL)
    SpectFrameDestroy(spect->frames[ix]);
  }
  free(spect->frames);
 }
 spect->frames = calloc(n, sizeof(SpectFrame *));

 spect->numframes = 0;
 spect->max_x = 3000;
 if (spect->max_y == 0) {
  set_max_y = 1;
  spect->max_y = 1;
 }
 for (ix = 0; ix < n; ix++) {
  SpectFrame *frame = SpectFrameCreate();
  if (!frame) {
   fclose(stream);
   return ENOMEM;
  }

  espeak_ng_STATUS status = LoadFrame(frame, stream,
spect->file_format);
  if (status != ENS_OK) {
   free(frame);
   fclose(stream);
   return status;
  }
```

```
  spect->frames[spect->numframes++] = frame;

  if (set_max_y && (frame->max_y > spect->max_y))
   spect->max_y = frame->max_y;
  if (frame->nx * frame->dx > spect->max_x) spect->max_x =
(int)(frame->nx * frame->dx);
 }
 spect->max_x = 9000; // disable auto-xscaling

 frame_width =
(int)((FRAME_WIDTH*spect->max_x)/MAX_DISPLAY_FREQ);
 if (frame_width > FRAME_WIDTH) frame_width = FRAME_WIDTH;

 // start times from zero
 time_offset = spect->frames[0]->time;
 for (ix = 0; ix < spect->numframes; ix++)
  spect->frames[ix]->time -= time_offset;

 spect->pitch1 = spect->pitchenv.pitch1;
 spect->pitch2 = spect->pitchenv.pitch2;
 spect->duration = (int)(spect->frames[spect->numframes-1]->time
* 1000);

 if (spect->max_y < 400)
  spect->max_y = 200;
 else
  spect->max_y = 29000; // disable auto height scaling

 for (ix = 0; ix < spect->numframes; ix++) {
  if (spect->frames[ix]->keyframe)
   spect->frames[ix]->length_adjust = spect->frames[ix]->length -
GetFrameLength(spect, ix);
 }
 fclose(stream);
 return ENS_OK;
}
```

```
#pragma GCC visibility pop
```

Chapter 55

./src/libespeak-ng/synth_mbrola.c

```
#include "config.h"

#include <ctype.h>
#include <errno.h>
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "dictionary.h"
#include "mbrola.h"
#include "readclause.h"
#include "setlengths.h"
#include "synthdata.h"
```

```
#include "wavegen.h"

#include "speech.h"
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

// included here so tests can find these even without OPT_MBROLA
set
int mbrola_delay;
char mbrola_name[20];

#ifdef INCLUDE_MBROLA

#if defined(_WIN32) || defined(_WIN64)
#include <windows.h>
#endif

#include "mbrowrap.h"

static MBROLA_TAB *mbrola_tab = NULL;
static int mbrola_control = 0;
static int mbr_name_prefix = 0;

espeak_ng_STATUS LoadMbrolaTable(const char *mbrola_voice, const
char *phtrans, int *srate)
{
 // Load a phoneme name translation table from espeak-ng-
data/mbrola

 int size;
 int ix;
 int *pw;
 FILE *f_in;
 char path[sizeof(path_home)+15];
```

```
 mbrola_name[0] = 0;
 mbrola_delay = 0;
 mbr_name_prefix = 0;

 if (mbrola_voice == NULL) {
  samplerate = samplerate_native;
  SetParameter(espeakVOICETYPE, 0, 0);
  return ENS_OK;
 }

 if (!load_MBR())
  return ENS_MBROLA_NOT_FOUND;

 sprintf(path, "%s/mbrola/%s", path_home, mbrola_voice);
#ifdef PLATFORM_POSIX
 // if not found, then also look in
 //   usr/share/mbrola/xx, /usr/share/mbrola/xx/xx,
/usr/share/mbrola/voices/xx
 if (GetFileLength(path) <= 0) {
  sprintf(path, "/usr/share/mbrola/%s", mbrola_voice);

  if (GetFileLength(path) <= 0) {
   sprintf(path, "/usr/share/mbrola/%s/%s", mbrola_voice,
mbrola_voice);

   if (GetFileLength(path) <= 0)
    sprintf(path, "/usr/share/mbrola/voices/%s", mbrola_voice);
  }
 }
 close_MBR();
#endif

 if (init_MBR(path) != 0) // initialise the required mbrola voice
  return ENS_MBROLA_VOICE_NOT_FOUND;

 setNoError_MBR(1); // don't stop on phoneme errors
```

```
 // read eSpeak's mbrola phoneme translation data, eg.
en1_phtrans
 sprintf(path, "%s/mbrola_ph/%s", path_home, phtrans);
 size = GetFileLength(path);
 if (size < 0) // size == -errno
  return -size;
 if ((f_in = fopen(path, "rb")) == NULL) {
  int error = errno;
  close_MBR();
  return error;
 }

 MBROLA_TAB *new_mbrola_tab = (MBROLA_TAB *)realloc(mbrola_tab,
size);
 if (new_mbrola_tab == NULL) {
  fclose(f_in);
  close_MBR();
  return ENOMEM;
 }
 mbrola_tab = new_mbrola_tab;

 mbrola_control = Read4Bytes(f_in);
 pw = (int *)mbrola_tab;
 for (ix = 4; ix < size; ix += 4)
  *pw++ = Read4Bytes(f_in);
 fclose(f_in);

 setVolumeRatio_MBR((float)(mbrola_control & 0xff) /16.0f);
 samplerate = *srate = getFreq_MBR();
 if (*srate == 22050)
  SetParameter(espeakVOICETYPE, 0, 0);
 else
  SetParameter(espeakVOICETYPE, 1, 0);
 strcpy(mbrola_name, mbrola_voice);
 mbrola_delay = 1000; // improve synchronization of events
 return ENS_OK;
}
```

```
static int GetMbrName(PHONEME_LIST *plist, PHONEME_TAB *ph,
PHONEME_TAB *ph_prev, PHONEME_TAB *ph_next, int *name2, int
*split, int *control)
{
 // Look up a phoneme in the mbrola phoneme name translation
table
 // It may give none, 1, or 2 mbrola phonemes

 MBROLA_TAB *pr;
 PHONEME_TAB *other_ph;
 bool found = false;
 static int mnem;

 // control
 // bit 0  skip the next phoneme
 // bit 1  match this and Previous phoneme
 // bit 2  only at the start of a word
 // bit 3  don't match two phonemes across a word boundary
 // bit 4  add this phoneme name as a prefix to the next phoneme
name (used for de4 phoneme prefix '?')
 // bit 5  only in stressed syllable
 // bit 6  only at the end of a word

 mnem = ph->mnemonic;

 pr = mbrola_tab;
 while (pr->name != 0) {
  if (mnem == pr->name) {
   if (pr->next_phoneme == 0)
    found = true;
   else if ((pr->next_phoneme == ':') && (plist->synthflags &
SFLAG_LENGTHEN))
    found = true;
   else {
    if (pr->control & 2)
     other_ph = ph_prev;
```

```
    else if ((pr->control & 8) && ((plist+1)->newword))
     other_ph = phoneme_tab[phPAUSE]; // don't match the next
phoneme over a word boundary
    else
     other_ph = ph_next;

    if ((pr->next_phoneme == other_ph->mnemonic) ||
        ((pr->next_phoneme == 2) && (other_ph->type == phVOWEL))
||
        ((pr->next_phoneme == '_') && (other_ph->type ==
phPAUSE)))
     found = true;
   }

   if ((pr->control & 4) && (plist->newword == 0)) // only at
start of word
    found = false;

   if ((pr->control & 0x40) && (plist[1].newword == 0)) // only
at the end of a word
    found = false;

   if ((pr->control & 0x20) && (plist->stresslevel <
plist->wordstress))
    found = false; // only in stressed syllables

   if (found) {
    *name2 = pr->mbr_name2;
    *split = pr->percent;
    *control = pr->control;

    if (pr->control & 0x10) {
     mbr_name_prefix = pr->mbr_name;
     return 0;
    }
    mnem = pr->mbr_name;
    break;
```

```
   }
  }

  pr++;
 }

 if (mbr_name_prefix != 0)
  mnem = (mnem << 8) | (mbr_name_prefix & 0xff);
 mbr_name_prefix = 0;
 return mnem;
}

static char *WritePitch(int env, int pitch1, int pitch2, int
split, int final)
{
 // final=1:  only give the final pitch value.
 int x;
 int ix;
 int pitch_base;
 int pitch_range;
 int p1, p2, p_end;
 unsigned char *pitch_env;
 int max = -1;
 int min = 999;
 int y_max = 0;
 int y_min = 0;
 int env100 = 80; // apply the pitch change only over this
proportion of the mbrola phoneme(s)
 int y2;
 int y[4];
 int env_split;
 char buf[50];
 static char output[50];

 output[0] = 0;
 pitch_env = envelope_data[env];
```

```
SetPitch2(voice, pitch1, pitch2, &pitch_base, &pitch_range);

env_split = (split * 128)/100;
if (env_split < 0)
 env_split = 0-env_split;

// find max and min in the pitch envelope
for (x = 0; x < 128; x++) {
 if (pitch_env[x] > max) {
  max = pitch_env[x];
  y_max = x;
 }
 if (pitch_env[x] < min) {
  min = pitch_env[x];
  y_min = x;
 }
}
// set an additional pitch point half way through the phoneme.
// but look for a maximum or a minimum and use that instead
y[2] = 64;
if ((y_max > 0) && (y_max < 127))
 y[2] = y_max;
if ((y_min > 0) && (y_min < 127))
 y[2] = y_min;
y[1] = y[2] / 2;
y[3] = y[2] + (127 - y[2])/2;

// set initial pitch
p1 = ((pitch_env[0]*pitch_range)>>8) + pitch_base; // Hz << 12
p_end = ((pitch_env[127]*pitch_range)>>8) + pitch_base;

if (split >= 0) {
 sprintf(buf, " 0 %d", p1/4096);
 strcat(output, buf);
}

// don't use intermediate pitch points for linear rise and fall
```

```
 if (env > 1) {
  for (ix = 1; ix < 4; ix++) {
   p2 = ((pitch_env[y[ix]]*pitch_range)>>8) + pitch_base;

   if (split > 0)
    y2 = (y[ix] * env100)/env_split;
   else if (split < 0)
    y2 = ((y[ix]-env_split) * env100)/env_split;
   else
    y2 = (y[ix] * env100)/128;
   if ((y2 > 0) && (y2 <= env100)) {
    sprintf(buf, " %d %d", y2, p2/4096);
    strcat(output, buf);
   }
  }
 }

 p_end = p_end/4096;
 if (split <= 0) {
  sprintf(buf, " %d %d", env100, p_end);
  strcat(output, buf);
 }
 if (env100 < 100) {
  sprintf(buf, " %d %d", 100, p_end);
  strcat(output, buf);
 }
 strcat(output, "\n");

 if (final)
  sprintf(output, "\t100 %d\n", p_end);
 return output;
}

int MbrolaTranslate(PHONEME_LIST *plist, int n_phonemes, bool
resume, FILE *f_mbrola)
{
 // Generate a mbrola pho file
```

```
unsigned int name;
int len;
int len1;
PHONEME_TAB *ph;
PHONEME_TAB *ph_next;
PHONEME_TAB *ph_prev;
PHONEME_LIST *p;
PHONEME_LIST *next;
PHONEME_DATA phdata;
FMT_PARAMS fmtp;
int pause = 0;
bool released;
int name2;
int control;
bool done;
int len_percent;
const char *final_pitch;
char *ptr;
char mbr_buf[120];

static int phix;
static int embedded_ix;
static int word_count;

if (!resume) {
 phix = 1;
 embedded_ix = 0;
 word_count = 0;
}

while (phix < n_phonemes) {
 if (WcmdqFree() < MIN_WCMDQ)
  return 1;

 ptr = mbr_buf;

 p = &plist[phix];
```

```
  next = &plist[phix+1];
  ph = p->ph;
  ph_prev = plist[phix-1].ph;
  ph_next = plist[phix+1].ph;

  if (p->synthflags & SFLAG_EMBEDDED)
   DoEmbedded(&embedded_ix, p->sourceix);

  if (p->newword & PHLIST_START_OF_SENTENCE)
   DoMarker(espeakEVENT_SENTENCE, (p->sourceix & 0x7ff) +
clause_start_char, 0, count_sentences);
  if (p->newword & PHLIST_START_OF_SENTENCE)
   DoMarker(espeakEVENT_WORD, (p->sourceix & 0x7ff) +
clause_start_char, p->sourceix >> 11, clause_start_word +
word_count++);

  name = GetMbrName(p, ph, ph_prev, ph_next, &name2,
&len_percent, &control);
  if (control & 1)
   phix++;

  if (name == 0) {
   phix++;
   continue; // ignore this phoneme
  }

  if ((ph->type == phPAUSE) && (name == ph->mnemonic)) {
   // a pause phoneme, which has not been changed by the
translation
   name = '_';
   len = (p->length * speed.pause_factor)/256;
   if (len == 0)
    len = 1;
  } else
   len = (80 * speed.wav_factor)/256;

  if (ph->code != phonEND_WORD) {
```

```
   char phoneme_name[16];
   WritePhMnemonic(phoneme_name, p->ph, p, option_phoneme_events
& espeakINITIALIZE_PHONEME_IPA, NULL);
   DoPhonemeMarker(espeakEVENT_PHONEME, (p->sourceix & 0x7ff) +
clause_start_char, 0, phoneme_name);
  }

  ptr += sprintf(ptr, "%s\t", WordToString(name));

  if (name2 == '_') {
   // add a pause after this phoneme
   pause = len_percent;
   name2 = 0;
  }

  done = false;
  final_pitch = "";

  switch (ph->type)
  {
  case phVOWEL:
   len = ph->std_length;
   if (p->synthflags & SFLAG_LENGTHEN)
    len += phoneme_tab[phonLENGTHEN]->std_length; // phoneme was
followed by an extra : symbol

   if (ph_next->type == phPAUSE)
    len += 50; // lengthen vowels before a pause
   len = (len * p->length)/256;

   if (name2 == 0) {
    char *pitch = WritePitch(p->env, p->pitch1, p->pitch2, 0, 0);
    ptr += sprintf(ptr, "%d\t%s", len, pitch);
   } else {
    char *pitch;

    pitch = WritePitch(p->env, p->pitch1, p->pitch2, len_percent,
```

```
0);
    len1 = (len * len_percent)/100;
    ptr += sprintf(ptr, "%d\t%s", len1, pitch);

    pitch = WritePitch(p->env, p->pitch1, p->pitch2,
-len_percent, 0);
    ptr += sprintf(ptr, "%s\t%d\t%s", WordToString(name2), len-
len1, pitch);
   }
   done = true;
   break;
  case phSTOP:
   released = false;
   if (next->type == phVOWEL) released = true;
   if (next->type == phLIQUID && !next->newword) released = true;

   if (released == false)
    p->synthflags |= SFLAG_NEXT_PAUSE;
   InterpretPhoneme(NULL, 0, p, &phdata, NULL);
   len = DoSample3(&phdata, 0, -1);

   len = (len * 1000)/samplerate; // convert to mS
   len += PauseLength(p->prepause, 1);
   break;
  case phVSTOP:
   len = (80 * speed.wav_factor)/256;
   break;
  case phFRICATIVE:
   len = 0;
   InterpretPhoneme(NULL, 0, p, &phdata, NULL);
   if (p->synthflags & SFLAG_LENGTHEN)
    len = DoSample3(&phdata, p->length, -1); // play it twice for
[s:] etc.
   len += DoSample3(&phdata, p->length, -1);

   len = (len * 1000)/samplerate; // convert to mS
   break;
```

```
case phNASAL:
 if (next->type != phVOWEL) {
  memset(&fmtp, 0, sizeof(fmtp));
  InterpretPhoneme(NULL, 0, p, &phdata, NULL);
  fmtp.fmt_addr = phdata.sound_addr[pd_FMT];
  len = DoSpect2(p->ph, 0, &fmtp,  p, -1);
  len = (len * 1000)/samplerate;
  if (next->type == phPAUSE)
   len += 50;
  final_pitch = WritePitch(p->env, p->pitch1, p->pitch2, 0, 1);
 }
 break;
case phLIQUID:
 if (next->type == phPAUSE) {
  len += 50;
  final_pitch = WritePitch(p->env, p->pitch1, p->pitch2, 0, 1);
 }
 break;
}

if (!done) {
 if (name2 != 0) {
  len1 = (len * len_percent)/100;
  ptr += sprintf(ptr, "%d\n%s\t", len1, WordToString(name2));
  len -= len1;
 }
 ptr += sprintf(ptr, "%d%s\n", len, final_pitch);
}

if (pause) {
 len += PauseLength(pause, 0);
 ptr += sprintf(ptr, "_ \t%d\n", PauseLength(pause, 0));
 pause = 0;
}

if (f_mbrola)
 fwrite(mbr_buf, 1, (ptr-mbr_buf), f_mbrola); // write .pho to
```

```
a file
  else {
   int res = write_MBR(mbr_buf);
   if (res < 0)
    return 0;  // don't get stuck on error
   if (res == 0)
    return 1;
   wcmdq[wcmdq_tail][0] = WCMD_MBROLA_DATA;
   wcmdq[wcmdq_tail][1] = len;
   WcmdqInc();
  }

  phix++;
 }

 if (!f_mbrola) {
  flush_MBR();

  // flush the mbrola output buffer
  wcmdq[wcmdq_tail][0] = WCMD_MBROLA_DATA;
  wcmdq[wcmdq_tail][1] = 500;
  WcmdqInc();
 }

 return 0;
}

int MbrolaGenerate(PHONEME_LIST *phoneme_list, int *n_ph, bool
resume)
{
 FILE *f_mbrola = NULL;

 if (*n_ph == 0)
  return 0;

 if (option_phonemes & espeakPHONEMES_MBROLA) {
  // send mbrola data to a file, not to the mbrola library
```

```
  f_mbrola = f_trans;
 }

 int  again = MbrolaTranslate(phoneme_list, *n_ph, resume,
f_mbrola);
 if (!again)
  *n_ph = 0;
 return again;
}

int MbrolaFill(int length, bool resume, int amplitude)
{
 // Read audio data from Mbrola (length is in millisecs)

 static int n_samples;
 int req_samples, result;
 int ix;
 short value16;
 int value;

 if (!resume)
  n_samples = samplerate * length / 1000;

 req_samples = (out_end - out_ptr)/2;
 if (req_samples > n_samples)
  req_samples = n_samples;
 result = read_MBR((short *)out_ptr, req_samples);
 if (result <= 0)
  return 0;

 for (ix = 0; ix < result; ix++) {
  value16 = out_ptr[0] + (out_ptr[1] << 8);
  value = value16 * amplitude;
  value = value / 40; // adjust this constant to give a suitable
amplitude for mbrola voices
  if (value > 0x7fff)
   value = 0x7fff;
```

```
  if (value < -0x8000)
   value = 0x8000;
  out_ptr[0] = value;
  out_ptr[1] = value >> 8;
  out_ptr += 2;
 }
 n_samples -= result;
 return n_samples ? 1 : 0;
}

void MbrolaReset(void)
{
 // Reset the Mbrola engine and flush the pending audio

 reset_MBR();
}

#else

// mbrola interface is not compiled, provide dummy functions.

espeak_ng_STATUS LoadMbrolaTable(const char *mbrola_voice, const
char *phtrans, int *srate)
{
 (void)mbrola_voice; // unused parameter
 (void)phtrans; // unused parameter
 (void)srate; // unused parameter
 return ENS_NOT_SUPPORTED;
}

int MbrolaGenerate(PHONEME_LIST *phoneme_list, int *n_ph, bool
resume)
{
 (void)phoneme_list; // unused parameter
 (void)n_ph; // unused parameter
 (void)resume; // unused parameter
 return 0;
```

```
}

int MbrolaFill(int length, bool resume, int amplitude)
{
 (void)length; // unused parameter
 (void)resume; // unused parameter
 (void)amplitude; // unused parameter
 return 0;
}

void MbrolaReset(void)
{
}

#endif
```

Chapter 56

./src/libespeak-ng/synthesize.c

```
#include "config.h"

#include <ctype.h>
#include <errno.h>
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "dictionary.h"
#include "intonation.h"
#include "mbrola.h"
#include "setlengths.h"
#include "synthdata.h"
#include "wavegen.h"
```

```
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

extern FILE *f_log;
static void SmoothSpect(void);

// list of phonemes in a clause
int n_phoneme_list = 0;
PHONEME_LIST phoneme_list[N_PHONEME_LIST+1];

SPEED_FACTORS speed;

static int last_pitch_cmd;
static int last_amp_cmd;
static frame_t  *last_frame;
static int last_wcmdq;
static int pitch_length;
static int amp_length;
static int modn_flags;
static int fmt_amplitude = 0;

static int syllable_start;
static int syllable_end;
static int syllable_centre;

static voice_t *new_voice = NULL;

int n_soundicon_tab = N_SOUNDICON_SLOTS;
SOUND_ICON soundicon_tab[N_SOUNDICON_TAB];

#define RMS_GLOTTAL1 35   // vowel before glottal stop
#define RMS_START 28  // 28

#define VOWEL_FRONT_LENGTH  50
```

```
// a dummy phoneme_list entry which looks like a pause
static PHONEME_LIST next_pause;

const char *WordToString(unsigned int word)
{
 // Convert a phoneme mnemonic word into a string
 int ix;
 static char buf[5];

 for (ix = 0; ix < 4; ix++)
  buf[ix] = word >> (ix*8);
 buf[4] = 0;
 return buf;
}

void SynthesizeInit()
{
 last_pitch_cmd = 0;
 last_amp_cmd = 0;
 last_frame = NULL;
 syllable_centre = -1;

 // initialise next_pause, a dummy phoneme_list entry
 next_pause.type = phPAUSE;
 next_pause.newword = 0;
}

static void EndAmplitude(void)
{
 if (amp_length > 0) {
  if (wcmdq[last_amp_cmd][1] == 0)
   wcmdq[last_amp_cmd][1] = amp_length;
  amp_length = 0;
 }
}

static void EndPitch(int voice_break)
```

```
{
 // posssible end of pitch envelope, fill in the length
 if ((pitch_length > 0) && (last_pitch_cmd >= 0)) {
  if (wcmdq[last_pitch_cmd][1] == 0)
   wcmdq[last_pitch_cmd][1] = pitch_length;
  pitch_length = 0;
 }

 if (voice_break) {
  last_wcmdq = -1;
  last_frame = NULL;
  syllable_end = wcmdq_tail;
  SmoothSpect();
  syllable_centre = -1;
  memset(vowel_transition, 0, sizeof(vowel_transition));
 }
}

static void DoAmplitude(int amp, unsigned char *amp_env)
{
 intptr_t *q;

 last_amp_cmd = wcmdq_tail;
 amp_length = 0; // total length of vowel with this amplitude
envelope

 q = wcmdq[wcmdq_tail];
 q[0] = WCMD_AMPLITUDE;
 q[1] = 0; // fill in later from amp_length
 q[2] = (intptr_t)amp_env;
 q[3] = amp;
 WcmdqInc();
}

static void DoPitch(unsigned char *env, int pitch1, int pitch2)
{
 intptr_t *q;
```

```
 EndPitch(0);

 if (pitch1 == 255) {
  // pitch was not set
  pitch1 = 55;
  pitch2 = 76;
  env = envelope_data[PITCHfall];
 }
 last_pitch_cmd = wcmdq_tail;
 pitch_length = 0; // total length of spect with this pitch
envelope

 if (pitch2 < 0)
  pitch2 = 0;

 q = wcmdq[wcmdq_tail];
 q[0] = WCMD_PITCH;
 q[1] = 0; // length, fill in later from pitch_length
 q[2] = (intptr_t)env;
 q[3] = (pitch1 << 16) + pitch2;
 WcmdqInc();
}

int PauseLength(int pause, int control)
{
 unsigned int len;

 if (control == 0) {
  if (pause >= 200)
   len = (pause * speed.clause_pause_factor)/256;
  else
   len = (pause * speed.pause_factor)/256;
 } else
  len = (pause * speed.wav_factor)/256;

 if (len < speed.min_pause)
```

```
  len = speed.min_pause; // mS, limit the amount to which pauses
can be shortened
 return len;
}

static void DoPause(int length, int control)
{
 // length in nominal mS
 // control = 1, less shortening at fast speeds

 unsigned int len;
 int srate2;

 if (length == 0)
  len = 0;
 else {
  len = PauseLength(length, control);

  if (len < 90000)
   len = (len * samplerate) / 1000; // convert from mS to number
of samples
  else {
   srate2 = samplerate / 25; // avoid overflow
   len = (len * srate2) / 40;
  }
 }

 EndPitch(1);
 wcmdq[wcmdq_tail][0] = WCMD_PAUSE;
 wcmdq[wcmdq_tail][1] = len;
 WcmdqInc();
 last_frame = NULL;

 if (fmt_amplitude != 0) {
  wcmdq[wcmdq_tail][0] = WCMD_FMT_AMPLITUDE;
  wcmdq[wcmdq_tail][1] = fmt_amplitude = 0;
  WcmdqInc();
```

```
 }
}

extern int seq_len_adjust; // temporary fix to advance the start
point for playing the wav sample

static int DoSample2(int index, int which, int std_length, int
control, int length_mod, int amp)
{
 int length;
 int wav_length;
 int wav_scale;
 int min_length;
 int x;
 int len4;
 intptr_t *q;
 unsigned char *p;

 index = index & 0x7fffff;
 p = &wavefile_data[index];
 wav_scale = p[2];
 wav_length = (p[1] * 256);
 wav_length += p[0]; // length in bytes

 if (wav_length == 0)
  return 0;

 min_length = speed.min_sample_len;

 if (wav_scale == 0)
  min_length *= 2; // 16 bit samples

 if (std_length > 0) {
  std_length = (std_length * samplerate)/1000;
  if (wav_scale == 0)
   std_length *= 2;
```

```
 x = (min_length * std_length)/wav_length;
 if (x > min_length)
  min_length = x;
} else {
 // no length specified, use the length of the stored sound
 std_length = wav_length;
}

if (length_mod > 0)
 std_length = (std_length * length_mod)/256;

length = (std_length * speed.wav_factor)/256;

if (control & pd_DONTLENGTHEN) {
 // this option is used for Stops, with short noise bursts.
 // Don't change their length much.
 if (length > std_length) {
  // don't let length exceed std_length
  length = std_length;
 }
}

if (length < min_length)
 length = min_length;

if (wav_scale == 0) {
 // 16 bit samples
 length /= 2;
 wav_length /= 2;
}

if (amp < 0)
 return length;

len4 = wav_length / 4;

index += 4;
```

```
if (which & 0x100) {
 // mix this with synthesised wave
 last_wcmdq = wcmdq_tail;
 q = wcmdq[wcmdq_tail];
 q[0] = WCMD_WAVE2;
 q[1] = length | (wav_length << 16); // length in samples
 q[2] = (intptr_t)(&wavefile_data[index]);
 q[3] = wav_scale + (amp << 8);
 WcmdqInc();
 return length;
}

if (length > wav_length) {
 x = len4*3;
 length -= x;
} else {
 x = length;
 length = 0;
}

last_wcmdq = wcmdq_tail;
q = wcmdq[wcmdq_tail];
q[0] = WCMD_WAVE;
q[1] = x; // length in samples
q[2] = (intptr_t)(&wavefile_data[index]);
q[3] = wav_scale + (amp << 8);
WcmdqInc();

while (length > len4*3) {
 x = len4;
 if (wav_scale == 0)
  x *= 2;

 last_wcmdq = wcmdq_tail;
 q = wcmdq[wcmdq_tail];
 q[0] = WCMD_WAVE;
```

```
  q[1] = len4*2; // length in samples
  q[2] = (intptr_t)(&wavefile_data[index+x]);
  q[3] = wav_scale + (amp << 8);
  WcmdqInc();

  length -= len4*2;
 }

 if (length > 0) {
  x = wav_length - length;
  if (wav_scale == 0)
   x *= 2;
  last_wcmdq = wcmdq_tail;
  q = wcmdq[wcmdq_tail];
  q[0] = WCMD_WAVE;
  q[1] = length; // length in samples
  q[2] = (intptr_t)(&wavefile_data[index+x]);
  q[3] = wav_scale + (amp << 8);
  WcmdqInc();
 }

 return length;
}

int DoSample3(PHONEME_DATA *phdata, int length_mod, int amp)
{
 int amp2;
 int len;
 EndPitch(1);

 if (amp == -1) {
  // just get the length, don't produce sound
  amp2 = amp;
 } else {
  amp2 = phdata->sound_param[pd_WAV];
  if (amp2 == 0)
   amp2 = 100;
```

```
  amp2 = (amp2 * 32)/100;
 }

 seq_len_adjust = 0;

 if (phdata->sound_addr[pd_WAV] == 0)
  len = 0;
 else
  len = DoSample2(phdata->sound_addr[pd_WAV], 2,
phdata->pd_param[pd_LENGTHMOD]*2, phdata->pd_control, length_mod,
amp2);
 last_frame = NULL;
 return len;
}

static frame_t *AllocFrame()
{
 // Allocate a temporary spectrum frame for the wavegen queue.
Use a pool which is big
 // enough to use a round-robin without checks.
 // Only needed for modifying spectra for blending to consonants

 #define N_FRAME_POOL N_WCMDQ
 static int ix = 0;
 static frame_t frame_pool[N_FRAME_POOL];

 ix++;
 if (ix >= N_FRAME_POOL)
  ix = 0;
 return &frame_pool[ix];
}

static void set_frame_rms(frame_t *fr, int new_rms)
{
 // Each frame includes its RMS amplitude value, so to set a new
 // RMS just adjust the formant amplitudes by the appropriate
ratio
```

```
 int x;
 int h;
 int ix;

 static const short sqrt_tab[200] = {
    0,  64,  90, 110, 128, 143, 156, 169, 181, 192, 202, 212,
221, 230, 239, 247,
  256, 263, 271, 278, 286, 293, 300, 306, 313, 320, 326, 332,
338, 344, 350, 356,
  362, 367, 373, 378, 384, 389, 394, 399, 404, 409, 414, 419,
424, 429, 434, 438,
  443, 448, 452, 457, 461, 465, 470, 474, 478, 483, 487, 491,
495, 499, 503, 507,
  512, 515, 519, 523, 527, 531, 535, 539, 543, 546, 550, 554,
557, 561, 565, 568,
  572, 576, 579, 583, 586, 590, 593, 596, 600, 603, 607, 610,
613, 617, 620, 623,
  627, 630, 633, 636, 640, 643, 646, 649, 652, 655, 658, 662,
665, 668, 671, 674,
  677, 680, 683, 686, 689, 692, 695, 698, 701, 704, 706, 709,
712, 715, 718, 721,
  724, 726, 729, 732, 735, 738, 740, 743, 746, 749, 751, 754,
757, 759, 762, 765,
  768, 770, 773, 775, 778, 781, 783, 786, 789, 791, 794, 796,
799, 801, 804, 807,
  809, 812, 814, 817, 819, 822, 824, 827, 829, 832, 834, 836,
839, 841, 844, 846,
  849, 851, 853, 856, 858, 861, 863, 865, 868, 870, 872, 875,
877, 879, 882, 884,
  886, 889, 891, 893, 896, 898, 900, 902
 };

 if (voice->klattv[0]) {
  if (new_rms == -1)
   fr->klattp[KLATT_AV] = 50;
  return;
```

```
 }

 if (fr->rms == 0) return; // check for divide by zero
 x = (new_rms * 64)/fr->rms;
 if (x >= 200) x = 199;

 x = sqrt_tab[x]; // sqrt(new_rms/fr->rms)*0x200;

 for (ix = 0; ix < 8; ix++) {
  h = fr->fheight[ix] * x;
  fr->fheight[ix] = h/0x200;
 }
}

static void formants_reduce_hf(frame_t *fr, int level)
{
 // change height of peaks 2 to 8, percentage
 int ix;
 int x;

 if (voice->klattv[0])
  return;

 for (ix = 2; ix < 8; ix++) {
  x = fr->fheight[ix] * level;
  fr->fheight[ix] = x/100;
 }
}

static frame_t *CopyFrame(frame_t *frame1, int copy)
{
 // create a copy of the specified frame in temporary buffer

 frame_t *frame2;

 if ((copy == 0) && (frame1->frflags & FRFLAG_COPIED)) {
  // this frame has already been copied in temporary rw memory
```

```
  return frame1;
 }

 frame2 = AllocFrame();
 if (frame2 != NULL) {
  memcpy(frame2, frame1, sizeof(frame_t));
  frame2->length = 0;
  frame2->frflags |= FRFLAG_COPIED;
 }
 return frame2;
}

static frame_t *DuplicateLastFrame(frameref_t *seq, int n_frames,
int length)
{
 frame_t *fr;

 seq[n_frames-1].length = length;
 fr = CopyFrame(seq[n_frames-1].frame, 1);
 seq[n_frames].frame = fr;
 seq[n_frames].length = 0;
 return fr;
}

static void AdjustFormants(frame_t *fr, int target, int min, int
max, int f1_adj, int f3_adj, int hf_reduce, int flags)
{
 int x;

 target = (target * voice->formant_factor)/256;

 x = (target - fr->ffreq[2]) / 2;
 if (x > max) x = max;
 if (x < min) x = min;
 fr->ffreq[2] += x;
 fr->ffreq[3] += f3_adj;
```

```
 if (flags & 0x20)
  f3_adj = -f3_adj; // reverse direction for f4,f5 change
 fr->ffreq[4] += f3_adj;
 fr->ffreq[5] += f3_adj;

 if (f1_adj == 1) {
  x = (235 - fr->ffreq[1]);
  if (x < -100) x = -100;
  if (x > -60) x = -60;
  fr->ffreq[1] += x;
 }
 if (f1_adj == 2) {
  x = (235 - fr->ffreq[1]);
  if (x < -300) x = -300;
  if (x > -150) x = -150;
  fr->ffreq[1] += x;
  fr->ffreq[0] += x;
 }
 if (f1_adj == 3) {
  x = (100 - fr->ffreq[1]);
  if (x < -400) x = -400;
  if (x > -300) x = -400;
  fr->ffreq[1] += x;
  fr->ffreq[0] += x;
 }
 formants_reduce_hf(fr, hf_reduce);
}

static int VowelCloseness(frame_t *fr)
{
 // return a value 0-3 depending on the vowel's f1
 int f1;

 if ((f1 = fr->ffreq[1]) < 300)
  return 3;
 if (f1 < 400)
  return 2;
```

```
 if (f1 < 500)
  return 1;
 return 0;
}

int FormantTransition2(frameref_t *seq, int *n_frames, unsigned
int data1, unsigned int data2, PHONEME_TAB *other_ph, int which)
{
 int ix;
 int formant;
 int next_rms;

 int len;
 int rms;
 int f1;
 int f2;
 int f2_min;
 int f2_max;
 int f3_adj;
 int f3_amp;
 int flags;
 int vcolour;

 #define N_VCOLOUR 2
 // percentage change for each formant in 256ths
 static short vcolouring[N_VCOLOUR][5] = {
  { 243, 272, 256, 256, 256 }, // palatal consonant follows
  { 256, 256, 240, 240, 240 }, // retroflex
 };

 frame_t *fr = NULL;

 if (*n_frames < 2)
  return 0;

 len = (data1 & 0x3f) * 2;
 rms = (data1 >> 6) & 0x3f;
```

```
 flags = (data1 >> 12);

 f2 = (data2 & 0x3f) * 50;
 f2_min = (((data2 >> 6) & 0x1f) - 15) * 50;
 f2_max = (((data2 >> 11) & 0x1f) - 15) * 50;
 f3_adj = (((data2 >> 16) & 0x1f) - 15) * 50;
 f3_amp = ((data2 >> 21) & 0x1f) * 8;
 f1 = ((data2 >> 26) & 0x7);
 vcolour = (data2 >> 29);

 if ((other_ph != NULL) && (other_ph->mnemonic == '?'))
  flags |= 8;

 if (which == 1) {
  // entry to vowel
  fr = CopyFrame(seq[0].frame, 0);
  seq[0].frame = fr;
  seq[0].length = VOWEL_FRONT_LENGTH;
  if (len > 0)
   seq[0].length = len;
  seq[0].frflags |= FRFLAG_LEN_MOD2; // reduce length
modification
  fr->frflags |= FRFLAG_LEN_MOD2;

  next_rms = seq[1].frame->rms;

  if (voice->klattv[0])
   fr->klattp[KLATT_AV] = seq[1].frame->klattp[KLATT_AV] - 4;
  if (f2 != 0) {
   if (rms & 0x20)
    set_frame_rms(fr, (next_rms * (rms & 0x1f))/30);
   AdjustFormants(fr, f2, f2_min, f2_max, f1, f3_adj, f3_amp,
flags);

   if ((rms & 0x20) == 0)
    set_frame_rms(fr, rms*2);
  } else {
```

```
    if (flags & 8)
     set_frame_rms(fr, (next_rms*24)/32);
    else
     set_frame_rms(fr, RMS_START);
   }

   if (flags & 8)
    modn_flags = 0x800 + (VowelCloseness(fr) << 8);
  } else {
   // exit from vowel
   rms = rms*2;
   if ((f2 != 0) || (flags != 0)) {

    if (flags & 8) {
     fr = CopyFrame(seq[*n_frames-1].frame, 0);
     seq[*n_frames-1].frame = fr;
     rms = RMS_GLOTTAL1;

     // degree of glottal-stop effect depends on closeness of
vowel (indicated by f1 freq)
     modn_flags = 0x400 + (VowelCloseness(fr) << 8);
    } else {
     fr = DuplicateLastFrame(seq, (*n_frames)++, len);
     if (len > 36)
      seq_len_adjust += (len - 36);

     if (f2 != 0)
      AdjustFormants(fr, f2, f2_min, f2_max, f1, f3_adj, f3_amp,
flags);
    }

    set_frame_rms(fr, rms);

    if ((vcolour > 0) && (vcolour <= N_VCOLOUR)) {
     for (ix = 0; ix < *n_frames; ix++) {
      fr = CopyFrame(seq[ix].frame, 0);
      seq[ix].frame = fr;
```

```
    for (formant = 1; formant <= 5; formant++) {
     int x;
     x = fr->ffreq[formant] * vcolouring[vcolour-1][formant-1];
     fr->ffreq[formant] = x / 256;
    }
   }
  }
 }
}

if (fr != NULL) {
 if (flags & 4)
  fr->frflags |= FRFLAG_FORMANT_RATE;
 if (flags & 2)
  fr->frflags |= FRFLAG_BREAK; // don't merge with next frame
}

if (flags & 0x40)
 DoPause(20, 0); // add a short pause after the consonant

if (flags & 16)
 return len;
return 0;
}

static void SmoothSpect(void)
{
 // Limit the rate of frequence change of formants, to reduce
chirping

 intptr_t *q;
 frame_t *frame;
 frame_t *frame2;
 frame_t *frame1;
 frame_t *frame_centre;
 int ix;
```

```
 int len;
 int pk;
 bool modified;
 int allowed;
 int diff;

 if (syllable_start == syllable_end)
  return;

 if ((syllable_centre < 0) || (syllable_centre ==
syllable_start)) {
  syllable_start = syllable_end;
  return;
 }

 q = wcmdq[syllable_centre];
 frame_centre = (frame_t *)q[2];

 // backwards
 ix = syllable_centre -1;
 frame = frame2 = frame_centre;
 for (;;) {
  if (ix < 0) ix = N_WCMDQ-1;
  q = wcmdq[ix];

  if (q[0] == WCMD_PAUSE || q[0] == WCMD_WAVE)
   break;

  if (q[0] <= WCMD_SPECT2) {
   len = q[1] & 0xffff;

   frame1 = (frame_t *)q[3];
   if (frame1 == frame) {
    q[3] = (intptr_t)frame2;
    frame1 = frame2;
   } else
    break; // doesn't follow on from previous frame
```

```
   frame = frame2 = (frame_t *)q[2];
   modified = false;

   if (frame->frflags & FRFLAG_BREAK)
    break;

   if (frame->frflags & FRFLAG_FORMANT_RATE)
    len = (len * 12)/10; // allow slightly greater rate of change
for this frame (was 12/10)

   for (pk = 0; pk < 6; pk++) {
    int f1, f2;

    if ((frame->frflags & FRFLAG_BREAK_LF) && (pk < 3))
     continue;

    f1 = frame1->ffreq[pk];
    f2 = frame->ffreq[pk];

    // backwards
    if ((diff = f2 - f1) > 0)
     allowed = f1*2 + f2;
    else
     allowed = f1 + f2*2;

    // the allowed change is specified as percentage (%*10) of
the frequency
    // take "frequency" as 1/3 from the lower freq
    allowed = (allowed * formant_rate[pk])/3000;
    allowed = (allowed * len)/256;

    if (diff > allowed) {
     if (modified == false) {
      frame2 = CopyFrame(frame, 0);
      modified = true;
     }
```

```
    frame2->ffreq[pk] = frame1->ffreq[pk] + allowed;
    q[2] = (intptr_t)frame2;
   } else if (diff < -allowed) {
    if (modified == false) {
     frame2 = CopyFrame(frame, 0);
     modified = true;
    }
    frame2->ffreq[pk] = frame1->ffreq[pk] - allowed;
    q[2] = (intptr_t)frame2;
   }
  }
 }

 if (ix == syllable_start)
  break;
 ix--;
}

// forwards
ix = syllable_centre;

frame = NULL;
for (;;) {
 q = wcmdq[ix];

 if (q[0] == WCMD_PAUSE || q[0] == WCMD_WAVE)
  break;

 if (q[0] <= WCMD_SPECT2) {
  len = q[1] & 0xffff;

  frame1 = (frame_t *)q[2];
  if (frame != NULL) {
   if (frame1 == frame) {
    q[2] = (intptr_t)frame2;
    frame1 = frame2;
   } else
```

```
      break; // doesn't follow on from previous frame
    }

    frame = frame2 = (frame_t *)q[3];
    modified = false;

    if (frame1->frflags & FRFLAG_BREAK)
     break;

    if (frame1->frflags & FRFLAG_FORMANT_RATE)
     len = (len *6)/5; // allow slightly greater rate of change
for this frame

    for (pk = 0; pk < 6; pk++) {
     int f1, f2;
     f1 = frame1->ffreq[pk];
     f2 = frame->ffreq[pk];

     // forwards
     if ((diff = f2 - f1) > 0)
      allowed = f1*2 + f2;
     else
      allowed = f1 + f2*2;
     allowed = (allowed * formant_rate[pk])/3000;
     allowed = (allowed * len)/256;

     if (diff > allowed) {
      if (modified == false) {
       frame2 = CopyFrame(frame, 0);
       modified = true;
      }
      frame2->ffreq[pk] = frame1->ffreq[pk] + allowed;
      q[3] = (intptr_t)frame2;
     } else if (diff < -allowed) {
      if (modified == false) {
       frame2 = CopyFrame(frame, 0);
       modified = true;
```

```
     }
     frame2->ffreq[pk] = frame1->ffreq[pk] - allowed;
     q[3] = (intptr_t)frame2;
    }
   }
  }

  ix++;
  if (ix >= N_WCMDQ) ix = 0;
  if (ix == syllable_end)
   break;
 }

 syllable_start = syllable_end;
}

static void StartSyllable(void)
{
 // start of syllable, if not already started
 if (syllable_end == syllable_start)
  syllable_end = wcmdq_tail;
}

int DoSpect2(PHONEME_TAB *this_ph, int which, FMT_PARAMS
*fmt_params,  PHONEME_LIST *plist, int modulation)
{
 // which:  0 not a vowel, 1  start of vowel,   2 body and end of
vowel
 // length_mod: 256 = 100%
 // modulation: -1 = don't write to wcmdq

 int n_frames;
 frameref_t *frames;
 int frameix;
 frame_t *frame1;
 frame_t *frame2;
 frame_t *fr;
```

```
 int ix;
 intptr_t *q;
 int len;
 int frame_length;
 int length_factor;
 int length_mod;
 int length_sum;
 int length_min;
 int total_len = 0;
 static int wave_flag = 0;
 int wcmd_spect = WCMD_SPECT;
 int frame_lengths[N_SEQ_FRAMES];

 if (fmt_params->fmt_addr == 0)
  return 0;

 length_mod = plist->length;
 if (length_mod == 0) length_mod = 256;

 length_min = (samplerate/70); // greater than one cycle at low
pitch (Hz)
 if (which == 2) {
  if ((translator->langopts.param[LOPT_LONG_VOWEL_THRESHOLD] > 0)
&& ((this_ph->std_length >=
translator->langopts.param[LOPT_LONG_VOWEL_THRESHOLD]) ||
(plist->synthflags & SFLAG_LENGTHEN) || (this_ph->phflags &
phLONG)))
   length_min *= 2; // ensure long vowels are longer
 }

 if (which == 1) {
  // limit the shortening of sonorants before shortened (eg.
unstressed vowels)
  if ((this_ph->type == phLIQUID) || (plist[-1].type == phLIQUID)
|| (plist[-1].type == phNASAL)) {
   if (length_mod < (len =
translator->langopts.param[LOPT_SONORANT_MIN]))
```

```
   length_mod = len;
  }
 }

 modn_flags = 0;
 frames = LookupSpect(this_ph, which, fmt_params, &n_frames,
plist);
 if (frames == NULL)
  return 0; // not found

 if (fmt_params->fmt_amp != fmt_amplitude) {
  // an amplitude adjustment is specified for this sequence
  q = wcmdq[wcmdq_tail];
  q[0] = WCMD_FMT_AMPLITUDE;
  q[1] = fmt_amplitude = fmt_params->fmt_amp;
  WcmdqInc();
 }

 frame1 = frames[0].frame;
 if (voice->klattv[0])
  wcmd_spect = WCMD_KLATT;

 wavefile_ix = fmt_params->wav_addr;

 if (fmt_params->wav_amp == 0)
  wavefile_amp = 32;
 else
  wavefile_amp = (fmt_params->wav_amp * 32)/100;

 if (wavefile_ix == 0) {
  if (wave_flag) {
   // cancel any wavefile that was playing previously
   wcmd_spect = WCMD_SPECT2;
   if (voice->klattv[0])
    wcmd_spect = WCMD_KLATT2;
   wave_flag = 0;
  } else {
```

```
   wcmd_spect = WCMD_SPECT;
   if (voice->klattv[0])
    wcmd_spect = WCMD_KLATT;
  }
 }

 if (last_frame != NULL) {
  if (((last_frame->length < 2) || (last_frame->frflags &
FRFLAG_VOWEL_CENTRE))
      && !(last_frame->frflags & FRFLAG_BREAK)) {
   // last frame of previous sequence was zero-length, replace
with first of this sequence
   wcmdq[last_wcmdq][3] = (intptr_t)frame1;

   if (last_frame->frflags & FRFLAG_BREAK_LF) {
    // but flag indicates keep HF peaks in last segment
    fr = CopyFrame(frame1, 1);
    for (ix = 3; ix < 8; ix++) {
     if (ix < 7)
      fr->ffreq[ix] = last_frame->ffreq[ix];
     fr->fheight[ix] = last_frame->fheight[ix];
    }
    wcmdq[last_wcmdq][3] = (intptr_t)fr;
   }
  }
 }

 if ((this_ph->type == phVOWEL) && (which == 2)) {
  SmoothSpect(); // process previous syllable

  // remember the point in the output queue of the centre of the
vowel
  syllable_centre = wcmdq_tail;
 }

 length_sum = 0;
 for (frameix = 1; frameix < n_frames; frameix++) {
```

```
  length_factor = length_mod;
  if (frames[frameix-1].frflags & FRFLAG_LEN_MOD) // reduce
effect of length mod
   length_factor = (length_mod*(256-speed.lenmod_factor) +
256*speed.lenmod_factor)/256;
  else if (frames[frameix-1].frflags & FRFLAG_LEN_MOD2) // reduce
effect of length mod, used for the start of a vowel
   length_factor = (length_mod*(256-speed.lenmod2_factor) +
256*speed.lenmod2_factor)/256;

  frame_length = frames[frameix-1].length;
  len = (frame_length * samplerate)/1000;
  len = (len * length_factor)/256;
  length_sum += len;
  frame_lengths[frameix] = len;
 }

 if ((length_sum > 0) && (length_sum < length_min)) {
  // lengthen, so that the sequence is greater than one cycle at
low pitch
  for (frameix = 1; frameix < n_frames; frameix++)
   frame_lengths[frameix] = (frame_lengths[frameix] * length_min)
/ length_sum;
 }

 for (frameix = 1; frameix < n_frames; frameix++) {
  frame2 = frames[frameix].frame;

  if ((fmt_params->wav_addr != 0) && ((frame1->frflags &
FRFLAG_DEFER_WAV) == 0)) {
   // there is a wave file to play along with this synthesis
   seq_len_adjust = 0;
   DoSample2(fmt_params->wav_addr, which+0x100, 0,
fmt_params->fmt_control, 0, wavefile_amp);
   wave_flag = 1;
   wavefile_ix = 0;
   fmt_params->wav_addr = 0;
```

```
  }

  if (modulation >= 0) {
   if (frame1->frflags & FRFLAG_MODULATE)
    modulation = 6;
   if ((frameix == n_frames-1) && (modn_flags & 0xf00))
    modulation |= modn_flags; // before or after a glottal stop
  }

  len = frame_lengths[frameix];
  pitch_length += len;
  amp_length += len;

  if (len == 0) {
   last_frame = NULL;
   frame1 = frame2;
  } else {
   last_wcmdq = wcmdq_tail;

   if (modulation >= 0) {
    q = wcmdq[wcmdq_tail];
    q[0] = wcmd_spect;
    q[1] = len + (modulation << 16);
    q[2] = (intptr_t)frame1;
    q[3] = (intptr_t)frame2;

    WcmdqInc();
   }
   last_frame = frame1 = frame2;
   total_len += len;
  }
 }

 if ((which != 1) && (fmt_amplitude != 0)) {
  q = wcmdq[wcmdq_tail];
  q[0] = WCMD_FMT_AMPLITUDE;
  q[1] = fmt_amplitude = 0;
```

```
  WcmdqInc();
 }

 return total_len;
}

void DoMarker(int type, int char_posn, int length, int value)
{
 // This could be used to return an index to the word currently
being spoken
 // Type 1=word, 2=sentence, 3=named marker, 4=play audio, 5=end

 if (WcmdqFree() > 5) {
  wcmdq[wcmdq_tail][0] = WCMD_MARKER + (type << 8);
  wcmdq[wcmdq_tail][1] = (char_posn & 0xffffff) | (length << 24);
  wcmdq[wcmdq_tail][2] = value;
  WcmdqInc();
 }
}

void DoPhonemeMarker(int type, int char_posn, int length, char
*name)
{
 // This could be used to return an index to the word currently
being spoken
 // Type 7=phoneme

 int *p;

 if (WcmdqFree() > 5) {
  wcmdq[wcmdq_tail][0] = WCMD_MARKER + (type << 8);
  wcmdq[wcmdq_tail][1] = (char_posn & 0xffffff) | (length << 24);
  p = (int *)name;
  wcmdq[wcmdq_tail][2] = p[0]; // up to 8 bytes of UTF8
characters
  wcmdq[wcmdq_tail][3] = p[1];
  WcmdqInc();
```

```
 }
}

#if HAVE_SONIC_H
void DoSonicSpeed(int value)
{
 // value, multiplier * 1024
 wcmdq[wcmdq_tail][0] = WCMD_SONIC_SPEED;
 wcmdq[wcmdq_tail][1] = value;
 WcmdqInc();
}
#endif

espeak_ng_STATUS DoVoiceChange(voice_t *v)
{
 // allocate memory for a copy of the voice data, and free it in
wavegenfill()
 voice_t *v2;
 if ((v2 = (voice_t *)malloc(sizeof(voice_t))) == NULL)
  return ENOMEM;
 memcpy(v2, v, sizeof(voice_t));
 wcmdq[wcmdq_tail][0] = WCMD_VOICE;
 wcmdq[wcmdq_tail][2] = (intptr_t)v2;
 WcmdqInc();
 return ENS_OK;
}

void DoEmbedded(int *embix, int sourceix)
{
 // There were embedded commands in the text at this point
 unsigned int word; // bit 7=last command for this word, bits 5,6
sign, bits 0-4 command
 unsigned int value;
 int command;

 do {
  word = embedded_list[*embix];
```

```
  value = word >> 8;
  command = word & 0x7f;

  if (command == 0)
   return; // error

  (*embix)++;

  switch (command & 0x1f)
  {
  case EMBED_S: // speed
   SetEmbedded((command & 0x60) + EMBED_S2, value); // adjusts
embedded_value[EMBED_S2]
   SetSpeed(2);
   break;
  case EMBED_I: // play dynamically loaded wav data (sound icon)
   if ((int)value < n_soundicon_tab) {
    if (soundicon_tab[value].length != 0) {
     DoPause(10, 0); // ensure a break in the speech
     wcmdq[wcmdq_tail][0] = WCMD_WAVE;
     wcmdq[wcmdq_tail][1] = soundicon_tab[value].length;
     wcmdq[wcmdq_tail][2] = (intptr_t)soundicon_tab[value].data +
44; // skip WAV header
     wcmdq[wcmdq_tail][3] = 0x1500; // 16 bit data, amp=21
     WcmdqInc();
    }
   }
   break;
  case EMBED_M: // named marker
   DoMarker(espeakEVENT_MARK, (sourceix & 0x7ff) +
clause_start_char, 0, value);
   break;
  case EMBED_U: // play sound
   DoMarker(espeakEVENT_PLAY, count_characters+1, 0, value); //
always occurs at end of clause
   break;
  default:
```

```
   DoPause(10, 0); // ensure a break in the speech
   wcmdq[wcmdq_tail][0] = WCMD_EMBEDDED;
   wcmdq[wcmdq_tail][1] = command;
   wcmdq[wcmdq_tail][2] = value;
   WcmdqInc();
   break;
  }
 } while ((word & 0x80) == 0);
}

int Generate(PHONEME_LIST *phoneme_list, int *n_ph, bool resume)
{
 static int ix;
 static int embedded_ix;
 static int word_count;
 PHONEME_LIST *prev;
 PHONEME_LIST *next;
 PHONEME_LIST *next2;
 PHONEME_LIST *p;
 bool released;
 int stress;
 int modulation;
 bool  pre_voiced;
 int free_min;
 int value;
 unsigned char *pitch_env = NULL;
 unsigned char *amp_env;
 PHONEME_TAB *ph;
 int use_ipa = 0;
 bool done_phoneme_marker;
 int vowelstart_prev;
 char phoneme_name[16];
 static int sourceix = 0;

 PHONEME_DATA phdata;
 PHONEME_DATA phdata_prev;
 PHONEME_DATA phdata_next;
```

```
PHONEME_DATA phdata_tone;
FMT_PARAMS fmtp;
static WORD_PH_DATA worddata;

if (option_phoneme_events & espeakINITIALIZE_PHONEME_IPA)
 use_ipa = 1;

if (mbrola_name[0] != 0)
 return MbrolaGenerate(phoneme_list, n_ph, resume);

if (resume == false) {
 ix = 1;
 embedded_ix = 0;
 word_count = 0;
 pitch_length = 0;
 amp_length = 0;
 last_frame = NULL;
 last_wcmdq = -1;
 syllable_start = wcmdq_tail;
 syllable_end = wcmdq_tail;
 syllable_centre = -1;
 last_pitch_cmd = -1;
 memset(vowel_transition, 0, sizeof(vowel_transition));
 memset(&worddata, 0, sizeof(worddata));
 DoPause(0, 0); // isolate from the previous clause
}

while ((ix < (*n_ph)) && (ix < N_PHONEME_LIST-2)) {
 p = &phoneme_list[ix];

 if (p->type == phPAUSE)
  free_min = 10;
 else if (p->type != phVOWEL)
  free_min = 15; // we need less Q space for non-vowels, and we
need to generate phonemes after a vowel so that the pitch_length
is filled in
 else
```

```
    free_min = MIN_WCMDQ;

   if (WcmdqFree() <= free_min)
    return 1; // wait

   prev = &phoneme_list[ix-1];
   next = &phoneme_list[ix+1];
   next2 = &phoneme_list[ix+2];

   if (p->synthflags & SFLAG_EMBEDDED)
    DoEmbedded(&embedded_ix, p->sourceix);

   if (p->newword) {
    if (((p->type == phVOWEL) &&
(translator->langopts.param[LOPT_WORD_MERGE] & 1)) ||
        (p->ph->phflags & phNOPAUSE)) {
    } else
     last_frame = NULL;

    sourceix = (p->sourceix & 0x7ff) + clause_start_char;

    if (p->newword & PHLIST_START_OF_SENTENCE)
     DoMarker(espeakEVENT_SENTENCE, sourceix, 0, count_sentences);
// start of sentence

    if (p->newword & PHLIST_START_OF_WORD)
     DoMarker(espeakEVENT_WORD, sourceix, p->sourceix >> 11,
clause_start_word + word_count++); // NOTE, this count doesn't
include multiple-word pronunciations in *_list. eg (of a)
   }

   EndAmplitude();

   if ((p->prepause > 0) && !(p->ph->phflags & phPREVOICE))
    DoPause(p->prepause, 1);

   done_phoneme_marker = false;
```

```
  if (option_phoneme_events && (p->ph->code != phonEND_WORD)) {
   if ((p->type == phVOWEL) && (prev->type == phLIQUID ||
prev->type == phNASAL)) {
    // For vowels following a liquid or nasal, do the phoneme
event after the vowel-start
   } else {
    WritePhMnemonic(phoneme_name, p->ph, p, use_ipa, NULL);
    DoPhonemeMarker(espeakEVENT_PHONEME, sourceix, 0,
phoneme_name);
    done_phoneme_marker = true;
   }
  }

  switch (p->type)
  {
  case phPAUSE:
   DoPause(p->length, 0);
   p->std_length = p->ph->std_length;
   break;
  case phSTOP:
   released = false;
   ph = p->ph;
   if (next->type == phVOWEL)
    released = true;
   else if (!next->newword) {
    if (next->type == phLIQUID) released = true;
   }
   if (released == false)
    p->synthflags |= SFLAG_NEXT_PAUSE;

   if (ph->phflags & phPREVOICE) {
    // a period of voicing before the release
    memset(&fmtp, 0, sizeof(fmtp));
    InterpretPhoneme(NULL, 0x01, p, &phdata, &worddata);
    fmtp.fmt_addr = phdata.sound_addr[pd_FMT];
    fmtp.fmt_amp = phdata.sound_param[pd_FMT];
```

```
    if (last_pitch_cmd < 0) {
     DoAmplitude(next->amp, NULL);
     DoPitch(envelope_data[p->env], next->pitch1, next->pitch2);
    }

    DoSpect2(ph, 0, &fmtp, p, 0);
   }

   InterpretPhoneme(NULL, 0, p, &phdata, &worddata);
   phdata.pd_control |= pd_DONTLENGTHEN;
   DoSample3(&phdata, 0, 0);
   break;
  case phFRICATIVE:
   InterpretPhoneme(NULL, 0, p, &phdata, &worddata);

   if (p->synthflags & SFLAG_LENGTHEN)
    DoSample3(&phdata, p->length, 0); // play it twice for [s:]
etc.
   DoSample3(&phdata, p->length, 0);
   break;
  case phVSTOP:
   ph = p->ph;
   memset(&fmtp, 0, sizeof(fmtp));
   fmtp.fmt_control = pd_DONTLENGTHEN;

   pre_voiced = false;
   if (next->type == phVOWEL) {
    DoAmplitude(p->amp, NULL);
    DoPitch(envelope_data[p->env], p->pitch1, p->pitch2);
    pre_voiced = true;
   } else if ((next->type == phLIQUID) && !next->newword) {
    DoAmplitude(next->amp, NULL);
    DoPitch(envelope_data[next->env], next->pitch1,
next->pitch2);
    pre_voiced = true;
   } else {
    if (last_pitch_cmd < 0) {
```

```
  DoAmplitude(next->amp, NULL);
  DoPitch(envelope_data[p->env], p->pitch1, p->pitch2);
 }
}

if ((prev->type == phVOWEL) || (ph->phflags & phPREVOICE)) {
 // a period of voicing before the release
 InterpretPhoneme(NULL, 0x01, p, &phdata, &worddata);
 fmtp.fmt_addr = phdata.sound_addr[pd_FMT];
 fmtp.fmt_amp = phdata.sound_param[pd_FMT];

 DoSpect2(ph, 0, &fmtp, p, 0);
 if (p->synthflags & SFLAG_LENGTHEN) {
  DoPause(25, 1);
  DoSpect2(ph, 0, &fmtp, p, 0);
 }
} else {
 if (p->synthflags & SFLAG_LENGTHEN)
  DoPause(50, 0);
}

if (pre_voiced) {
 // followed by a vowel, or liquid + vowel
 StartSyllable();
} else
 p->synthflags |= SFLAG_NEXT_PAUSE;
InterpretPhoneme(NULL, 0, p, &phdata, &worddata);
fmtp.fmt_addr = phdata.sound_addr[pd_FMT];
fmtp.fmt_amp = phdata.sound_param[pd_FMT];
fmtp.wav_addr = phdata.sound_addr[pd_ADDWAV];
fmtp.wav_amp = phdata.sound_param[pd_ADDWAV];
DoSpect2(ph, 0, &fmtp, p, 0);

if ((p->newword == 0) && (next2->newword == 0)) {
 if (next->type == phVFRICATIVE)
  DoPause(20, 0);
 if (next->type == phFRICATIVE)
```

```
      DoPause(12, 0);
    }
    break;
   case phVFRICATIVE:
    if (next->type == phVOWEL) {
     DoAmplitude(p->amp, NULL);
     DoPitch(envelope_data[p->env], p->pitch1, p->pitch2);
    } else if (next->type == phLIQUID) {
     DoAmplitude(next->amp, NULL);
     DoPitch(envelope_data[next->env], next->pitch1,
next->pitch2);
    } else {
     if (last_pitch_cmd < 0) {
      DoAmplitude(p->amp, NULL);
      DoPitch(envelope_data[p->env], p->pitch1, p->pitch2);
     }
    }

    if ((next->type == phVOWEL) || ((next->type == phLIQUID) &&
(next->newword == 0))) // ?? test 14.Aug.2007
     StartSyllable();
    else
     p->synthflags |= SFLAG_NEXT_PAUSE;
    InterpretPhoneme(NULL, 0, p, &phdata, &worddata);
    memset(&fmtp, 0, sizeof(fmtp));
    fmtp.std_length = phdata.pd_param[i_SET_LENGTH]*2;
    fmtp.fmt_addr = phdata.sound_addr[pd_FMT];
    fmtp.fmt_amp = phdata.sound_param[pd_FMT];
    fmtp.wav_addr = phdata.sound_addr[pd_ADDWAV];
    fmtp.wav_amp = phdata.sound_param[pd_ADDWAV];

    if (p->synthflags & SFLAG_LENGTHEN)
     DoSpect2(p->ph, 0, &fmtp, p, 0);
    DoSpect2(p->ph, 0, &fmtp, p, 0);
    break;
   case phNASAL:
    memset(&fmtp, 0, sizeof(fmtp));
```

```
  if (!(p->synthflags & SFLAG_SEQCONTINUE)) {
   DoAmplitude(p->amp, NULL);
   DoPitch(envelope_data[p->env], p->pitch1, p->pitch2);
  }

  if (prev->type == phNASAL)
   last_frame = NULL;

  InterpretPhoneme(NULL, 0, p, &phdata, &worddata);
  fmtp.std_length = phdata.pd_param[i_SET_LENGTH]*2;
  fmtp.fmt_addr = phdata.sound_addr[pd_FMT];
  fmtp.fmt_amp = phdata.sound_param[pd_FMT];

  if (next->type == phVOWEL) {
   StartSyllable();
   DoSpect2(p->ph, 0, &fmtp, p, 0);
  } else if (prev->type == phVOWEL && (p->synthflags &
SFLAG_SEQCONTINUE))
   DoSpect2(p->ph, 0, &fmtp, p, 0);
  else {
   last_frame = NULL; // only for nasal ?
   DoSpect2(p->ph, 0, &fmtp, p, 0);
   last_frame = NULL;
  }

  break;
 case phLIQUID:
  memset(&fmtp, 0, sizeof(fmtp));
  modulation = 0;
  if (p->ph->phflags & phTRILL)
   modulation = 5;

  if (!(p->synthflags & SFLAG_SEQCONTINUE)) {
   DoAmplitude(p->amp, NULL);
   DoPitch(envelope_data[p->env], p->pitch1, p->pitch2);
  }
```

```
   if (prev->type == phNASAL)
    last_frame = NULL;

   if (next->type == phVOWEL)
    StartSyllable();
   InterpretPhoneme(NULL, 0, p, &phdata, &worddata);

   if ((value = (phdata.pd_param[i_PAUSE_BEFORE] - p->prepause))
> 0)
    DoPause(value, 1);
   fmtp.std_length = phdata.pd_param[i_SET_LENGTH]*2;
   fmtp.fmt_addr = phdata.sound_addr[pd_FMT];
   fmtp.fmt_amp = phdata.sound_param[pd_FMT];
   fmtp.wav_addr = phdata.sound_addr[pd_ADDWAV];
   fmtp.wav_amp = phdata.sound_param[pd_ADDWAV];
   DoSpect2(p->ph, 0, &fmtp, p, modulation);
   break;
  case phVOWEL:
   ph = p->ph;
   stress = p->stresslevel & 0xf;

   memset(&fmtp, 0, sizeof(fmtp));

   InterpretPhoneme(NULL, 0, p, &phdata, &worddata);
   fmtp.std_length = phdata.pd_param[i_SET_LENGTH] * 2;
   vowelstart_prev = 0;

   if (((fmtp.fmt_addr = phdata.sound_addr[pd_VWLSTART]) != 0) &&
((phdata.pd_control & pd_FORNEXTPH) == 0)) {
    // a vowel start has been specified by the Vowel program
    fmtp.fmt_length = phdata.sound_param[pd_VWLSTART];
   } else if (prev->type != phPAUSE) {
    // check the previous phoneme
    InterpretPhoneme(NULL, 0, prev, &phdata_prev, NULL);
    if (((fmtp.fmt_addr = phdata_prev.sound_addr[pd_VWLSTART]) !=
0) && (phdata_prev.pd_control & pd_FORNEXTPH)) {
     // a vowel start has been specified by the previous phoneme
```

```
    vowelstart_prev = 1;
    fmtp.fmt2_lenadj = phdata_prev.sound_param[pd_VWLSTART];
   }
   fmtp.transition0 = phdata_prev.vowel_transition[0];
   fmtp.transition1 = phdata_prev.vowel_transition[1];
  }

  if (fmtp.fmt_addr == 0) {
   // use the default start for this vowel
   fmtp.use_vowelin = 1;
   fmtp.fmt_control = 1;
   fmtp.fmt_addr = phdata.sound_addr[pd_FMT];
  }

  fmtp.fmt_amp = phdata.sound_param[pd_FMT];

  pitch_env = envelope_data[p->env];
  amp_env = NULL;
  if (p->tone_ph != 0) {
   InterpretPhoneme2(p->tone_ph, &phdata_tone);
   pitch_env = GetEnvelope(phdata_tone.pitch_env);
   if (phdata_tone.amp_env > 0)
    amp_env = GetEnvelope(phdata_tone.amp_env);
  }

  StartSyllable();

  modulation = 2;
  if (stress <= 1)
   modulation = 1; // 16ths
  else if (stress >= 7)
   modulation = 3;

  if (prev->type == phVSTOP || prev->type == phVFRICATIVE) {
   DoAmplitude(p->amp, amp_env);
   DoPitch(pitch_env, p->pitch1, p->pitch2); // don't use
prevocalic rising tone
```

```
    DoSpect2(ph, 1, &fmtp, p, modulation);
   } else if (prev->type == phLIQUID || prev->type == phNASAL) {
    DoAmplitude(p->amp, amp_env);
    DoSpect2(ph, 1, &fmtp, p, modulation); // continue with pre-
vocalic rising tone
    DoPitch(pitch_env, p->pitch1, p->pitch2);
   } else if (vowelstart_prev) {
    // VowelStart from the previous phoneme, but not phLIQUID or
phNASAL
    DoPitch(envelope_data[PITCHrise], p->pitch2 - 15, p->pitch2);
    DoAmplitude(p->amp-1, amp_env);
    DoSpect2(ph, 1, &fmtp, p, modulation); // continue with pre-
vocalic rising tone
    DoPitch(pitch_env, p->pitch1, p->pitch2);
   } else {
    if (!(p->synthflags & SFLAG_SEQCONTINUE)) {
     DoAmplitude(p->amp, amp_env);
     DoPitch(pitch_env, p->pitch1, p->pitch2);
    }

    DoSpect2(ph, 1, &fmtp, p, modulation);
   }

   if ((option_phoneme_events) && (done_phoneme_marker == false))
{
    WritePhMnemonic(phoneme_name, p->ph, p, use_ipa, NULL);
    DoPhonemeMarker(espeakEVENT_PHONEME, sourceix, 0,
phoneme_name);
   }

   fmtp.fmt_addr = phdata.sound_addr[pd_FMT];
   fmtp.fmt_amp = phdata.sound_param[pd_FMT];
   fmtp.transition0 = 0;
   fmtp.transition1 = 0;

   if ((fmtp.fmt2_addr = phdata.sound_addr[pd_VWLEND]) != 0)
    fmtp.fmt2_lenadj = phdata.sound_param[pd_VWLEND];
```

```
   else if (next->type != phPAUSE) {
    fmtp.fmt2_lenadj = 0;
    InterpretPhoneme(NULL, 0, next, &phdata_next, NULL);

    fmtp.use_vowelin = 1;
    fmtp.transition0 = phdata_next.vowel_transition[2]; // always
do vowel_transition, even if ph_VWLEND ??  consider [N]
    fmtp.transition1 = phdata_next.vowel_transition[3];

    if ((fmtp.fmt2_addr = phdata_next.sound_addr[pd_VWLEND]) !=
0)
     fmtp.fmt2_lenadj = phdata_next.sound_param[pd_VWLEND];
   }

   DoSpect2(ph, 2, &fmtp, p, modulation);
   break;
  }
  ix++;
 }
 EndPitch(1);
 if (*n_ph > 0) {
  DoMarker(espeakEVENT_END, count_characters, 0,
count_sentences); // end of clause
  *n_ph = 0;
 }

 return 0; // finished the phoneme list
}

int SpeakNextClause(int control)
{
 // Speak text from memory (text_in)
 // control 0: start
 //    text_in is set

 // The other calls have text_in = NULL
 // control 1: speak next text
```

```
//        2: stop

int clause_tone;
char *voice_change;
const char *phon_out;

if (control == 2) {
 // stop speaking
 n_phoneme_list = 0;
 WcmdqStop();

 return 0;
}

if (text_decoder_eof(p_decoder)) {
 skipping_text = false;
 return 0;
}

if (current_phoneme_table != voice->phoneme_tab_ix)
 SelectPhonemeTable(voice->phoneme_tab_ix);

// read the next clause from the input text file, translate it,
and generate
// entries in the wavegen command queue
TranslateClause(translator, &clause_tone, &voice_change);

CalcPitches(translator, clause_tone);
CalcLengths(translator);

if ((option_phonemes & 0xf) || (phoneme_callback != NULL)) {
 phon_out = GetTranslatedPhonemeString(option_phonemes);
 if (option_phonemes & 0xf)
  fprintf(f_trans, "%s\n", phon_out);
 if (phoneme_callback != NULL)
  phoneme_callback(phon_out);
}
```

```
 if (skipping_text) {
  n_phoneme_list = 0;
  return 1;
 }

 Generate(phoneme_list, &n_phoneme_list, 0);

 if (voice_change != NULL) {
  // voice change at the end of the clause (i.e. clause was
terminated by a voice change)
  new_voice = LoadVoiceVariant(voice_change, 0); // add a Voice
instruction to wavegen at the end of the clause
 }

 if (new_voice) {
  // finished the current clause, now change the voice if there
was an embedded
  // change voice command at the end of it (i.e. clause was
broken at the change voice command)
  DoVoiceChange(voice);
  new_voice = NULL;
 }

 return 1;
}
```

Chapter 57

./src/libespeak-ng/mnemonics.c

```
#include "config.h"

#include <string.h>

#include <espeak-ng/espeak_ng.h>

#include "speech.h"

int LookupMnem(MNEM_TAB *table, const char *string)
{
 while (table->mnem != NULL) {
  if (string && strcmp(string, table->mnem) == 0)
   return table->value;
  table++;
 }
 return table->value;
}

const char *LookupMnemName(MNEM_TAB *table, const int value)
{
 while (table->mnem != NULL) {
  if (table->value == value)
```

```
    return table->mnem;
   table++;
  }
 return ""; // not found
}
```

Chapter 58

./src/libespeak-ng/compilembrola.c

```
#include "config.h"

#include <errno.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>

#include "mbrola.h"

#include "error.h"
#include "phoneme.h"
#include "speech.h"
#include "voice.h"
#include "synthesize.h"

static const char *basename(const char *filename)
```

```
{
 const char *current = filename + strlen(filename);
 while (current != filename && !(*current == '/' || *current ==
'\\'))
  --current;
 return current == filename ? current : current + 1;
}

static unsigned int StringToWord(const char *string)
{
 // Pack 4 characters into a word
 int ix;
 unsigned char c;
 unsigned int word;

 if (string == NULL)
  return 0;

 word = 0;
 for (ix = 0; ix < 4; ix++) {
  if (string[ix] == 0) break;
  c = string[ix];
  word |= (c << (ix*8));
 }
 return word;
}

#pragma GCC visibility push(default)
espeak_ng_STATUS espeak_ng_CompileMbrolaVoice(const char
*filepath, FILE *log, espeak_ng_ERROR_CONTEXT *context)
{
 if (!log) log = stderr;

 char *p;
 FILE *f_in;
 FILE *f_out;
 int percent;
```

```
 int n;
 int *pw;
 int *pw_end;
 int count = 0;
 int control;
 char phoneme[40];
 char phoneme2[40];
 char name1[40];
 char name2[40];
 char mbrola_voice[40];
 char buf[sizeof(path_home)+30];
 int mbrola_ctrl = 20; // volume in 1/16 ths
 MBROLA_TAB data[N_PHONEME_TAB];

 if ((f_in = fopen(filepath, "r")) == NULL)
  return create_file_error_context(context, errno, filepath);

 while (fgets(buf, sizeof(phoneme), f_in) != NULL) {
  buf[sizeof(phoneme)-1] = 0;

  if ((p = strstr(buf, "//")) != NULL)
   *p = 0; // truncate line at comment

  if (memcmp(buf, "volume", 6) == 0) {
   mbrola_ctrl = atoi(&buf[6]);
   continue;
  }

  n = sscanf(buf, "%d %s %s %d %s %s", &control, phoneme,
phoneme2, &percent, name1, name2);
  if (n >= 5) {
   data[count].name = StringToWord(phoneme);
   if (strcmp(phoneme2, "NULL") == 0)
    data[count].next_phoneme = 0;
   else if (strcmp(phoneme2, "VWL") == 0)
    data[count].next_phoneme = 2;
   else
```

```
    data[count].next_phoneme = StringToWord(phoneme2);
   data[count].mbr_name = 0;
   data[count].mbr_name2 = 0;
   data[count].percent = percent;
   data[count].control = control;
   if (strcmp(name1, "NULL") != 0)
    data[count].mbr_name = StringToWord(name1);
   if (n == 6)
    data[count].mbr_name2 = StringToWord(name2);

   count++;
  }
 }
 fclose(f_in);

 strcpy(mbrola_voice, basename(filepath));
 sprintf(buf, "%s/mbrola_ph/%s_phtrans", path_home,
mbrola_voice);
 if ((f_out = fopen(buf, "wb")) == NULL)
  return create_file_error_context(context, errno, buf);

 memset(&data[count], 0, sizeof(data[count]));
 data[count].name = 0; // list terminator
 Write4Bytes(f_out, mbrola_ctrl);

 pw_end = (int *)(&data[count+1]);
 for (pw = (int *)data; pw < pw_end; pw++)
  Write4Bytes(f_out, *pw);
 fclose(f_out);
 fprintf(log, "Mbrola translation file: %s -- %d phonemes\n",
buf, count);
 return ENS_OK;
}
#pragma GCC visibility pop
```

Chapter 59

./src/libespeak-ng/readclause.c

```
#include "config.h"

#include <ctype.h>
#include <errno.h>
#include <locale.h>
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <wchar.h>
#include <wctype.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>
#include <ucd/ucd.h>

#include "dictionary.h"
#include "readclause.h"
#include "synthdata.h"
```

```
#include "error.h"
#include "speech.h"
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"
#include "ssml.h"

#define N_XML_BUF   500

static const char *xmlbase = ""; // base URL from <speak>

static int namedata_ix = 0;
static int n_namedata = 0;
char *namedata = NULL;

static int ungot_char2 = 0;
espeak_ng_TEXT_DECODER *p_decoder = NULL;
static int ungot_char;
static const char *ungot_word = NULL;

static bool ignore_text = false; // set during <sub> ... </sub>
to ignore text which has been replaced by an alias
static bool audio_text = false; // set during <audio> ...
</audio>
static bool clear_skipping_text = false; // next clause should
clear the skipping_text flag
int count_characters = 0;
static int sayas_mode;
static int sayas_start;
static int ssml_ignore_l_angle = 0;

#define N_SSML_STACK  20
static int n_ssml_stack;
static SSML_STACK ssml_stack[N_SSML_STACK];
```

```
static espeak_VOICE base_voice;
static char base_voice_variant_name[40] = { 0 };
static char current_voice_id[40] = { 0 };

static int n_param_stack;
PARAM_STACK param_stack[N_PARAM_STACK];

static int speech_parameters[N_SPEECH_PARAM]; // current values,
from param_stack
int saved_parameters[N_SPEECH_PARAM]; // Parameters saved on
synthesis start

#define ESPEAKNG_CLAUSE_TYPE_PROPERTY_MASK 0xFFF0000000000000ull

int clause_type_from_codepoint(uint32_t c)
{
 ucd_category cat = ucd_lookup_category(c);
 ucd_property props = ucd_properties(c, cat);

 switch (props & ESPEAKNG_CLAUSE_TYPE_PROPERTY_MASK)
 {
 case ESPEAKNG_PROPERTY_FULL_STOP:
  return CLAUSE_PERIOD;
 case ESPEAKNG_PROPERTY_FULL_STOP |
ESPEAKNG_PROPERTY_OPTIONAL_SPACE_AFTER:
  return CLAUSE_PERIOD | CLAUSE_OPTIONAL_SPACE_AFTER;
 case ESPEAKNG_PROPERTY_QUESTION_MARK:
  return CLAUSE_QUESTION;
 case ESPEAKNG_PROPERTY_QUESTION_MARK |
ESPEAKNG_PROPERTY_OPTIONAL_SPACE_AFTER:
  return CLAUSE_QUESTION | CLAUSE_OPTIONAL_SPACE_AFTER;
 case ESPEAKNG_PROPERTY_QUESTION_MARK |
ESPEAKNG_PROPERTY_PUNCTUATION_IN_WORD:
  return CLAUSE_QUESTION | CLAUSE_PUNCTUATION_IN_WORD;
 case ESPEAKNG_PROPERTY_EXCLAMATION_MARK:
  return CLAUSE_EXCLAMATION;
 case ESPEAKNG_PROPERTY_EXCLAMATION_MARK |
```

```
ESPEAKNG_PROPERTY_OPTIONAL_SPACE_AFTER:
   return CLAUSE_EXCLAMATION | CLAUSE_OPTIONAL_SPACE_AFTER;
  case ESPEAKNG_PROPERTY_EXCLAMATION_MARK |
ESPEAKNG_PROPERTY_PUNCTUATION_IN_WORD:
   return CLAUSE_EXCLAMATION | CLAUSE_PUNCTUATION_IN_WORD;
  case ESPEAKNG_PROPERTY_COMMA:
   return CLAUSE_COMMA;
  case ESPEAKNG_PROPERTY_COMMA |
ESPEAKNG_PROPERTY_OPTIONAL_SPACE_AFTER:
   return CLAUSE_COMMA | CLAUSE_OPTIONAL_SPACE_AFTER;
  case ESPEAKNG_PROPERTY_COLON:
   return CLAUSE_COLON;
  case ESPEAKNG_PROPERTY_COLON |
ESPEAKNG_PROPERTY_OPTIONAL_SPACE_AFTER:
   return CLAUSE_COLON | CLAUSE_OPTIONAL_SPACE_AFTER;
  case ESPEAKNG_PROPERTY_SEMI_COLON:
  case ESPEAKNG_PROPERTY_EXTENDED_DASH:
   return CLAUSE_SEMICOLON;
  case ESPEAKNG_PROPERTY_SEMI_COLON |
ESPEAKNG_PROPERTY_OPTIONAL_SPACE_AFTER:
  case ESPEAKNG_PROPERTY_QUESTION_MARK |
ESPEAKNG_PROPERTY_OPTIONAL_SPACE_AFTER |
ESPEAKNG_PROPERTY_INVERTED_TERMINAL_PUNCTUATION:
  case ESPEAKNG_PROPERTY_EXCLAMATION_MARK |
ESPEAKNG_PROPERTY_OPTIONAL_SPACE_AFTER |
ESPEAKNG_PROPERTY_INVERTED_TERMINAL_PUNCTUATION:
   return CLAUSE_SEMICOLON | CLAUSE_OPTIONAL_SPACE_AFTER;
  case ESPEAKNG_PROPERTY_ELLIPSIS:
   return CLAUSE_SEMICOLON | CLAUSE_SPEAK_PUNCTUATION_NAME |
CLAUSE_OPTIONAL_SPACE_AFTER;
  case ESPEAKNG_PROPERTY_PARAGRAPH_SEPARATOR:
   return CLAUSE_PARAGRAPH;
  }

 return CLAUSE_NONE;
}
```

```
int is_str_totally_null(const char* str, int size) {
 // Tests if all bytes of str are null up to size
 // This should never be reimplemented with integers, because
 // this function has to work with unaligned char*
 // (casting to int when unaligned may result in ungaranteed
behaviors)
 return (*str == 0 && memcmp(str, str+1, size-1) == 0);
}

int towlower2(unsigned int c, Translator *translator)
{
 // check for non-standard upper to lower case conversions
 if (c == 'I' && translator->langopts.dotless_i)
  return 0x131; // I -> ı

 return ucd_tolower(c);
}

static int IsRomanU(unsigned int c)
{
 if ((c == 'I') || (c == 'V') || (c == 'X') || (c == 'L'))
  return 1;
 return 0;
}

int Eof(void)
{
 if (ungot_char != 0)
  return 0;

 return text_decoder_eof(p_decoder);
}

static int GetC(void)
{
 int c1;
```

```
 if ((c1 = ungot_char) != 0) {
  ungot_char = 0;
  return c1;
 }

 count_characters++;
 return text_decoder_getc(p_decoder);
}

static void UngetC(int c)
{
 ungot_char = c;
}

const char *WordToString2(unsigned int word)
{
 // Convert a language mnemonic word into a string
 int ix;
 static char buf[5];
 char *p;

 p = buf;
 for (ix = 3; ix >= 0; ix--) {
  if ((*p = word >> (ix*8)) != 0)
   p++;
 }

 return buf;
}

static const char *LookupSpecial(Translator *tr, const char
*string, char *text_out)
{
 unsigned int flags[2];
 char phonemes[55];
 char phonemes2[55];
 char *string1 = (char *)string;
```

```
 flags[0] = flags[1] = 0;
 if (LookupDictList(tr, &string1, phonemes, flags, 0, NULL)) {
  SetWordStress(tr, phonemes, flags, -1, 0);
  DecodePhonemes(phonemes, phonemes2);
  sprintf(text_out, "[\002%s]]", phonemes2);
  return text_out;
 }
 return NULL;
}

static const char *LookupCharName(Translator *tr, int c, int
only)
{
 // Find the phoneme string (in ascii) to speak the name of
character c
 // Used for punctuation characters and symbols

 int ix;
 unsigned int flags[2];
 char single_letter[24];
 char phonemes[60];
 char phonemes2[60];
 const char *lang_name = NULL;
 char *string;
 static char buf[60];

 buf[0] = 0;
 flags[0] = 0;
 flags[1] = 0;
 single_letter[0] = 0;
 single_letter[1] = '_';
 ix = utf8_out(c, &single_letter[2]);
 single_letter[2+ix] = 0;

 if (only) {
  string = &single_letter[2];
```

```
  LookupDictList(tr, &string, phonemes, flags, 0, NULL);
 } else {
  string = &single_letter[1];
  if (LookupDictList(tr, &string, phonemes, flags, 0, NULL) == 0)
{
   // try _* then *
   string = &single_letter[2];
   if (LookupDictList(tr, &string, phonemes, flags, 0, NULL) ==
0) {
    // now try the rules
    single_letter[1] = ' ';
    TranslateRules(tr, &single_letter[2], phonemes,
sizeof(phonemes), NULL, 0, NULL);
   }
  }
 }

 if ((only == 0) && ((phonemes[0] == 0) || (phonemes[0] ==
phonSWITCH)) && (tr->translator_name != L('e', 'n'))) {
  // not found, try English
  SetTranslator2("en");
  string = &single_letter[1];
  single_letter[1] = '_';
  if (LookupDictList(translator2, &string, phonemes, flags, 0,
NULL) == 0) {
   string = &single_letter[2];
   LookupDictList(translator2, &string, phonemes, flags, 0,
NULL);
  }
  if (phonemes[0])
   lang_name = "en";
  else
   SelectPhonemeTable(voice->phoneme_tab_ix); // revert to
original phoneme table
 }

 if (phonemes[0]) {
```

```
  if (lang_name) {
   SetWordStress(translator2, phonemes, flags, -1, 0);
   DecodePhonemes(phonemes, phonemes2);
   sprintf(buf, "[\002_^_%s %s _^_%s]]", "en", phonemes2,
WordToString2(tr->translator_name));
   SelectPhonemeTable(voice->phoneme_tab_ix); // revert to
original phoneme table
  } else {
   SetWordStress(tr, phonemes, flags, -1, 0);
   DecodePhonemes(phonemes, phonemes2);
   sprintf(buf, "[\002%s]] ", phonemes2);
  }
 } else if (only == 0)
  strcpy(buf, "[\002(X1)(X1)(X1)]]");

 return buf;
}

int Read4Bytes(FILE *f)
{
 // Read 4 bytes (least significant first) into a word
 int ix;
 unsigned char c;
 int acc = 0;

 for (ix = 0; ix < 4; ix++) {
  c = fgetc(f) & 0xff;
  acc += (c << (ix*8));
 }
 return acc;
}

static espeak_ng_STATUS LoadSoundFile(const char *fname, int
index, espeak_ng_ERROR_CONTEXT *context)
{
 FILE *f;
 char *p;
```

```
int *ip;
int length;
char fname_temp[100];
char fname2[sizeof(path_home)+13+40];

if (fname == NULL) {
 // filename is already in the table
 fname = soundicon_tab[index].filename;
}

if (fname == NULL)
 return EINVAL;

if (fname[0] != '/') {
 // a relative path, look in espeak-ng-data/soundicons
 sprintf(fname2, "%s%csoundicons%c%s", path_home, PATHSEP,
PATHSEP, fname);
 fname = fname2;
}

f = NULL;
if ((f = fopen(fname, "rb")) != NULL) {
 int ix;
 int fd_temp;
 int header[3];
 char command[sizeof(fname2)+sizeof(fname2)+40];

 if (fseek(f, 20, SEEK_SET) == -1) {
  int error = errno;
  fclose(f);
  return create_file_error_context(context, error, fname);
 }

 for (ix = 0; ix < 3; ix++)
  header[ix] = Read4Bytes(f);

 // if the sound file is not mono, 16 bit signed, at the correct
```

```
sample rate, then convert it
  if ((header[0] != 0x10001) || (header[1] != samplerate) ||
(header[2] != samplerate*2)) {
   fclose(f);
   f = NULL;

#ifdef HAVE_MKSTEMP
   strcpy(fname_temp, "/tmp/espeakXXXXXX");
   if ((fd_temp = mkstemp(fname_temp)) >= 0)
    close(fd_temp);
#else
   strcpy(fname_temp, tmpnam(NULL));
#endif

   sprintf(command, "sox \"%s\" -r %d -c1 -t wav %s\n", fname,
samplerate, fname_temp);
   if (system(command) == 0)
    fname = fname_temp;
  }
 }

 if (f == NULL) {
  f = fopen(fname, "rb");
  if (f == NULL)
   return create_file_error_context(context, errno, fname);
 }

 length = GetFileLength(fname);
 if (length < 0) { // length == -errno
  fclose(f);
  return create_file_error_context(context, -length, fname);
 }
 if (fseek(f, 0, SEEK_SET) == -1) {
  int error = errno;
  fclose(f);
  return create_file_error_context(context, error, fname);
 }
```

```
 if ((p = (char *)realloc(soundicon_tab[index].data, length)) ==
NULL) {
  fclose(f);
  return ENOMEM;
 }
 if (fread(p, 1, length, f) != length) {
  int error = errno;
  fclose(f);
  remove(fname_temp);
  free(p);
  return create_file_error_context(context, error, fname);
 }
 fclose(f);
 remove(fname_temp);

 ip = (int *)(&p[40]);
 soundicon_tab[index].length = (*ip) / 2; // length in samples
 soundicon_tab[index].data = p;
 return ENS_OK;
}

static int LookupSoundicon(int c)
{
 // Find the sound icon number for a punctuation chatacter
 int ix;

 for (ix = N_SOUNDICON_SLOTS; ix < n_soundicon_tab; ix++) {
  if (soundicon_tab[ix].name == c) {
   if (soundicon_tab[ix].length == 0) {
    if (LoadSoundFile(NULL, ix, NULL) != ENS_OK)
     return -1; // sound file is not available
   }
   return ix;
  }
 }
 return -1;
}
```

```
int LoadSoundFile2(const char *fname)
{
 // Load a sound file into one of the reserved slots in the sound
icon table
 // (if it'snot already loaded)

 int ix;
 static int slot = -1;

 for (ix = 0; ix < n_soundicon_tab; ix++) {
  if (((soundicon_tab[ix].filename != NULL) && strcmp(fname,
soundicon_tab[ix].filename) == 0))
   return ix; // already loaded
 }

 // load the file into the next slot
 slot++;
 if (slot >= N_SOUNDICON_SLOTS)
  slot = 0;

 if (LoadSoundFile(fname, slot, NULL) != ENS_OK)
  return -1;

 soundicon_tab[slot].filename = (char
*)realloc(soundicon_tab[ix].filename, strlen(fname)+1);
 strcpy(soundicon_tab[slot].filename, fname);
 return slot;
}

static int AnnouncePunctuation(Translator *tr, int c1, int
*c2_ptr, char *output, int *bufix, int end_clause)
{
 // announce punctuation names
 // c1:  the punctuation character
 // c2:  the following character
```

```
 int punct_count;
 const char *punctname = NULL;
 int soundicon;
 int attributes;
 int short_pause;
 int c2;
 int len;
 int bufix1;
 char buf[200];
 char buf2[80];
 char ph_buf[30];

 c2 = *c2_ptr;
 buf[0] = 0;

 if ((soundicon = LookupSoundicon(c1)) >= 0) {
  // add an embedded command to play the soundicon
  sprintf(buf, "\001%dI ", soundicon);
  UngetC(c2);
 } else {
  if ((c1 == '.') && (end_clause) && (c2 != '.')) {
   if (LookupSpecial(tr, "_.p", ph_buf))
    punctname = ph_buf; // use word for 'period' instead of 'dot'
  }
  if (punctname == NULL)
   punctname = LookupCharName(tr, c1, 0);

  if (punctname == NULL)
   return -1;

  if ((*bufix == 0) || (end_clause == 0) ||
(tr->langopts.param[LOPT_ANNOUNCE_PUNCT] & 2)) {
   punct_count = 1;
   while ((c2 == c1) && (c1 != '<')) { // don't eat extra '<', it
can miss XML tags
    punct_count++;
    c2 = GetC();
```

```
   }
   *c2_ptr = c2;
   if (end_clause)
    UngetC(c2);

   if (punct_count == 1)
    sprintf(buf, " %s", punctname); // we need the space before
punctname, to ensure it doesn't merge with the previous word
(eg.  "2.-a")
   else if (punct_count < 4) {
    buf[0] = 0;
    if (embedded_value[EMBED_S] < 300)
     sprintf(buf, "\001+10S"); // Speak punctuation name faster,
unless we are already speaking fast.  It would upset Sonic
SpeedUp

    while (punct_count-- > 0) {
     sprintf(buf2, " %s", punctname);
     strcat(buf, buf2);
    }

    if (embedded_value[EMBED_S] < 300) {
     sprintf(buf2, " \001-10S");
     strcat(buf, buf2);
    }
   } else
    sprintf(buf, " %s %d %s",
            punctname, punct_count, punctname);
  } else {
   // end the clause now and pick up the punctuation next time
   UngetC(c2);
   if (option_ssml) {
    if ((c1 == '<') || (c1 == '&'))
     ssml_ignore_l_angle = c1; // this was &lt; which was
converted to <, don't pick it up again as <
   }
   ungot_char2 = c1;
```

```
   buf[0] = ' ';
   buf[1] = 0;
  }
 }

 bufix1 = *bufix;
 len = strlen(buf);
 strcpy(&output[*bufix], buf);

 if (end_clause == 0)
  return -1;

 if (c1 == '-')
  return CLAUSE_NONE; // no pause

 attributes = clause_type_from_codepoint(c1);

 short_pause = CLAUSE_SHORTFALL;
 if ((attributes & CLAUSE_INTONATION_TYPE) == 0x1000)
  short_pause = CLAUSE_SHORTCOMMA;

 if ((bufix1 > 0) && !(tr->langopts.param[LOPT_ANNOUNCE_PUNCT] &
2)) {
  if ((attributes & ~CLAUSE_OPTIONAL_SPACE_AFTER) ==
CLAUSE_SEMICOLON)
   return CLAUSE_SHORTFALL;
  return short_pause;
 }

 if (attributes & CLAUSE_TYPE_SENTENCE)
  return attributes;

 return short_pause;
}

int AddNameData(const char *name, int wide)
{
```

```
 // Add the name to the namedata and return its position
 // (Used by the Windows SAPI wrapper)

 int ix;
 int len;
 void *vp;

 if (wide) {
  len = (wcslen((const wchar_t *)name)+1)*sizeof(wchar_t);
  n_namedata = (n_namedata + sizeof(wchar_t) - 1) %
sizeof(wchar_t);  // round to wchar_t boundary
 } else
  len = strlen(name)+1;

 if (namedata_ix+len >= n_namedata) {
  // allocate more space for marker names
  if ((vp = realloc(namedata, namedata_ix+len + 1000)) == NULL)
   return -1;  // failed to allocate, original data is unchanged
but ignore this new name
  // !!! Bug?? If the allocated data shifts position, then
pointers given to user application will be invalid

  namedata = (char *)vp;
  n_namedata = namedata_ix+len + 1000;
 }
 memcpy(&namedata[ix = namedata_ix], name, len);
 namedata_ix += len;
 return ix;
}

void SetVoiceStack(espeak_VOICE *v, const char *variant_name)
{
 SSML_STACK *sp;
 sp = &ssml_stack[0];

 if (v == NULL) {
  memset(sp, 0, sizeof(ssml_stack[0]));
```

```
  return;
 }
 if (v->languages != NULL)
  strcpy(sp->language, v->languages);
 if (v->name != NULL)
  strncpy0(sp->voice_name, v->name, sizeof(sp->voice_name));
 sp->voice_variant_number = v->variant;
 sp->voice_age = v->age;
 sp->voice_gender = v->gender;

 if (variant_name[0] == '!' && variant_name[1] == 'v' &&
variant_name[2] == PATHSEP)
  variant_name += 3; // strip variant directory name, !v plus
PATHSEP
 strncpy0(base_voice_variant_name, variant_name,
sizeof(base_voice_variant_name));
 memcpy(&base_voice, &current_voice_selected,
sizeof(base_voice));
}

static void RemoveChar(char *p)
{
 // Replace a UTF-8 character by spaces
 int c;

 memset(p, ' ', utf8_in(&c, p));
}

static MNEM_TAB xml_char_mnemonics[] = {
 { "gt",   '>' },
 { "lt",   0xe000 + '<' },   // private usage area, to avoid
confusion with XML tag
 { "amp",  '&' },
 { "quot", '"' },
 { "nbsp", ' ' },
 { "apos", '\'' },
 { NULL,   -1 }
```

```
};

int ReadClause(Translator *tr, char *buf, short *charix, int
*charix_top, int n_buf, int *tone_type, char *voice_change)
{
 /* Find the end of the current clause.
     Write the clause into  buf

     returns: clause type (bits 0-7: pause x10mS, bits 8-11
intonation type)

     Also checks for blank line (paragraph) as end-of-clause
indicator.

     Does not end clause for:
         punctuation immediately followed by alphanumeric  eg.
1.23  !Speak  :path
         repeated punctuation, eg.   ...   !!!
  */

 int c1 = ' '; // current character
 int c2; // next character
 int cprev = ' '; // previous character
 int cprev2 = ' ';
 int c_next;
 int parag;
 int ix = 0;
 int j;
 int nl_count;
 int linelength = 0;
 int phoneme_mode = 0;
 int n_xml_buf;
 int terminator;
 int found;
 bool any_alnum = false;
 bool self_closing;
 int punct_data = 0;
```

```
 bool is_end_clause;
 int announced_punctuation = 0;
 bool stressed_word = false;
 int end_clause_after_tag = 0;
 int end_clause_index = 0;
 wchar_t xml_buf[N_XML_BUF+1];

 #define N_XML_BUF2 20
 char xml_buf2[N_XML_BUF2+2]; // for &<name> and &<number>
sequences
 static char ungot_string[N_XML_BUF2+4];
 static int ungot_string_ix = -1;

 if (clear_skipping_text) {
  skipping_text = false;
  clear_skipping_text = false;
 }

 tr->phonemes_repeat_count = 0;
 tr->clause_upper_count = 0;
 tr->clause_lower_count = 0;

 if (ungot_word != NULL) {
  strcpy(buf, ungot_word);
  ix += strlen(ungot_word);
  ungot_word = NULL;
 }

 if (ungot_char2 != 0)
  c2 = ungot_char2;
 else
  c2 = GetC();

 while (!Eof() || (ungot_char != 0) || (ungot_char2 != 0) ||
(ungot_string_ix >= 0)) {
  if (!iswalnum(c1)) {
   if ((end_character_position > 0) && (count_characters >
```

```
end_character_position)) {
    return CLAUSE_EOF;
   }

   if ((skip_characters > 0) && (count_characters >=
skip_characters)) {
    // reached the specified start position
    // don't break a word
    clear_skipping_text = true;
    skip_characters = 0;
    UngetC(c2);
    return CLAUSE_NONE;
   }
  }

  cprev2 = cprev;
  cprev = c1;
  c1 = c2;

  if (ungot_string_ix >= 0) {
   if (ungot_string[ungot_string_ix] == 0)
    ungot_string_ix = -1;
  }

  if ((ungot_string_ix == 0) && (ungot_char2 == 0))
   c1 = ungot_string[ungot_string_ix++];
  if (ungot_string_ix >= 0)
   c2 = ungot_string[ungot_string_ix++];
  else {
   c2 = GetC();

   if (Eof())
    c2 = ' ';
  }
  ungot_char2 = 0;

  if ((option_ssml) && (phoneme_mode == 0)) {
```

```
   if ((ssml_ignore_l_angle != '&') && (c1 == '&') && ((c2 ==
'#') || ((c2 >= 'a') && (c2 <= 'z')))) {
    n_xml_buf = 0;
    c1 = c2;
    while (!Eof() && (iswalnum(c1) || (c1 == '#')) && (n_xml_buf
< N_XML_BUF2)) {
     xml_buf2[n_xml_buf++] = c1;
     c1 = GetC();
    }
    xml_buf2[n_xml_buf] = 0;
    c2 = GetC();
    sprintf(ungot_string, "%s%c%c", &xml_buf2[0], c1, c2);

    if (c1 == ';') {
     if (xml_buf2[0] == '#') {
      // character code number
      if (xml_buf2[1] == 'x')
       found = sscanf(&xml_buf2[2], "%x", (unsigned int *)(&c1));
      else
       found = sscanf(&xml_buf2[1], "%d", &c1);
     } else {
      if ((found = LookupMnem(xml_char_mnemonics, xml_buf2)) !=
-1) {
       c1 = found;
       if (c2 == 0)
        c2 = ' ';
      }
     }
    } else
     found = -1;

    if (found <= 0) {
     ungot_string_ix = 0;
     c1 = '&';
     c2 = ' ';
    }
```

```
    if ((c1 <= 0x20) && ((sayas_mode == SAYAS_SINGLE_CHARS) ||
(sayas_mode == SAYAS_KEY)))
     c1 += 0xe000; // move into unicode private usage area
   } else if ((c1 == '<') && (ssml_ignore_l_angle != '<')) {
    if ((c2 == '!') || (c2 == '?')) {
     // a comment, ignore until closing '<'  (or <?xml tag )
     while (!Eof() && (c1 != '>'))
      c1 = GetC();
     c2 = ' ';
    } else if ((c2 == '/') || iswalpha(c2)) {
     // check for space in the output buffer for embedded
commands produced by the SSML tag
     if (ix > (n_buf - 20)) {
      // Perhaps not enough room, end the clause before the SSML
tag
      UngetC(c2);
      ungot_char2 = c1;
      buf[ix] = ' ';
      buf[ix+1] = 0;
      return CLAUSE_NONE;
     }

     // SSML Tag
     n_xml_buf = 0;
     c1 = c2;
     while (!Eof() && (c1 != '>') && (n_xml_buf < N_XML_BUF)) {
      xml_buf[n_xml_buf++] = c1;
      c1 = GetC();
     }
     xml_buf[n_xml_buf] = 0;
     c2 = ' ';

     self_closing = false;
     if (xml_buf[n_xml_buf-1] == '/') {
      // a self-closing tag
      xml_buf[n_xml_buf-1] = ' ';
      self_closing = true;
```

```
    }

    terminator = ProcessSsmlTag(xml_buf, buf, &ix, n_buf,
self_closing, xmlbase, &audio_text, current_voice_id,
&base_voice, base_voice_variant_name, &ignore_text,
&clear_skipping_text, &sayas_mode, &sayas_start, ssml_stack,
&n_ssml_stack, &n_param_stack, (int *)speech_parameters);

    if (terminator != 0) {
     buf[ix] = ' ';
     buf[ix++] = 0;

     if (terminator & CLAUSE_TYPE_VOICE_CHANGE)
      strcpy(voice_change, current_voice_id);
     return terminator;
    }
    c1 = ' ';
    c2 = GetC();
    continue;
   }
  }
 }
 ssml_ignore_l_angle = 0;

 if (ignore_text)
  continue;

 if ((c2 == '\n') && (option_linelength == -1)) {
  // single-line mode, return immediately on NL
  if ((terminator = clause_type_from_codepoint(c1)) ==
CLAUSE_NONE) {
   charix[ix] = count_characters - clause_start_char;
   *charix_top = ix;
   ix += utf8_out(c1, &buf[ix]);
   terminator = CLAUSE_PERIOD; // line doesn't end in
punctuation, assume period
  }
```

```
   buf[ix] = ' ';
   buf[ix+1] = 0;
   return terminator;
  }

  if ((c1 == CTRL_EMBEDDED) || (c1 == ctrl_embedded)) {
   // an embedded command. If it's a voice change, end the clause
   if (c2 == 'V') {
    buf[ix++] = 0; // end the clause at this point
    while (!iswspace(c1 = GetC()) && !Eof() && (ix < (n_buf-1)))
     buf[ix++] = c1; // add voice name to end of buffer, after
the text
    buf[ix++] = 0;
    return CLAUSE_VOICE;
   } else if (c2 == 'B') {
    // set the punctuation option from an embedded command
    //  B0     B1     B<punct list><space>
    strcpy(&buf[ix], "   ");
    ix += 3;

    if ((c2 = GetC()) == '0')
     option_punctuation = 0;
    else {
     option_punctuation = 1;
     option_punctlist[0] = 0;
     if (c2 != '1') {
      // a list of punctuation characters to be spoken,
terminated by space
      j = 0;
      while (!iswspace(c2) && !Eof()) {
       option_punctlist[j++] = c2;
       c2 = GetC();
       buf[ix++] = ' ';
      }
      option_punctlist[j] = 0; // terminate punctuation list
      option_punctuation = 2;
     }
```

```
    }
    c2 = GetC();
    continue;
   }
  }

  linelength++;

  if ((j = lookupwchar2(tr->chars_ignore, c1)) != 0) {
   if (j == 1) {
    // ignore this character (eg. zero-width-non-joiner U+200C)
    continue;
   }
   c1 = j; // replace the character
  }

  if (iswalnum(c1))
   any_alnum = true;
  else {
   if (stressed_word) {
    stressed_word = false;
    c1 = CHAR_EMPHASIS; // indicate this word is stressed
    UngetC(c2);
    c2 = ' ';
   }

   if (c1 == 0xf0b)
    c1 = ' '; // Tibet inter-syllabic mark, ?? replace by space
??

   if (iswspace(c1)) {
    char *p_word;

    if (tr->translator_name == 0x6a626f) {
     // language jbo : lojban
     // treat "i" or ".i" as end-of-sentence
     p_word = &buf[ix-1];
```

```
      if (p_word[0] == 'i') {
       if (p_word[-1] == '.')
        p_word--;
       if (p_word[-1] == ' ') {
        ungot_word = "i ";
        UngetC(c2);
        p_word[0] = 0;
        return CLAUSE_PERIOD;
       }
      }
     }
    }

    if (c1 == 0xd4d) {
     // Malayalam virama, check if next character is Zero-width-
joiner
     if (c2 == 0x200d)
      c1 = 0xd4e; // use this unofficial code for chillu-virama
    }
   }

   if (iswupper(c1)) {
    tr->clause_upper_count++;
    if ((option_capitals == 2) && (sayas_mode == 0) &&
!iswupper(cprev)) {
     char text_buf[40];
     char text_buf2[30];
     if (LookupSpecial(tr, "_cap", text_buf2) != NULL) {
      sprintf(text_buf, "%s", text_buf2);
      j = strlen(text_buf);
      if ((ix + j) < n_buf) {
       strcpy(&buf[ix], text_buf);
       ix += j;
      }
     }
    }
   } else if (iswalpha(c1))
```

```
   tr->clause_lower_count++;

  if (option_phoneme_input) {
   if (phoneme_mode > 0)
    phoneme_mode--;
   else if ((c1 == '[') && (c2 == '['))
    phoneme_mode = -1; // input is phoneme mnemonics, so don't
look for punctuation
   else if ((c1 == ']') && (c2 == ']'))
    phoneme_mode = 2; // set phoneme_mode to zero after the next
two characters
  }

  if (c1 == '\n') {
   parag = 0;

   // count consecutive newlines, ignoring other spaces
   while (!Eof() && iswspace(c2)) {
    if (c2 == '\n')
     parag++;
    c2 = GetC();
   }
   if (parag > 0) {
    // 2nd newline, assume paragraph
    UngetC(c2);

    if (end_clause_after_tag)
     RemoveChar(&buf[end_clause_index]); // delete clause-end
punctiation
    buf[ix] = ' ';
    buf[ix+1] = 0;
    if (parag > 3)
     parag = 3;
    if (option_ssml) parag = 1;
    return (CLAUSE_PARAGRAPH-30) + 30*parag; // several blank
lines, longer pause
   }
```

```
   if (linelength <= option_linelength) {
    // treat lines shorter than a specified length as end-of-
clause
    UngetC(c2);
    buf[ix] = ' ';
    buf[ix+1] = 0;
    return CLAUSE_COLON;
   }

   linelength = 0;
  }

  announced_punctuation = 0;

  if ((phoneme_mode == 0) && (sayas_mode == 0)) {
   is_end_clause = false;

   if (end_clause_after_tag) {
    // Because of an xml tag, we are waiting for the
    // next non-blank character to decide whether to end the
clause
    // i.e. is dot followed by an upper-case letter?

    if (!iswspace(c1)) {
     if (!IsAlpha(c1) || !iswlower(c1)) {
      UngetC(c2);
      ungot_char2 = c1;
      buf[end_clause_index] = ' '; // delete the end-clause
punctuation
      buf[end_clause_index+1] = 0;
      return end_clause_after_tag;
     }
     end_clause_after_tag = 0;
    }
   }
```

```
   if ((c1 == '.') && (c2 == '.')) {
    while ((c_next = GetC()) == '.') {
     // 3 or more dots, replace by elipsis
     c1 = 0x2026;
     c2 = ' ';
    }
    if (c1 == 0x2026)
     c2 = c_next;
    else
     UngetC(c_next);
   }

   punct_data = 0;
   if ((punct_data = clause_type_from_codepoint(c1)) !=
CLAUSE_NONE) {
    if (punct_data & CLAUSE_PUNCTUATION_IN_WORD) {
     // Armenian punctuation inside a word
     stressed_word = true;
     *tone_type = punct_data >> 12 & 0xf; // override the end-of-
sentence type
     continue;
    }

    if ((iswspace(c2) || (punct_data &
CLAUSE_OPTIONAL_SPACE_AFTER) || IsBracket(c2) || (c2 == '?') ||
Eof() || (c2 == ctrl_embedded))) { // don't check for '-' because
it prevents recognizing ':-)'
     // note: (c2='?') is for when a smart-quote has been
replaced by '?'
     is_end_clause = true;
    }
   }

   // don't announce punctuation for the alternative text inside
inside <audio> ... </audio>
   if (c1 == 0xe000+'<')  c1 = '<';
   if (option_punctuation && iswpunct(c1) && (audio_text ==
```

```
false)) {
    // option is set to explicitly speak punctuation characters
    // if a list of allowed punctuation has been set up, check
whether the character is in it
    if ((option_punctuation == 1) || (wcschr(option_punctlist,
c1) != NULL)) {
     tr->phonemes_repeat_count = 0;
     if ((terminator = AnnouncePunctuation(tr, c1, &c2, buf, &ix,
is_end_clause)) >= 0)
      return terminator;
     announced_punctuation = c1;
    }
   }

   if ((punct_data & CLAUSE_SPEAK_PUNCTUATION_NAME) &&
(announced_punctuation == 0)) {
    // used for elipsis (and 3 dots) if a pronunciation for
elipsis is given in *_list
    char *p2;

    p2 = &buf[ix];
    sprintf(p2, "%s", LookupCharName(tr, c1, 1));
    if (p2[0] != 0) {
     ix += strlen(p2);
     announced_punctuation = c1;
     punct_data = punct_data & ~CLAUSE_INTONATION_TYPE; // change
intonation type to 0 (full-stop)
    }
   }

   if (is_end_clause) {
    nl_count = 0;
    c_next = c2;

    if (iswspace(c_next)) {
     while (!Eof() && iswspace(c_next)) {
      if (c_next == '\n')
```

```
      nl_count++;
     c_next = GetC(); // skip past space(s)
    }
   }

   if ((c1 == '.') && (nl_count < 2))
    punct_data |= CLAUSE_DOT_AFTER_LAST_WORD;

   if (nl_count == 0) {
    if ((c1 == ',') && (cprev == '.') && (tr->translator_name ==
L('h', 'u')) && iswdigit(cprev2) && (iswdigit(c_next) ||
(iswlower(c_next)))) {
     // lang=hu, fix for ordinal numbers, eg:  "december 2.,
szerda", ignore ',' after ordinal number
     c1 = CHAR_COMMA_BREAK;
     is_end_clause = false;
    }

    if (c1 == '.') {
     if ((tr->langopts.numbers & NUM_ORDINAL_DOT) &&
         (iswdigit(cprev) || (IsRomanU(cprev) &&
(IsRomanU(cprev2) || iswspace(cprev2))))) { // lang=hu
      // dot after a number indicates an ordinal number
      if (!iswdigit(cprev))
       is_end_clause = false; // Roman number followed by dot
      else if (iswlower(c_next) || (c_next == '-')) // hyphen is
needed for lang-hu (eg. 2.-kal)
       is_end_clause = false; // only if followed by lower-case,
(or if there is a XML tag)
     } else if (c_next == '\'')
      is_end_clause = false;    // eg. u.s.a.'s
     if (iswlower(c_next)) {
      // next word has no capital letter, this dot is probably
from an abbreviation
      is_end_clause = 0;
     }
     if (any_alnum == false) {
```

```
       // no letters or digits yet, so probably not a sentence
terminator
       // Here, dot is followed by space or bracket
       c1 = ' ';
       is_end_clause = false;
      }
     } else {
      if (any_alnum == false) {
       // no letters or digits yet, so probably not a sentence
terminator
       is_end_clause = false;
      }
     }

     if (is_end_clause && (c1 == '.') && (c_next == '<') &&
option_ssml) {
      // wait until after the end of the xml tag, then look for
upper-case letter
      is_end_clause = false;
      end_clause_index = ix;
      end_clause_after_tag = punct_data;
     }
    }

    if (is_end_clause) {
     UngetC(c_next);
     buf[ix] = ' ';
     buf[ix+1] = 0;

     if (iswdigit(cprev) && !IsAlpha(c_next)) // ????
      punct_data &= ~CLAUSE_DOT_AFTER_LAST_WORD;
     if (nl_count > 1) {
      if ((punct_data == CLAUSE_QUESTION) || (punct_data ==
CLAUSE_EXCLAMATION))
       return punct_data + 35; // with a longer pause
      return CLAUSE_PARAGRAPH;
     }
```

```
     return punct_data; // only recognise punctuation if followed
by a blank or bracket/quote
    } else if (!Eof()) {
     if (iswspace(c2))
      UngetC(c_next);
    }
   }
  }

  if (speech_parameters[espeakSILENCE] == 1)
   continue;

  if (c1 == announced_punctuation) {
   // This character has already been announced, so delete it so
that it isn't spoken a second time.
   // Unless it's a hyphen or apostrophe (which is used by
TranslateClause() )
   if (IsBracket(c1))
    c1 = 0xe000 + '('; // Unicode private useage area.  So
TranslateRules() knows the bracket name has been spoken
   else if (c1 != '-')
    c1 = ' ';
  }

  j = ix+1;

  if (c1 == 0xe000 + '<') c1 = '<';

  ix += utf8_out(c1, &buf[ix]);
  if (!iswspace(c1) && !IsBracket(c1)) {
   charix[ix] = count_characters - clause_start_char;
   while (j < ix)
    charix[j++] = -1; // subsequent bytes of a multibyte
character
  }
  *charix_top = ix;
```

```
  if (((ix > (n_buf-75)) && !IsAlpha(c1) && !iswdigit(c1))  ||
(ix >= (n_buf-4))) {
   // clause too long, getting near end of buffer, so break here
   // try to break at a word boundary (unless we actually reach
the end of buffer).
   // (n_buf-4) is to allow for 3 bytes of multibyte character
plus terminator.
   buf[ix] = ' ';
   buf[ix+1] = 0;
   UngetC(c2);
   return CLAUSE_NONE;
  }
 }

 if (stressed_word)
  ix += utf8_out(CHAR_EMPHASIS, &buf[ix]);
 if (end_clause_after_tag)
  RemoveChar(&buf[end_clause_index]); // delete clause-end
punctiation
 buf[ix] = ' ';
 buf[ix+1] = 0;
 return CLAUSE_EOF; // end of file
}

void InitNamedata(void)
{
 namedata_ix = 0;
 if (namedata != NULL) {
  free(namedata);
  namedata = NULL;
  n_namedata = 0;
 }
}

void InitText2(void)
{
 int param;
```

```
 ungot_char = 0;
 ungot_char2 = 0;

 n_ssml_stack = 1;
 n_param_stack = 1;
 ssml_stack[0].tag_type = 0;

 for (param = 0; param < N_SPEECH_PARAM; param++)
  speech_parameters[param] = param_stack[0].parameter[param]; //
set all speech parameters to defaults

 option_punctuation = speech_parameters[espeakPUNCTUATION];
 option_capitals = speech_parameters[espeakCAPITALS];

 current_voice_id[0] = 0;

 ignore_text = false;
 audio_text = false;
 clear_skipping_text = false;
 count_characters = -1;
 sayas_mode = 0;

 xmlbase = NULL;
}
```

Chapter 60

./src/libespeak-ng/espeak_command.c

```
#include "config.h"

#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>

#include <espeak-ng/espeak_ng.h>

#include "espeak_command.h"

#ifdef USE_ASYNC

static unsigned int my_current_text_id = 0;

t_espeak_command *create_espeak_text(const void *text, size_t
size, unsigned int position, espeak_POSITION_TYPE position_type,
unsigned int end_position, unsigned int flags, void *user_data)
{
 if (!text || !size)
```

```
  return NULL;

 void *a_text = NULL;
 t_espeak_text *data = NULL;

 t_espeak_command *a_command = (t_espeak_command
*)malloc(sizeof(t_espeak_command));
 if (!a_command)
  return NULL;

 a_text = malloc(size+1);
 if (!a_text) {
  free(a_command);
  return NULL;
 }
 memcpy(a_text, text, size);

 a_command->type = ET_TEXT;
 a_command->state = CS_UNDEFINED;
 data = &(a_command->u.my_text);
 data->unique_identifier = ++my_current_text_id;
 data->text = a_text;
 data->position = position;
 data->position_type = position_type;
 data->end_position = end_position;
 data->flags = flags;
 data->user_data = user_data;

 return a_command;
}

t_espeak_command *create_espeak_terminated_msg(unsigned int
unique_identifier, void *user_data)
{
 t_espeak_terminated_msg *data = NULL;
 t_espeak_command *a_command = (t_espeak_command
*)malloc(sizeof(t_espeak_command));
```

```
 if (!a_command)
  return NULL;

 a_command->type = ET_TERMINATED_MSG;
 a_command->state = CS_UNDEFINED;
 data = &(a_command->u.my_terminated_msg);
 data->unique_identifier = unique_identifier;
 data->user_data = user_data;

 return a_command;
}

t_espeak_command *create_espeak_mark(const void *text, size_t
size, const char *index_mark, unsigned int end_position, unsigned
int flags, void *user_data)
{
 if (!text || !size || !index_mark)
  return NULL;

 void *a_text = NULL;
 char *a_index_mark = NULL;
 t_espeak_mark *data = NULL;

 t_espeak_command *a_command = (t_espeak_command
*)malloc(sizeof(t_espeak_command));
 if (!a_command)
  return NULL;

 a_text = malloc(size);
 if (!a_text) {
  free(a_command);
  return NULL;
 }
 memcpy(a_text, text, size);

 a_index_mark = strdup(index_mark);
```

```
 a_command->type = ET_MARK;
 a_command->state = CS_UNDEFINED;
 data = &(a_command->u.my_mark);
 data->unique_identifier = ++my_current_text_id;
 data->text = a_text;
 data->index_mark = a_index_mark;
 data->end_position = end_position;
 data->flags = flags;
 data->user_data = user_data;

 return a_command;
}

t_espeak_command *create_espeak_key(const char *key_name, void
*user_data)
{
 if (!key_name)
  return NULL;

 t_espeak_command *a_command = (t_espeak_command
*)malloc(sizeof(t_espeak_command));
 if (!a_command)
  return NULL;

 a_command->type = ET_KEY;
 a_command->state = CS_UNDEFINED;
 a_command->u.my_key.user_data = user_data;
 a_command->u.my_key.unique_identifier = ++my_current_text_id;
 a_command->u.my_key.key_name = strdup(key_name);

 return a_command;
}

t_espeak_command *create_espeak_char(wchar_t character, void
*user_data)
{
 t_espeak_command *a_command = (t_espeak_command
```

```
*)malloc(sizeof(t_espeak_command));
 if (!a_command)
  return NULL;

 a_command->type = ET_CHAR;
 a_command->state = CS_UNDEFINED;
 a_command->u.my_char.user_data = user_data;
 a_command->u.my_char.unique_identifier = ++my_current_text_id;
 a_command->u.my_char.character = character;

 return a_command;
}

t_espeak_command *create_espeak_parameter(espeak_PARAMETER
parameter, int value, int relative)
{
 t_espeak_parameter *data = NULL;

 t_espeak_command *a_command = (t_espeak_command
*)malloc(sizeof(t_espeak_command));
 if (!a_command)
  return NULL;

 a_command->type = ET_PARAMETER;
 a_command->state = CS_UNDEFINED;
 data = &(a_command->u.my_param);
 data->parameter = parameter;
 data->value = value;
 data->relative = relative;

 return a_command;
}

t_espeak_command *create_espeak_punctuation_list(const wchar_t
*punctlist)
{
 if (!punctlist)
```

```
  return NULL;

 t_espeak_command *a_command = (t_espeak_command
*)malloc(sizeof(t_espeak_command));
 if (!a_command)
  return NULL;

 a_command->type = ET_PUNCTUATION_LIST;
 a_command->state = CS_UNDEFINED;

 size_t len = (wcslen(punctlist) + 1)*sizeof(wchar_t);
 wchar_t *a_list = (wchar_t *)malloc(len);
 if (a_list == NULL) {
  free(a_command);
  return NULL;
 }
 memcpy(a_list, punctlist, len);
 a_command->u.my_punctuation_list = a_list;

 return a_command;
}

t_espeak_command *create_espeak_voice_name(const char *name)
{
 if (!name)
  return NULL;

 t_espeak_command *a_command = (t_espeak_command
*)malloc(sizeof(t_espeak_command));
 if (!a_command)
  return NULL;

 a_command->type = ET_VOICE_NAME;
 a_command->state = CS_UNDEFINED;
 a_command->u.my_voice_name = strdup(name);

 return a_command;
```

```
}

t_espeak_command *create_espeak_voice_spec(espeak_VOICE *voice)
{
 if (!voice)
  return NULL;

 t_espeak_command *a_command = (t_espeak_command
*)malloc(sizeof(t_espeak_command));
 if (!a_command)
  return NULL;

 a_command->type = ET_VOICE_SPEC;
 a_command->state = CS_UNDEFINED;

 espeak_VOICE *data = &(a_command->u.my_voice_spec);
 memcpy(data, voice, sizeof(espeak_VOICE));

 if (voice->name)
  data->name = strdup(voice->name);

 if (voice->languages)
  data->languages = strdup(voice->languages);

 if (voice->identifier)
  data->identifier = strdup(voice->identifier);

 return a_command;
}

int delete_espeak_command(t_espeak_command *the_command)
{
 int a_status = 0;
 if (the_command) {
  switch (the_command->type)
  {
  case ET_TEXT:
```

```
   if (the_command->u.my_text.text)
    free(the_command->u.my_text.text);
   break;
  case ET_MARK:
   if (the_command->u.my_mark.text)
    free(the_command->u.my_mark.text);
   if (the_command->u.my_mark.index_mark)
    free((void *)(the_command->u.my_mark.index_mark));
   break;
  case ET_TERMINATED_MSG:
  {
   // if the terminated msg is pending,
   // it must be processed here for informing the calling program
   // that its message is finished.
   // This can be important for cleaning the related user data.
   t_espeak_terminated_msg *data =
&(the_command->u.my_terminated_msg);
   if (the_command->state == CS_PENDING) {
    the_command->state = CS_PROCESSED;
    sync_espeak_terminated_msg(data->unique_identifier,
data->user_data);
   }
  }
   break;
  case ET_KEY:
   if (the_command->u.my_key.key_name)
    free((void *)(the_command->u.my_key.key_name));
   break;
  case ET_CHAR:
  case ET_PARAMETER:
   // No allocation
   break;
  case ET_PUNCTUATION_LIST:
   if (the_command->u.my_punctuation_list)
    free((void *)(the_command->u.my_punctuation_list));
   break;
  case ET_VOICE_NAME:
```

```
   if (the_command->u.my_voice_name)
    free((void *)(the_command->u.my_voice_name));
   break;
  case ET_VOICE_SPEC:
  {
   espeak_VOICE *data = &(the_command->u.my_voice_spec);

   if (data->name)
    free((void *)data->name);

   if (data->languages)
    free((void *)data->languages);

   if (data->identifier)
    free((void *)data->identifier);
  }
   break;
  default:
   assert(0);
  }
  free(the_command);
  a_status = 1;
 }
 return a_status;
}

void process_espeak_command(t_espeak_command *the_command)
{
 if (the_command == NULL)
  return;

 the_command->state = CS_PROCESSED;

 switch (the_command->type)
 {
 case ET_TEXT:
 {
```

```
  t_espeak_text *data = &(the_command->u.my_text);
  sync_espeak_Synth(data->unique_identifier, data->text,
                    data->position, data->position_type,
                    data->end_position, data->flags,
data->user_data);
 }
  break;
 case ET_MARK:
 {
  t_espeak_mark *data = &(the_command->u.my_mark);
  sync_espeak_Synth_Mark(data->unique_identifier, data->text,
                         data->index_mark, data->end_position,
data->flags,
                         data->user_data);
 }
  break;
 case ET_TERMINATED_MSG:
 {
  t_espeak_terminated_msg *data =
&(the_command->u.my_terminated_msg);
  sync_espeak_terminated_msg(data->unique_identifier,
data->user_data);
 }
  break;
 case ET_KEY:
 {
  const char *data = the_command->u.my_key.key_name;
  sync_espeak_Key(data);
 }
  break;
 case ET_CHAR:
 {
  const wchar_t data = the_command->u.my_char.character;
  sync_espeak_Char(data);
 }
  break;
 case ET_PARAMETER:
```

```
 {
  t_espeak_parameter *data = &(the_command->u.my_param);
  SetParameter(data->parameter, data->value, data->relative);
 }
  break;
 case ET_PUNCTUATION_LIST:
 {
  const wchar_t *data = the_command->u.my_punctuation_list;
  sync_espeak_SetPunctuationList(data);
 }
  break;
 case ET_VOICE_NAME:
 {
  const char *data = the_command->u.my_voice_name;
  espeak_SetVoiceByName(data);
 }
  break;
 case ET_VOICE_SPEC:
 {
  espeak_VOICE *data = &(the_command->u.my_voice_spec);
  espeak_SetVoiceByProperties(data);
 }
  break;
 default:
  assert(0);
  break;
 }
}

#endif
```

Chapter 61

./src/libespeak-ng/dictionary.c

```
#include "config.h"

#include <ctype.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wctype.h>
#include <wchar.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "dictionary.h"
#include "numbers.h"
#include "readclause.h"
#include "synthdata.h"

#include "speech.h"
#include "phoneme.h"
#include "voice.h"
```

```
#include "synthesize.h"
#include "translate.h"

typedef struct {
 int points;
 const char *phonemes;
 int end_type;
 char *del_fwd;
} MatchRecord;

int dictionary_skipwords;
char dictionary_name[40];

// accented characters which indicate (in some languages) the
start of a separate syllable
static const unsigned short diereses_list[7] = { 0xe4, 0xeb,
0xef, 0xf6, 0xfc, 0xff, 0 };

// convert characters to an approximate 7 bit ascii equivalent
// used for checking for vowels (up to 0x259=schwa)
#define N_REMOVE_ACCENT  0x25e
static unsigned char remove_accent[N_REMOVE_ACCENT] = {
 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'c', 'e', 'e', 'e', 'e', 'i',
'i', 'i', 'i',  // 0c0
 'd', 'n', 'o', 'o', 'o', 'o', 'o',   0, 'o', 'u', 'u', 'u', 'u',
'y', 't', 's',  // 0d0
 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'c', 'e', 'e', 'e', 'e', 'i',
'i', 'i', 'i',  // 0e0
 'd', 'n', 'o', 'o', 'o', 'o', 'o',   0, 'o', 'u', 'u', 'u', 'u',
'y', 't', 'y',  // 0f0

 'a', 'a', 'a', 'a', 'a', 'a', 'c', 'c', 'c', 'c', 'c', 'c', 'c',
'c', 'd', 'd',  // 100
 'd', 'd', 'e', 'e', 'e', 'e', 'e', 'e', 'e', 'e', 'e', 'e', 'g',
'g', 'g', 'g',  // 110
 'g', 'g', 'g', 'g', 'h', 'h', 'h', 'h', 'i', 'i', 'i', 'i', 'i',
'i', 'i', 'i',  // 120
```

```
 'i', 'i', 'i', 'i', 'j', 'j', 'k', 'k', 'k', 'l', 'l', 'l', 'l',
'l', 'l', 'l',  // 130
 'l', 'l', 'l', 'n', 'n', 'n', 'n', 'n', 'n', 'n', 'n', 'n', 'o',
'o', 'o', 'o',  // 140
 'o', 'o', 'o', 'o', 'r', 'r', 'r', 'r', 'r', 'r', 's', 's', 's',
's', 's', 's',  // 150
 's', 's', 't', 't', 't', 't', 't', 't', 'u', 'u', 'u', 'u', 'u',
'u', 'u', 'u',  // 160
 'u', 'u', 'u', 'u', 'w', 'w', 'y', 'y', 'y', 'z', 'z', 'z', 'z',
'z', 'z', 's',  // 170
 'b', 'b', 'b', 'b',   0,   0, 'o', 'c', 'c', 'd', 'd', 'd', 'd',
'd', 'e', 'e',  // 180
 'e', 'f', 'f', 'g', 'g', 'h', 'i', 'i', 'k', 'k', 'l', 'l', 'm',
'n', 'n', 'o',  // 190
 'o', 'o', 'o', 'o', 'p', 'p', 'y',   0,   0, 's', 's', 't', 't',
't', 't', 'u',  // 1a0
 'u', 'u', 'v', 'y', 'y', 'z', 'z', 'z', 'z', 'z', 'z', 'z',   0,
0,   0, 'w',  // 1b0
 't', 't', 't', 'k', 'd', 'd', 'd', 'l', 'l', 'l', 'n', 'n', 'n',
'a', 'a', 'i',  // 1c0
 'i', 'o', 'o', 'u', 'u', 'u', 'u', 'u', 'u', 'u', 'u', 'u', 'u',
'e', 'a', 'a',  // 1d0
 'a', 'a', 'a', 'a', 'g', 'g', 'g', 'g', 'k', 'k', 'o', 'o', 'o',
'o', 'z', 'z',  // 1e0
 'j', 'd', 'd', 'd', 'g', 'g', 'w', 'w', 'n', 'n', 'a', 'a', 'a',
'a', 'o', 'o',  // 1f0

 'a', 'a', 'a', 'a', 'e', 'e', 'e', 'e', 'i', 'i', 'i', 'i', 'o',
'o', 'o', 'o',  // 200
 'r', 'r', 'r', 'r', 'u', 'u', 'u', 'u', 's', 's', 't', 't', 'y',
'y', 'h', 'h',  // 210
 'n', 'd', 'o', 'o', 'z', 'z', 'a', 'a', 'e', 'e', 'o', 'o', 'o',
'o', 'o', 'o',  // 220
 'o', 'o', 'y', 'y', 'l', 'n', 't', 'j', 'd', 'q', 'a', 'c', 'c',
'l', 't', 's',  // 230
 'z',   0,   0, 'b', 'u', 'v', 'e', 'e', 'j', 'j', 'q', 'q', 'r',
'r', 'y', 'y',  // 240
```

```
 'a', 'a', 'a', 'b', 'o', 'c', 'd', 'd', 'e', 'e', 'e', 'e', 'e',
'e'
};

#pragma GCC visibility push(default)
void strncpy0(char *to, const char *from, int size)
{
 // strcpy with limit, ensures a zero terminator
 strncpy(to, from, size);
 to[size-1] = 0;
}
#pragma GCC visibility pop

static int Reverse4Bytes(int word)
{
 // reverse the order of bytes from little-endian to big-endian
#ifdef ARCH_BIG
 int ix;
 int word2 = 0;

 for (ix = 0; ix <= 24; ix += 8) {
  word2 = word2 << 8;
  word2 |= (word >> ix) & 0xff;
 }
 return word2;
#else
 return word;
#endif
}

static void InitGroups(Translator *tr)
{
 // Called after dictionary 1 is loaded, to set up table of entry
points for translation rule chains
 // for single-letters and two-letter combinations

 int ix;
```

```
 char *p;
 char *p_name;
 unsigned char c, c2;
 int len;

 tr->n_groups2 = 0;
 for (ix = 0; ix < 256; ix++) {
  tr->groups1[ix] = NULL;
  tr->groups2_count[ix] = 0;
  tr->groups2_start[ix] = 255; // indicates "not set"
 }
 memset(tr->letterGroups, 0, sizeof(tr->letterGroups));
 memset(tr->groups3, 0, sizeof(tr->groups3));

 p = tr->data_dictrules;
 // If there are no rules in the dictionary, compile_dictrules
will not
 // write a RULE_GROUP_START (written in the for loop), but will
write
 // a RULE_GROUP_END.
 if (*p != RULE_GROUP_END) while (*p != 0) {
  if (*p != RULE_GROUP_START) {
   fprintf(stderr, "Bad rules data in '%s_dict' at 0x%x (%c)\n",
dictionary_name, (unsigned int)(p - tr->data_dictrules), *p);
   break;
  }
  p++;

  if (p[0] == RULE_REPLACEMENTS) {
   p = (char *)(((intptr_t)p+4) & ~3); // advance to next word
boundary
   tr->langopts.replace_chars = (unsigned char *)p;

   while ( !is_str_totally_null(p, 4) ) {
    p++;
   }
```

```
 while (*p != RULE_GROUP_END) p++;
 p++;
 continue;
}

if (p[0] == RULE_LETTERGP2) {
 ix = p[1] - 'A';
 if (ix < 0)
  ix += 256;
 p += 2;
 if ((ix >= 0) && (ix < N_LETTER_GROUPS))
  tr->letterGroups[ix] = p;
} else {
 len = strlen(p);
 p_name = p;
 c = p_name[0];
 c2 = p_name[1];

 p += (len+1);
 if (len == 1)
  tr->groups1[c] = p;
 else if (len == 0)
  tr->groups1[0] = p;
 else if (c == 1) {
  // index by offset from letter base
  tr->groups3[c2 - 1] = p;
 } else {
  if (tr->groups2_start[c] == 255)
   tr->groups2_start[c] = tr->n_groups2;

  tr->groups2_count[c]++;
  tr->groups2[tr->n_groups2] = p;
  tr->groups2_name[tr->n_groups2++] = (c + (c2 << 8));
 }
}

// skip over all the rules in this group
```

```
  while (*p != RULE_GROUP_END)
   p += (strlen(p) + 1);
  p++;
 }
}

int LoadDictionary(Translator *tr, const char *name, int
no_error)
{
 int hash;
 char *p;
 int *pw;
 int length;
 FILE *f;
 int size;
 char fname[sizeof(path_home)+20];

 if (dictionary_name != name)
  strncpy(dictionary_name, name, 40); // currently loaded
dictionary name
 if (tr->dictionary_name != name)
  strncpy(tr->dictionary_name, name, 40);

 // Load a pronunciation data file into memory
 // bytes 0-3:  offset to rules data
 // bytes 4-7:  number of hash table entries
 sprintf(fname, "%s%c%s_dict", path_home, PATHSEP, name);
 size = GetFileLength(fname);

 if (tr->data_dictlist != NULL) {
  free(tr->data_dictlist);
  tr->data_dictlist = NULL;
 }

 f = fopen(fname, "rb");
 if ((f == NULL) || (size <= 0)) {
  if (no_error == 0)
```

```
   fprintf(stderr, "Can't read dictionary file: '%s'\n", fname);
  if (f != NULL)
   fclose(f);
  return 1;
 }

 if ((tr->data_dictlist = malloc(size)) == NULL) {
  fclose(f);
  return 3;
 }
 size = fread(tr->data_dictlist, 1, size, f);
 fclose(f);

 pw = (int *)(tr->data_dictlist);
 length = Reverse4Bytes(pw[1]);

 if (size <= (N_HASH_DICT + sizeof(int)*2)) {
  fprintf(stderr, "Empty _dict file: '%s\n", fname);
  return 2;
 }

 if ((Reverse4Bytes(pw[0]) != N_HASH_DICT) ||
     (length <= 0) || (length > 0x8000000)) {
  fprintf(stderr, "Bad data: '%s' (%x length=%x)\n", fname,
Reverse4Bytes(pw[0]), length);
  return 2;
 }
 tr->data_dictrules = &(tr->data_dictlist[length]);

 // set up indices into data_dictrules
 InitGroups(tr);

 // set up hash table for data_dictlist
 p = &(tr->data_dictlist[8]);

 for (hash = 0; hash < N_HASH_DICT; hash++) {
  tr->dict_hashtab[hash] = p;
```

```
  while ((length = *(uint8_t *)p) != 0)
   p += length;
  p++; // skip over the zero which terminates the list for this
hash value
 }

 if ((tr->dict_min_size > 0) && (size < (unsigned
int)tr->dict_min_size))
  fprintf(stderr, "Full dictionary is not installed for '%s'\n",
name);

 return 0;
}
```

    This is used to access the dictionary_2 word-lookup dictionary

```
int HashDictionary(const char *string)
{
 int c;
 int chars = 0;
 int hash = 0;

 while ((c = (*string++ & 0xff)) != 0) {
  hash = hash * 8 + c;
  hash = (hash & 0x3ff) ^ (hash >> 8); // exclusive or
  chars++;
 }

 return (hash+chars) & 0x3ff; // a 10 bit hash code
}
```

   from 'p' up to next blank .
   Returns advanced 'p'
   outptr contains encoded phonemes, unrecognized phoneme stops the encoding
   bad_phoneme must point to char array of length 2 of more

```
const char *EncodePhonemes(const char *p, char *outptr, int
*bad_phoneme)
{
 int ix;
 unsigned char c;
 int count;     // num. of matching characters
 int max;       // highest num. of matching found so far
 int max_ph;    // corresponding phoneme with highest matching
 int consumed;
 unsigned int mnemonic_word;

 if (bad_phoneme != NULL)
  *bad_phoneme = 0;

 // skip initial blanks
 while ((uint8_t)*p < 0x80 && isspace(*p))
  p++;

 while (((c = *p) != 0) && !isspace(c)) {
  consumed = 0;

  switch (c)
  {
  case '|':
   // used to separate phoneme mnemonics if needed, to prevent
characters being treated
   // as a multi-letter mnemonic

   if ((c = p[1]) == '|') {
    // treat double || as a word-break symbol, drop through
    // to the default case with c = '|'
   } else {
    p++;
    break;
   }
  default:
```

```
   // lookup the phoneme mnemonic, find the phoneme with the
highest number of
   // matching characters
   max = -1;
   max_ph = 0;

   for (ix = 1; ix < n_phoneme_tab; ix++) {
    if (phoneme_tab[ix] == NULL)
     continue;
    if (phoneme_tab[ix]->type == phINVALID)
     continue; // this phoneme is not defined for this language

    count = 0;
    mnemonic_word = phoneme_tab[ix]->mnemonic;

    while (((c = p[count]) > ' ') && (count < 4) &&
           (c == ((mnemonic_word >> (count*8)) & 0xff)))
     count++;

    if ((count > max) &&
        ((count == 4) || (((mnemonic_word >> (count*8)) & 0xff)
== 0))) {
     max = count;
     max_ph = phoneme_tab[ix]->code;
    }
   }

   if (max_ph == 0) {
    // not recognised, report and ignore
    if (bad_phoneme != NULL)
     utf8_in(bad_phoneme, p);
    *outptr++ = 0;
    return p+1;
   }

   if (max <= 0)
    max = 1;
```

```
   p += (consumed + max);
   *outptr++ = (char)(max_ph);

   if (max_ph == phonSWITCH) {
    // Switch Language: this phoneme is followed by a text string
    char *p_lang = outptr;
    while (!isspace(c = *p) && (c != 0)) {
     p++;
     *outptr++ = tolower(c);
    }
    *outptr = 0;
    if (c == 0) {
     if (strcmp(p_lang, "en") == 0) {
      *p_lang = 0; // don't need "en", it's assumed by default
      return p;
     }
    } else
     *outptr++ = '|'; // more phonemes follow, terminate language
string with separator
   }
   break;
  }
 }
 // terminate the encoded string

 return p;
}

void DecodePhonemes(const char *inptr, char *outptr)
{
 // Translate from internal phoneme codes into phoneme mnemonics
 unsigned char phcode;
 unsigned char c;
 unsigned int mnem;
 PHONEME_TAB *ph;
 static const char *stress_chars = "==,,'*  ";
```

```
 sprintf(outptr, "* ");
 while ((phcode = *inptr++) > 0) {
  if (phcode == 255)
   continue; // indicates unrecognised phoneme
  if ((ph = phoneme_tab[phcode]) == NULL)
   continue;

  if ((ph->type == phSTRESS) && (ph->std_length <= 4) &&
(ph->program == 0)) {
   if (ph->std_length > 1)
    *outptr++ = stress_chars[ph->std_length];
  } else {
   mnem = ph->mnemonic;

   while ((c = (mnem & 0xff)) != 0) {
    *outptr++ = c;
    mnem = mnem >> 8;
   }
   if (phcode == phonSWITCH) {
    while (isalpha(*inptr))
     *outptr++ = *inptr++;
   }
  }
 }

}

// using Kirschenbaum to IPA translation, ascii 0x20 to 0x7f
unsigned short ipa1[96] = {
 0x20,  0x21,  0x22,  0x2b0, 0x24,  0x25,  0x0e6, 0x2c8, 0x28,
0x29,  0x27e, 0x2b,  0x2cc, 0x2d,  0x2e,  0x2f,
 0x252, 0x31,  0x32,  0x25c, 0x34,  0x35,  0x36,  0x37,  0x275,
0x39,  0x2d0, 0x2b2, 0x3c,  0x3d,  0x3e,  0x294,
 0x259, 0x251, 0x3b2, 0xe7,  0xf0,  0x25b, 0x46,  0x262, 0x127,
0x26a, 0x25f, 0x4b,  0x26b, 0x271, 0x14b, 0x254,
 0x3a6, 0x263, 0x280, 0x283, 0x3b8, 0x28a, 0x28c, 0x153, 0x3c7,
0xf8,  0x292, 0x32a, 0x5c,  0x5d,  0x5e,  0x5f,
```

```
 0x60,  0x61,  0x62,  0x63,  0x64,  0x65,  0x66,  0x261, 0x68,
0x69,  0x6a,  0x6b,  0x6c,  0x6d,  0x6e,  0x6f,
 0x70,  0x71,  0x72,  0x73,  0x74,  0x75,  0x76,  0x77,  0x78,
0x79,  0x7a,  0x7b,  0x7c,  0x7d,  0x303, 0x7f
};

#define N_PHON_OUT  500  // realloc increment
static char *phon_out_buf = NULL;   // passes the result of
GetTranslatedPhonemeString()
static unsigned int phon_out_size = 0;

char *WritePhMnemonic(char *phon_out, PHONEME_TAB *ph,
PHONEME_LIST *plist, int use_ipa, int *flags)
{
 int c;
 int mnem;
 int len;
 bool first;
 int ix = 0;
 char *p;
 PHONEME_DATA phdata;

 if (ph->code == phonEND_WORD) {
  // ignore
  phon_out[0] = 0;
  return phon_out;
 }

 if (ph->code == phonSWITCH) {
  // the tone_ph field contains a phoneme table number
  p = phoneme_tab_list[plist->tone_ph].name;
  sprintf(phon_out, "(%s)", p);
  return phon_out + strlen(phon_out);
 }

 if (use_ipa) {
  // has an ipa name been defined for this phoneme ?
```

```
  phdata.ipa_string[0] = 0;

  if (plist == NULL)
   InterpretPhoneme2(ph->code, &phdata);
  else
   InterpretPhoneme(NULL, 0, plist, &phdata, NULL);

  p = phdata.ipa_string;
  if (*p == 0x20) {
   // indicates no name for this phoneme
   *phon_out = 0;
   return phon_out;
  }
  if ((*p != 0) && ((*p & 0xff) < 0x20)) {
   // name starts with a flags byte
   if (flags != NULL)
    *flags = *p;
   p++;
  }

  len = strlen(p);
  if (len > 0) {
   strcpy(phon_out, p);
   phon_out += len;
   *phon_out = 0;
   return phon_out;
  }
 }

 first = true;
 for (mnem = ph->mnemonic; (c = mnem & 0xff) != 0; mnem = mnem >>
8) {
  if (c == '/')
   break; // discard phoneme variant indicator

  if (use_ipa) {
   // convert from ascii to ipa
```

```
   if (first && (c == '_'))
    break; // don't show pause phonemes

   if ((c == '#') && (ph->type == phVOWEL))
    break; // # is subscript-h, but only for consonants

   // ignore digits after the first character
   if (!first && IsDigit09(c))
    continue;

   if ((c >= 0x20) && (c < 128))
    c = ipa1[c-0x20];

   ix += utf8_out(c, &phon_out[ix]);
  } else
   phon_out[ix++] = c;
  first = false;
 }

 phon_out = &phon_out[ix];

 return phon_out;
}

const char *GetTranslatedPhonemeString(int phoneme_mode)
{
 /* Called after a clause has been translated into phonemes, in
order
    to display the clause in phoneme mnemonic form.

    phoneme_mode
                  bit  1:   use IPA phoneme names
                  bit  7:   use tie between letters in multi-
character phoneme names
                  bits 8-23 tie or separator character

  */
```

```
int ix;
unsigned int len;
int phon_out_ix = 0;
int stress;
int c;
char *p;
char *buf;
int count;
int flags;
int use_ipa;
int use_tie;
int separate_phonemes;
char phon_buf[30];
char phon_buf2[30];
PHONEME_LIST *plist;

static const char *stress_chars = "==,,''";

if (phon_out_buf == NULL) {
 phon_out_size = N_PHON_OUT;
 if ((phon_out_buf = (char *)malloc(phon_out_size)) == NULL) {
  phon_out_size = 0;
  return "";
 }
}

use_ipa = phoneme_mode & espeakPHONEMES_IPA;
if (phoneme_mode & espeakPHONEMES_TIE) {
 use_tie = phoneme_mode >> 8;
 separate_phonemes = 0;
} else {
 separate_phonemes = phoneme_mode >> 8;
 use_tie = 0;
}

for (ix = 1; ix < (n_phoneme_list-2); ix++) {
```

```
  buf = phon_buf;

  plist = &phoneme_list[ix];

  WritePhMnemonic(phon_buf2, plist->ph, plist, use_ipa, &flags);
  if (plist->newword & PHLIST_START_OF_WORD && !(plist->newword &
(PHLIST_START_OF_SENTENCE | PHLIST_START_OF_CLAUSE)))
   *buf++ = ' ';

  if ((!plist->newword) || (separate_phonemes == ' ')) {
   if ((separate_phonemes != 0) && (ix > 1)) {
    utf8_in(&c, phon_buf2);
    if ((c < 0x2b0) || (c > 0x36f)) // not if the phoneme starts
with a superscript letter
     buf += utf8_out(separate_phonemes, buf);
   }
  }

  if (plist->synthflags & SFLAG_SYLLABLE) {
   if ((stress = plist->stresslevel) > 1) {
    c = 0;
    if (stress > STRESS_IS_PRIORITY) stress = STRESS_IS_PRIORITY;

    if (use_ipa) {
     c = 0x2cc; // ipa, secondary stress
     if (stress > STRESS_IS_SECONDARY)
      c = 0x02c8; // ipa, primary stress
    } else
     c = stress_chars[stress];

    if (c != 0)
     buf += utf8_out(c, buf);
   }
  }

  flags = 0;
  count = 0;
```

```
  for (p = phon_buf2; *p != 0;) {
   p += utf8_in(&c, p);
   if (use_tie != 0) {
    // look for non-inital alphabetic character, but not
diacritic, superscript etc.
    if ((count > 0) && !(flags & (1 << (count-1))) && ((c <
0x2b0) || (c > 0x36f)) && iswalpha(c))
     buf += utf8_out(use_tie, buf);
   }
   buf += utf8_out(c, buf);
   count++;
  }

  if (plist->ph->code != phonSWITCH) {
   if (plist->synthflags & SFLAG_LENGTHEN)
    buf = WritePhMnemonic(buf, phoneme_tab[phonLENGTHEN], plist,
use_ipa, NULL);
   if ((plist->synthflags & SFLAG_SYLLABLE) && (plist->type !=
phVOWEL)) {
    // syllablic consonant
    buf = WritePhMnemonic(buf, phoneme_tab[phonSYLLABIC], plist,
use_ipa, NULL);
   }
   if (plist->tone_ph > 0)
    buf = WritePhMnemonic(buf, phoneme_tab[plist->tone_ph],
plist, use_ipa, NULL);
  }

  len = buf - phon_buf;
  if ((phon_out_ix + len) >= phon_out_size) {
   // enlarge the phoneme buffer
   phon_out_size = phon_out_ix + len + N_PHON_OUT;
   char *new_phon_out_buf = (char *)realloc(phon_out_buf,
phon_out_size);
   if (new_phon_out_buf == NULL) {
    phon_out_size = 0;
    return "";
```

```
   } else
     phon_out_buf = new_phon_out_buf;
   }

   phon_buf[len] = 0;
   strcpy(&phon_out_buf[phon_out_ix], phon_buf);
   phon_out_ix += len;
  }

 if (!phon_out_buf)
  return "";

 phon_out_buf[phon_out_ix] = 0;

 return phon_out_buf;
}

static int LetterGroupNo(char *rule)
{
 /*
  * Returns number of letter group
  */
 int groupNo = *rule;
 groupNo = groupNo - 'A'; // substracting 'A' makes letter_group
equal to number in .Lxx definition
 if (groupNo < 0)         // fix sign if necessary
  groupNo += 256;
 return groupNo;
}

static int IsLetterGroup(Translator *tr, char *word, int group,
int pre)
{
 /* Match the word against a list of utf-8 strings.
  * returns length of matching letter group or -1
  *
  * How this works:
```

```
 *
 *       +-+
 *       |c|<-(tr->letterGroups[group])
 *       |0|
 *   *p->|c|<-len+              +-+
 *       |s|<----+              |a|<-(Actual word to be tested)
 *       |0|            *word-> |t|<-*w=word-len+1 (for pre-
rule)
 *       |~|                    |a|<-*w=word       (for post-
rule)
 *       |7|                    |s|
 *       +-+                    +-+
 *
 *     7=RULE_GROUP_END
 *     0=null terminator
 *     pre==1 - pre-rule
 *     pre==0 - post-rule
 */
 char *p; // group counter
 char *w; // word counter
 int len = 0;

 p = tr->letterGroups[group];
 if (p == NULL)
  return -1;

 while (*p != RULE_GROUP_END) {
  if (pre) {
   len = strlen(p);
   w = word - len + 1;
  } else
   w = word;

  // If '~' (no character) is allowed in group, return 0.
  if (*p == '~')
   return 0;
```

```
  //  Check current group
  while ((*p == *w) && (*w != 0)) {
   w++;
   p++;
  }
  if (*p == 0) { // Matched the current group.
   if (pre)
    return len;
   return w - word;
  }

  // No match, so skip the rest of this group.
  while (*p++ != 0)
   ;
 }
 // Not found
 return -1;
}

static int IsLetter(Translator *tr, int letter, int group)
{
 int letter2;

 if (tr->letter_groups[group] != NULL) {
  if (wcschr(tr->letter_groups[group], letter))
   return 1;
  return 0;
 }

 if (group > 7)
  return 0;

 if (tr->letter_bits_offset > 0) {
  if (((letter2 = (letter - tr->letter_bits_offset)) > 0) &&
(letter2 < 0x100))
   letter = letter2;
  else
```

```
  return 0;
 } else if ((letter >= 0xc0) && (letter < N_REMOVE_ACCENT))
  return tr->letter_bits[remove_accent[letter-0xc0]] & (1L <<
group);

 if ((letter >= 0) && (letter < 0x100))
  return tr->letter_bits[letter] & (1L << group);

 return 0;
}

int IsVowel(Translator *tr, int letter)
{
 return IsLetter(tr, letter, LETTERGP_VOWEL2);
}

static int Unpronouncable2(Translator *tr, char *word)
{
 int c;
 int end_flags;
 char ph_buf[N_WORD_PHONEMES];

 ph_buf[0] = 0;
 c = word[-1];
 word[-1] = ' '; // ensure there is a space before the "word"
 end_flags = TranslateRules(tr, word, ph_buf, sizeof(ph_buf),
NULL, FLAG_UNPRON_TEST, NULL);
 word[-1] = c;
 if ((end_flags == 0) || (end_flags & SUFX_UNPRON))
  return 1;
 return 0;
}

int Unpronouncable(Translator *tr, char *word, int posn)
{
 /* Determines whether a word in 'unpronouncable', i.e. whether
it should
```

```
    be spoken as individual letters.

    This function may be language specific. This is a generic
version.
  */

 int c;
 int c1 = 0;
 int vowel_posn = 9;
 int index;
 int count;
 ALPHABET *alphabet;

 utf8_in(&c, word);
 if ((tr->letter_bits_offset > 0) && (c < 0x241)) {
  // Latin characters for a language with a non-latin alphabet
  return 0;  // so we can re-translate the word as English
 }

 if (((alphabet = AlphabetFromChar(c)) != NULL)  &&
(alphabet->offset != tr->letter_bits_offset)) {
  // Character is not in our alphabet
  return 0;
 }

 if (tr->langopts.param[LOPT_UNPRONOUNCABLE] == 1)
  return 0;

 if (((c = *word) == ' ') || (c == 0) || (c == '\''))
  return 0;

 index = 0;
 count = 0;
 for (;;) {
  index += utf8_in(&c, &word[index]);
  if ((c == 0) || (c == ' '))
   break;
```

```
  if ((c == '\'') && ((count > 1) || (posn > 0)))
   break; // "tv'" but not "l'"

  if (count == 0)
   c1 = c;

  if ((c == '\'') && (tr->langopts.param[LOPT_UNPRONOUNCABLE] ==
3)) {
   // don't count apostrophe
  } else
   count++;

  if (IsVowel(tr, c)) {
   vowel_posn = count; // position of the first vowel
   break;
  }

  if ((c != '\'') && !iswalpha(c))
   return 0;
 }

 if ((vowel_posn > 2) && (tr->langopts.param[LOPT_UNPRONOUNCABLE]
== 2)) {
  // Lookup unpronounable rules in *_rules
  return Unpronouncable2(tr, word);
 }

 if (c1 == tr->langopts.param[LOPT_UNPRONOUNCABLE])
  vowel_posn--; // disregard this as the initial letter when
counting

 if (vowel_posn > (tr->langopts.max_initial_consonants+1))
  return 1; // no vowel, or no vowel in first few letters

 return 0;
}
```

```
static int GetVowelStress(Translator *tr, unsigned char
*phonemes, signed char *vowel_stress, int *vowel_count, int
*stressed_syllable, int control)
{
 // control = 1, set stress to 1 for forced unstressed vowels
 unsigned char phcode;
 PHONEME_TAB *ph;
 unsigned char *ph_out = phonemes;
 int count = 1;
 int max_stress = -1;
 int ix;
 int j;
 int stress = -1;
 int primary_posn = 0;

 vowel_stress[0] = STRESS_IS_UNSTRESSED;
 while (((phcode = *phonemes++) != 0) && (count <
(N_WORD_PHONEMES/2)-1)) {
  if ((ph = phoneme_tab[phcode]) == NULL)
   continue;

  if ((ph->type == phSTRESS) && (ph->program == 0)) {
   // stress marker, use this for the following vowel

   if (phcode == phonSTRESS_PREV) {
    // primary stress on preceeding vowel
    j = count - 1;
    while ((j > 0) && (*stressed_syllable == 0) &&
(vowel_stress[j] < STRESS_IS_PRIMARY)) {
     if ((vowel_stress[j] != STRESS_IS_DIMINISHED) &&
(vowel_stress[j] != STRESS_IS_UNSTRESSED)) {
      // don't promote a phoneme which must be unstressed
      vowel_stress[j] = STRESS_IS_PRIMARY;

      if (max_stress < STRESS_IS_PRIMARY) {
       max_stress = STRESS_IS_PRIMARY;
```

```
      primary_posn = j;
     }

     /* reduce any preceding primary stress markers */
     for (ix = 1; ix < j; ix++) {
      if (vowel_stress[ix] == STRESS_IS_PRIMARY)
       vowel_stress[ix] = STRESS_IS_SECONDARY;
     }
     break;
    }
    j--;
   }
  } else {
   if ((ph->std_length < 4) || (*stressed_syllable == 0)) {
    stress = ph->std_length;

    if (stress > max_stress)
     max_stress = stress;
   }
  }
  continue;
 }

 if ((ph->type == phVOWEL) && !(ph->phflags & phNONSYLLABIC)) {
  vowel_stress[count] = (char)stress;
  if ((stress >= STRESS_IS_PRIMARY) && (stress >= max_stress)) {
   primary_posn = count;
   max_stress = stress;
  }

  if ((stress < 0) && (control & 1) && (ph->phflags &
phUNSTRESSED))
   vowel_stress[count] = STRESS_IS_UNSTRESSED; // weak vowel,
must be unstressed

  count++;
  stress = -1;
```

```
  } else if (phcode == phonSYLLABIC) {
   // previous consonant phoneme is syllablic
   vowel_stress[count] = (char)stress;
   if ((stress == 0) && (control & 1))
    vowel_stress[count++] = STRESS_IS_UNSTRESSED; // syllabic
consonant, usually unstressed
  }

  *ph_out++ = phcode;
 }
 vowel_stress[count] = STRESS_IS_UNSTRESSED;

 // has the position of the primary stress been specified by $1,
$2, etc?
 if (*stressed_syllable > 0) {
  if (*stressed_syllable >= count)
   *stressed_syllable = count-1; // the final syllable

  vowel_stress[*stressed_syllable] = STRESS_IS_PRIMARY;
  max_stress = STRESS_IS_PRIMARY;
  primary_posn = *stressed_syllable;
 }

 if (max_stress == STRESS_IS_PRIORITY) {
  // priority stress, replaces any other primary stress marker
  for (ix = 1; ix < count; ix++) {
   if (vowel_stress[ix] == STRESS_IS_PRIMARY) {
    if (tr->langopts.stress_flags & S_PRIORITY_STRESS)
     vowel_stress[ix] = STRESS_IS_UNSTRESSED;
    else
     vowel_stress[ix] = STRESS_IS_SECONDARY;
   }

   if (vowel_stress[ix] == STRESS_IS_PRIORITY) {
    vowel_stress[ix] = STRESS_IS_PRIMARY;
    primary_posn = ix;
   }
```

```
  }
  max_stress = STRESS_IS_PRIMARY;
 }

 return max_stress;
}

static char stress_phonemes[] = {
 phonSTRESS_D, phonSTRESS_U, phonSTRESS_2, phonSTRESS_3,
 phonSTRESS_P, phonSTRESS_P2, phonSTRESS_TONIC
};

void ChangeWordStress(Translator *tr, char *word, int new_stress)
{
 int ix;
 unsigned char *p;
 int max_stress;
 int vowel_count; // num of vowels + 1
 int stressed_syllable = 0; // position of stressed syllable
 unsigned char phonetic[N_WORD_PHONEMES];
 signed char vowel_stress[N_WORD_PHONEMES/2];

 strcpy((char *)phonetic, word);
 max_stress = GetVowelStress(tr, phonetic, vowel_stress,
&vowel_count, &stressed_syllable, 0);

 if (new_stress >= STRESS_IS_PRIMARY) {
  // promote to primary stress
  for (ix = 1; ix < vowel_count; ix++) {
   if (vowel_stress[ix] >= max_stress) {
    vowel_stress[ix] = new_stress;
    break;
   }
  }
 } else {
  // remove primary stress
  for (ix = 1; ix < vowel_count; ix++) {
```

```
   if (vowel_stress[ix] > new_stress) // >= allows for diminished
stress (=1)
    vowel_stress[ix] = new_stress;
  }
 }

 // write out phonemes
 ix = 1;
 p = phonetic;
 while (*p != 0) {
  if ((phoneme_tab[*p]->type == phVOWEL) &&
!(phoneme_tab[*p]->phflags & phNONSYLLABIC)) {
   if ((vowel_stress[ix] == STRESS_IS_DIMINISHED) ||
(vowel_stress[ix] > STRESS_IS_UNSTRESSED))
    *word++ = stress_phonemes[(unsigned char)vowel_stress[ix]];

   ix++;
  }
  *word++ = *p++;
 }

}

void SetWordStress(Translator *tr, char *output, unsigned int
*dictionary_flags, int tonic, int control)
{
 /* Guess stress pattern of word.  This is language specific

    'output' is used for input and output

    'dictionary_flags' has bits 0-3   position of stressed vowel
(if > 0)
                                      or unstressed (if == 7) or
syllables 1 and 2 (if == 6)
                           bits 8...  dictionary flags

    If 'tonic' is set (>= 0), replace highest stress by this
```

```
value.

    control:  bit 0   This is an individual symbol, not a word
              bit 1   Suffix phonemes are still to be added
  */

 unsigned char phcode;
 unsigned char *p;
 PHONEME_TAB *ph;
 int stress;
 int max_stress;
 int max_stress_input; // any stress specified in the input?
 int vowel_count; // num of vowels + 1
 int ix;
 int v;
 int v_stress;
 int stressed_syllable; // position of stressed syllable
 int max_stress_posn;
 char *max_output;
 int final_ph;
 int final_ph2;
 int mnem;
 int opt_length;
 int stressflags;
 int dflags = 0;
 int first_primary;
 int long_vowel;

 signed char vowel_stress[N_WORD_PHONEMES/2];
 char syllable_weight[N_WORD_PHONEMES/2];
 char vowel_length[N_WORD_PHONEMES/2];
 unsigned char phonetic[N_WORD_PHONEMES];

 static char consonant_types[16] = { 0, 0, 0, 1, 1, 1, 1, 1, 1,
1, 0, 0, 0, 0, 0, 0 };

 stressflags = tr->langopts.stress_flags;
```

```
 if (dictionary_flags != NULL)
  dflags = dictionary_flags[0];

 // copy input string into internal buffer
 for (ix = 0; ix < N_WORD_PHONEMES; ix++) {
  phonetic[ix] = output[ix];
  // check for unknown phoneme codes
  if (phonetic[ix] >= n_phoneme_tab)
   phonetic[ix] = phonSCHWA;
  if (phonetic[ix] == 0)
   break;
 }
 if (ix == 0) return;
 final_ph = phonetic[ix-1];
 final_ph2 = phonetic[(ix > 1) ? ix-2 : ix-1];

 max_output = output + (N_WORD_PHONEMES-3); // check for overrun

 // any stress position marked in the xx_list dictionary ?
 bool unstressed_word = false;
 stressed_syllable = dflags & 0x7;
 if (dflags & 0x8) {
  // this indicates a word without a primary stress
  stressed_syllable = dflags & 0x3;
  unstressed_word = true;
 }

 max_stress = max_stress_input = GetVowelStress(tr, phonetic,
vowel_stress, &vowel_count, &stressed_syllable, 1);
 if ((max_stress < 0) && dictionary_flags)
  max_stress = STRESS_IS_DIMINISHED;

 // heavy or light syllables
 ix = 1;
 for (p = phonetic; *p != 0; p++) {
  if ((phoneme_tab[p[0]]->type == phVOWEL) &&
```

```
!(phoneme_tab[p[0]]->phflags & phNONSYLLABIC)) {
   int weight = 0;
   bool lengthened = false;

   if (phoneme_tab[p[1]]->code == phonLENGTHEN)
    lengthened = true;

   if (lengthened || (phoneme_tab[p[0]]->phflags & phLONG)) {
    // long vowel, increase syllable weight
    weight++;
   }
   vowel_length[ix] = weight;

   if (lengthened) p++; // advance over phonLENGTHEN

   if (consonant_types[phoneme_tab[p[1]]->type] &&
((phoneme_tab[p[2]]->type != phVOWEL) ||
(phoneme_tab[p[1]]->phflags & phLONG))) {
    // followed by two consonants, a long consonant, or consonant
and end-of-word
    weight++;
   }
   syllable_weight[ix] = weight;
   ix++;
  }
 }

 switch (tr->langopts.stress_rule)
 {
 case 8:
  // stress on first syllable, unless it is a light syllable
followed by a heavy syllable
  if ((syllable_weight[1] > 0) || (syllable_weight[2] == 0))
   break;
  // fallthrough:
 case 1:
  // stress on second syllable
```

```
  if ((stressed_syllable == 0) && (vowel_count > 2)) {
   stressed_syllable = 2;
   if (max_stress == STRESS_IS_DIMINISHED)
    vowel_stress[stressed_syllable] = STRESS_IS_PRIMARY;
   max_stress = STRESS_IS_PRIMARY;
  }
  break;
 case 10:  // penultimate, but final if only 1 or 2 syllables
  if (stressed_syllable == 0) {
   if (vowel_count < 4) {
    vowel_stress[vowel_count - 1] = STRESS_IS_PRIMARY;
    max_stress = STRESS_IS_PRIMARY;
    break;
   }
  }
  // fallthrough:
 case 2:
  // a language with stress on penultimate vowel

  if (stressed_syllable == 0) {
   // no explicit stress - stress the penultimate vowel
   max_stress = STRESS_IS_PRIMARY;

   if (vowel_count > 2) {
    stressed_syllable = vowel_count - 2;

    if (stressflags & S_FINAL_SPANISH) {
     // LANG=Spanish, stress on last vowel if the word ends in a
consonant other than 'n' or 's'
     if (phoneme_tab[final_ph]->type != phVOWEL) {
      mnem = phoneme_tab[final_ph]->mnemonic;

      if (tr->translator_name == L('a', 'n')) {
       if (((mnem != 's') && (mnem != 'n')) ||
phoneme_tab[final_ph2]->type != phVOWEL)
        stressed_syllable = vowel_count - 1; // stress on last
syllable
```

```
      } else if (tr->translator_name == L('i', 'a')) {
       if ((mnem != 's') || phoneme_tab[final_ph2]->type !=
phVOWEL)
        stressed_syllable = vowel_count - 1; // stress on last
syllable
      } else {
       if ((mnem == 's') && (phoneme_tab[final_ph2]->type ==
phNASAL)) {
        // -ns  stress remains on penultimate syllable
       } else if (((phoneme_tab[final_ph]->type != phNASAL) &&
(mnem != 's')) || (phoneme_tab[final_ph2]->type != phVOWEL))
        stressed_syllable = vowel_count - 1;
      }
     }
    }

    if (stressflags & S_FINAL_LONG) {
     // stress on last syllable if it has a long vowel, but
previous syllable has a short vowel
     if (vowel_length[vowel_count - 1] > vowel_length[vowel_count
- 2])
      stressed_syllable = vowel_count - 1;
    }

    if ((vowel_stress[stressed_syllable] == STRESS_IS_DIMINISHED)
|| (vowel_stress[stressed_syllable] == STRESS_IS_UNSTRESSED)) {
     // but this vowel is explicitly marked as unstressed
     if (stressed_syllable > 1)
      stressed_syllable--;
     else
      stressed_syllable++;
    }
   } else
    stressed_syllable = 1;

   // only set the stress if it's not already marked explicitly
   if (vowel_stress[stressed_syllable] < 0) {
```

```
    // don't stress if next and prev syllables are stressed
    if ((vowel_stress[stressed_syllable-1] < STRESS_IS_PRIMARY)
|| (vowel_stress[stressed_syllable+1] < STRESS_IS_PRIMARY))
     vowel_stress[stressed_syllable] = max_stress;
   }
  }
  break;
 case 3:
  // stress on last vowel
  if (stressed_syllable == 0) {
   // no explicit stress - stress the final vowel
   stressed_syllable = vowel_count - 1;

   while (stressed_syllable > 0) {
    // find the last vowel which is not unstressed
    if (vowel_stress[stressed_syllable] < STRESS_IS_DIMINISHED) {
     vowel_stress[stressed_syllable] = STRESS_IS_PRIMARY;
     break;
    } else
     stressed_syllable--;
   }
   max_stress = STRESS_IS_PRIMARY;
  }
  break;
 case 4: // stress on antipenultimate vowel
  if (stressed_syllable == 0) {
   stressed_syllable = vowel_count - 3;
   if (stressed_syllable < 1)
    stressed_syllable = 1;

   if (max_stress == STRESS_IS_DIMINISHED)
    vowel_stress[stressed_syllable] = STRESS_IS_PRIMARY;
   max_stress = STRESS_IS_PRIMARY;
  }
  break;
 case 5:
  // LANG=Russian
```

```
  if (stressed_syllable == 0) {
   // no explicit stress - guess the stress from the number of
syllables
   static char guess_ru[16] =   { 0, 0, 1, 1, 2, 3, 3, 4, 5, 6,
7, 7, 8, 9, 10, 11 };
   static char guess_ru_v[16] = { 0, 0, 1, 1, 2, 2, 3, 3, 4, 5,
6, 7, 7, 8, 9, 10 }; // for final phoneme is a vowel
   static char guess_ru_t[16] = { 0, 0, 1, 2, 3, 3, 3, 4, 5, 6,
7, 7, 7, 8, 9, 10 }; // for final phoneme is an unvoiced stop

   stressed_syllable = vowel_count - 3;
   if (vowel_count < 16) {
    if (phoneme_tab[final_ph]->type == phVOWEL)
     stressed_syllable = guess_ru_v[vowel_count];
    else if (phoneme_tab[final_ph]->type == phSTOP)
     stressed_syllable = guess_ru_t[vowel_count];
    else
     stressed_syllable = guess_ru[vowel_count];
   }
   vowel_stress[stressed_syllable] = STRESS_IS_PRIMARY;
   max_stress = STRESS_IS_PRIMARY;
  }
  break;
 case 6: // LANG=hi stress on the last heaviest syllable
  if (stressed_syllable == 0) {
   int wt;
   int max_weight = -1;

   // find the heaviest syllable, excluding the final syllable
   for (ix = 1; ix < (vowel_count-1); ix++) {
    if (vowel_stress[ix] < STRESS_IS_DIMINISHED) {
     if ((wt = syllable_weight[ix]) >= max_weight) {
      max_weight = wt;
      stressed_syllable = ix;
     }
    }
   }
```

```
   if ((syllable_weight[vowel_count-1] == 2) &&  (max_weight <
2)) {
    // the only double=heavy syllable is the final syllable, so
stress this
    stressed_syllable = vowel_count-1;
   } else if (max_weight <= 0) {
    // all syllables, exclusing the last, are light. Stress the
first syllable
    stressed_syllable = 1;
   }

   vowel_stress[stressed_syllable] = STRESS_IS_PRIMARY;
   max_stress = STRESS_IS_PRIMARY;
  }
  break;
 case 7: // LANG=tr, the last syllable for any vowel marked
explicitly as unstressed
  if (stressed_syllable == 0) {
   stressed_syllable = vowel_count - 1;
   for (ix = 1; ix < vowel_count; ix++) {
    if (vowel_stress[ix] == STRESS_IS_UNSTRESSED) {
     stressed_syllable = ix-1;
     break;
    }
   }
   vowel_stress[stressed_syllable] = STRESS_IS_PRIMARY;
   max_stress = STRESS_IS_PRIMARY;
  }
  break;
 case 9: // mark all as stressed
  for (ix = 1; ix < vowel_count; ix++) {
   if (vowel_stress[ix] < STRESS_IS_DIMINISHED)
    vowel_stress[ix] = STRESS_IS_PRIMARY;
  }
  break;
 case 12: // LANG=kl (Greenlandic)
```

```
  long_vowel = 0;
  for (ix = 1; ix < vowel_count; ix++) {
   if (vowel_stress[ix] == STRESS_IS_PRIMARY)
    vowel_stress[ix] = STRESS_IS_SECONDARY; // change marked
stress (consonant clusters) to secondary (except the last)

   if (vowel_length[ix] > 0) {
    long_vowel = ix;
    vowel_stress[ix] = STRESS_IS_SECONDARY; // give secondary
stress to all long vowels
   }
  }

  // 'stressed_syllable' gives the last marked stress
  if (stressed_syllable == 0) {
   // no marked stress, choose the last long vowel
   if (long_vowel > 0)
    stressed_syllable = long_vowel;
   else {
    // no long vowels or consonant clusters
    if (vowel_count > 5)
     stressed_syllable = vowel_count - 3; // more than 4
syllables
    else
     stressed_syllable = vowel_count - 1;
   }
  }
  vowel_stress[stressed_syllable] = STRESS_IS_PRIMARY;
  max_stress = STRESS_IS_PRIMARY;
  break;
 case 13: // LANG=ml, 1st unless 1st vowel is short and 2nd is
long
  if (stressed_syllable == 0) {
   stressed_syllable = 1;
   if ((vowel_length[1] == 0) && (vowel_count > 2) &&
(vowel_length[2] > 0))
    stressed_syllable = 2;
```

```
   vowel_stress[stressed_syllable] = STRESS_IS_PRIMARY;
   max_stress = STRESS_IS_PRIMARY;
  }
  break;
 }

 if ((stressflags & S_FINAL_VOWEL_UNSTRESSED) && ((control & 2)
== 0) && (vowel_count > 2) && (max_stress_input <
STRESS_IS_SECONDARY) && (vowel_stress[vowel_count - 1] ==
STRESS_IS_PRIMARY)) {
  // Don't allow stress on a word-final vowel
  // Only do this if there is no suffix phonemes to be added, and
if a stress position was not given explicitly
  if (phoneme_tab[final_ph]->type == phVOWEL) {
   vowel_stress[vowel_count - 1] = STRESS_IS_UNSTRESSED;
   vowel_stress[vowel_count - 2] = STRESS_IS_PRIMARY;
  }
 }

 // now guess the complete stress pattern
 if (max_stress < STRESS_IS_PRIMARY)
  stress = STRESS_IS_PRIMARY; // no primary stress marked, use
for 1st syllable
 else
  stress = STRESS_IS_SECONDARY;

 if (unstressed_word == false) {
  if ((stressflags & S_2_SYL_2) && (vowel_count == 3)) {
   // Two syllable word, if one syllable has primary stress, then
give the other secondary stress
   if (vowel_stress[1] == STRESS_IS_PRIMARY)
    vowel_stress[2] = STRESS_IS_SECONDARY;
   if (vowel_stress[2] == STRESS_IS_PRIMARY)
    vowel_stress[1] = STRESS_IS_SECONDARY;
  }

  if ((stressflags & S_INITIAL_2) && (vowel_stress[1] <
```

```
STRESS_IS_DIMINISHED)) {
   // If there is only one syllable before the primary stress,
give it a secondary stress
   if ((vowel_count > 3) && (vowel_stress[2] >=
STRESS_IS_PRIMARY))
    vowel_stress[1] = STRESS_IS_SECONDARY;
  }
 }

 bool done = false;
 first_primary = 0;
 for (v = 1; v < vowel_count; v++) {
  if (vowel_stress[v] < STRESS_IS_DIMINISHED) {
   if ((stressflags & S_FINAL_NO_2) && (stress <
STRESS_IS_PRIMARY) && (v == vowel_count-1)) {
    // flag: don't give secondary stress to final vowel
   } else if ((stressflags & 0x8000) && (done == false)) {
    vowel_stress[v] = (char)stress;
    done = true;
    stress = STRESS_IS_SECONDARY; // use secondary stress for
remaining syllables
   } else if ((vowel_stress[v-1] <= STRESS_IS_UNSTRESSED) &&
((vowel_stress[v+1] <= STRESS_IS_UNSTRESSED) || ((stress ==
STRESS_IS_PRIMARY) && (vowel_stress[v+1] <=
STRESS_IS_NOT_STRESSED)))) {
    // trochaic: give stress to vowel surrounded by unstressed
vowels

    if ((stress == STRESS_IS_SECONDARY) && (stressflags &
S_NO_AUTO_2))
     continue; // don't use secondary stress

    // don't put secondary stress on a light syllable if the rest
of the word (excluding last syllable) contains a heavy syllable
    if ((v > 1) && (stressflags & S_2_TO_HEAVY) &&
(syllable_weight[v] == 0)) {
     bool skip = false;
```

```
     for (int i = v; i < vowel_count - 1; i++) {
      if (syllable_weight[i] > 0) {
       skip = true;
       break;
      }
     }
     if (skip == true)
      continue;
    }

    if ((v > 1) && (stressflags & S_2_TO_HEAVY) &&
(syllable_weight[v] == 0) && (syllable_weight[v+1] > 0)) {
     // don't put secondary stress on a light syllable which is
followed by a heavy syllable
     continue;
    }

    // should start with secondary stress on the first syllable,
or should it count back from
    // the primary stress and put secondary stress on alternate
syllables?
    vowel_stress[v] = (char)stress;
    done = true;
    stress = STRESS_IS_SECONDARY; // use secondary stress for
remaining syllables
   }
  }

  if (vowel_stress[v] >= STRESS_IS_PRIMARY) {
   if (first_primary == 0)
    first_primary = v;
   else if (stressflags & S_FIRST_PRIMARY) {
    // reduce primary stresses after the first to secondary
    vowel_stress[v] = STRESS_IS_SECONDARY;
   }
  }
 }
```

```
 if ((unstressed_word) && (tonic < 0)) {
  if (vowel_count <= 2)
   tonic = tr->langopts.unstressed_wd1; // monosyllable -
unstressed
  else
   tonic = tr->langopts.unstressed_wd2; // more than one
syllable, used secondary stress as the main stress
 }

 max_stress = STRESS_IS_DIMINISHED;
 max_stress_posn = 0;
 for (v = 1; v < vowel_count; v++) {
  if (vowel_stress[v] >= max_stress) {
   max_stress = vowel_stress[v];
   max_stress_posn = v;
  }
 }

 if (tonic >= 0) {
  // find position of highest stress, and replace it by 'tonic'

  // don't disturb an explicitly set stress by 'unstress-at-end'
flag
  if ((tonic > max_stress) || (max_stress <= STRESS_IS_PRIMARY))
   vowel_stress[max_stress_posn] = (char)tonic;
  max_stress = tonic;
 }

 // produce output phoneme string
 p = phonetic;
 v = 1;

 if (!(control & 1) && ((ph = phoneme_tab[*p]) != NULL)) {
  while ((ph->type == phSTRESS) || (*p == phonEND_WORD)) {
   p++;
   ph = phoneme_tab[p[0]];
```

```
  }

  if ((tr->langopts.vowel_pause & 0x30) && (ph->type == phVOWEL))
{
   // word starts with a vowel

   if ((tr->langopts.vowel_pause & 0x20) && (vowel_stress[1] >=
STRESS_IS_PRIMARY))
    *output++ = phonPAUSE_NOLINK; // not to be replaced by link
   else
    *output++ = phonPAUSE_VSHORT; // break, but no pause
  }
 }

 p = phonetic;
 while (((phcode = *p++) != 0) && (output < max_output)) {
  if ((ph = phoneme_tab[phcode]) == NULL)
   continue;

  if (ph->type == phPAUSE)
   tr->prev_last_stress = 0;
  else if (((ph->type == phVOWEL) && !(ph->phflags &
phNONSYLLABIC)) || (*p == phonSYLLABIC)) {
   // a vowel, or a consonant followed by a syllabic consonant
marker

   v_stress = vowel_stress[v];
   tr->prev_last_stress = v_stress;

   if (v_stress <= STRESS_IS_UNSTRESSED) {
    if ((v > 1) && (max_stress >= 2) && (stressflags &
S_FINAL_DIM) && (v == (vowel_count-1))) {
     // option: mark unstressed final syllable as diminished
     v_stress = STRESS_IS_DIMINISHED;
    } else if ((stressflags & S_NO_DIM) || (v == 1) || (v ==
(vowel_count-1))) {
     // first or last syllable, or option 'don't set diminished
```

```
stress'
     v_stress = STRESS_IS_UNSTRESSED;
    } else if ((v == (vowel_count-2)) &&
(vowel_stress[vowel_count-1] <= STRESS_IS_UNSTRESSED)) {
     // penultimate syllable, followed by an unstressed final
syllable
     v_stress = STRESS_IS_UNSTRESSED;
    } else {
     // unstressed syllable within a word
     if ((vowel_stress[v-1] < STRESS_IS_DIMINISHED) ||
((stressflags & S_MID_DIM) == 0)) {
      v_stress = STRESS_IS_DIMINISHED;
      vowel_stress[v] = v_stress;
     }
    }
   }

   if ((v_stress == STRESS_IS_DIMINISHED) || (v_stress >
STRESS_IS_UNSTRESSED))
    *output++ = stress_phonemes[v_stress]; // mark stress of all
vowels except 1 (unstressed)

   if (vowel_stress[v] > max_stress)
    max_stress = vowel_stress[v];

   if ((*p == phonLENGTHEN) && ((opt_length =
tr->langopts.param[LOPT_IT_LENGTHEN]) & 1)) {
    // remove lengthen indicator from non-stressed syllables
    bool shorten = false;

    if (opt_length & 0x10) {
     // only allow lengthen indicator on the highest stress
syllable in the word
     if (v != max_stress_posn)
      shorten = true;
    } else if (v_stress < STRESS_IS_PRIMARY) {
     // only allow lengthen indicator if stress >=
```

```
STRESS_IS_PRIMARY.
     shorten = true;
    }

    if (shorten)
     p++;
   }
   v++;
  }

  if (phcode != 1)
   *output++ = phcode;
 }

 return;
}

void AppendPhonemes(Translator *tr, char *string, int size, const
char *ph)
{
 /* Add new phoneme string "ph" to "string"
     Keeps count of the number of vowel phonemes in the word, and
whether these
    can be stressed syllables.  These values can be used in
translation rules
  */

 const char *p;
 unsigned char c;
 int length;

 length = strlen(ph) + strlen(string);
 if (length >= size)
  return;

 // any stressable vowel ?
 bool unstress_mark = false;
```

```
 p = ph;
 while ((c = *p++) != 0) {
  if (c >= n_phoneme_tab) continue;

  if (phoneme_tab[c]->type == phSTRESS) {
   if (phoneme_tab[c]->std_length < 4)
    unstress_mark = true;
  } else {
   if (phoneme_tab[c]->type == phVOWEL) {
    if (((phoneme_tab[c]->phflags & phUNSTRESSED) == 0) &&
        (unstress_mark == false)) {
     tr->word_stressed_count++;
    }
    unstress_mark = false;
    tr->word_vowel_count++;
   }
  }
 }

 if (string != NULL)
  strcat(string, ph);
}

static void MatchRule(Translator *tr, char *word[], char
*word_start, int group_length, char *rule, MatchRecord
*match_out, int word_flags, int dict_flags)
{
 /* Checks a specified word against dictionary rules.
     Returns with phoneme code string, or NULL if no match found.

     word (indirect) points to current character group within the
input word
             This is advanced by this procedure as characters are
consumed

     group:  the initial characters used to choose the rules
group
```

```
     rule:  address of dictionary rule data for this character
group

     match_out:  returns best points score

     word_flags:  indicates whether this is a retranslation after
a suffix has been removed
  */

 unsigned char rb;     // current instuction from rule
 unsigned char letter; // current letter from input word, single
byte
 int letter_w;         // current letter, wide character
 int last_letter_w;    // last letter, wide character
 int letter_xbytes;    // number of extra bytes of multibyte
character (num bytes - 1)

 char *pre_ptr;
 char *post_ptr;       // pointer to first character after group

 char *rule_start;     // start of current match template
 char *p;
 int ix;

 int match_type;       // left, right, or consume
 int failed;
 int unpron_ignore;
 int consumed;         // number of letters consumed from input
 int syllable_count;
 int vowel;
 int letter_group;
 int distance_right;
 int distance_left;
 int lg_pts;
 int n_bytes;
 int add_points;
```

```
int command;
bool check_atstart;
unsigned int *flags;

MatchRecord match;
static MatchRecord best;

int total_consumed; // letters consumed for best match

unsigned char condition_num;
char *common_phonemes; // common to a group of entries
char *group_chars;
char word_buf[N_WORD_BYTES];

group_chars = *word;

if (rule == NULL) {
 match_out->points = 0;
 (*word)++;
 return;
}

total_consumed = 0;
common_phonemes = NULL;

best.points = 0;
best.phonemes = "";
best.end_type = 0;
best.del_fwd = NULL;

// search through dictionary rules
while (rule[0] != RULE_GROUP_END) {
 unpron_ignore = word_flags & FLAG_UNPRON_TEST;
 match_type = 0;
 consumed = 0;
 letter_w = 0;
 distance_right = -6; // used to reduce points for matches
```

```
further away the current letter
  distance_left = -2;
  check_atstart = false;

  match.points = 1;
  match.end_type = 0;
  match.del_fwd = NULL;

  pre_ptr = *word;
  post_ptr = *word + group_length;

  // work through next rule until end, or until no-match proved
  rule_start = rule;

  failed = 0;
  while (!failed) {
   rb = *rule++;

   if (rb <= RULE_LINENUM) {
    switch (rb)
    {
    case 0: // no phoneme string for this rule, use previous
common rule
     if (common_phonemes != NULL) {
      match.phonemes = common_phonemes;
      while (((rb = *match.phonemes++) != 0) && (rb !=
RULE_PHONEMES)) {
       if (rb == RULE_CONDITION)
        match.phonemes++; // skip over condition number
       if (rb == RULE_LINENUM)
        match.phonemes += 2; // skip over line number
      }
     } else
      match.phonemes = "";
     rule--; // so we are still pointing at the 0
     failed = 2; // matched OK
     break;
```

```
    case RULE_PRE_ATSTART: // pre rule with implied 'start of
word'
     check_atstart = true;
     unpron_ignore = 0;
     match_type = RULE_PRE;
     break;
    case RULE_PRE:
     match_type = RULE_PRE;
     if (word_flags & FLAG_UNPRON_TEST) {
      // checking the start of the word for unpronouncable
character sequences, only
      // consider rules which explicitly match the start of a
word
      // Note: Those rules now use RULE_PRE_ATSTART
      failed = 1;
     }
     break;
    case RULE_POST:
     match_type = RULE_POST;
     break;
    case RULE_PHONEMES:
     match.phonemes = rule;
     failed = 2; // matched OK
     break;
    case RULE_PH_COMMON:
     common_phonemes = rule;
     break;
    case RULE_CONDITION:
     // conditional rule, next byte gives condition number
     condition_num = *rule++;

     if (condition_num >= 32) {
      // allow the rule only if the condition number is NOT set
      if ((tr->dict_condition & (1L << (condition_num-32))) != 0)
       failed = 1;
     } else {
      // allow the rule only if the condition number is set
```

```
        if ((tr->dict_condition & (1L << condition_num)) == 0)
         failed = 1;
       }

       if (!failed)
        match.points++; // add one point for a matched conditional
rule
       break;
      case RULE_LINENUM:
       rule += 2;
       break;
      }
      continue;
     }

     add_points = 0;

     switch (match_type)
     {
     case 0:
      // match and consume this letter
      letter = *post_ptr++;

      if ((letter == rb) || ((letter == (unsigned char)REPLACED_E)
&& (rb == 'e'))) {
       if ((letter & 0xc0) != 0x80)
        add_points = 21; // don't add point for non-initial UTF-8
bytes
       consumed++;
      } else
       failed = 1;
      break;
     case RULE_POST:
      // continue moving forwards
      distance_right += 6;
      if (distance_right > 18)
       distance_right = 19;
```

```
    last_letter_w = letter_w;
    letter_xbytes = utf8_in(&letter_w, post_ptr)-1;
    letter = *post_ptr++;

    switch (rb)
    {
    case RULE_LETTERGP:
     letter_group = LetterGroupNo(rule++);
     if (IsLetter(tr, letter_w, letter_group)) {
      lg_pts = 20;
      if (letter_group == 2)
       lg_pts = 19; // fewer points for C, general consonant
      add_points = (lg_pts-distance_right);
      post_ptr += letter_xbytes;
     } else
      failed = 1;
     break;
    case RULE_LETTERGP2: // match against a list of utf-8 strings
     letter_group = LetterGroupNo(rule++);
     if ((n_bytes = IsLetterGroup(tr, post_ptr-1, letter_group,
0)) >= 0) {
      add_points = (20-distance_right);
      if (n_bytes >= 0) // move pointer, if group was found
       post_ptr += (n_bytes-1);
     } else
      failed = 1;
     break;
    case RULE_NOTVOWEL:
     if (IsLetter(tr, letter_w, 0) || ((letter_w == ' ') &&
(word_flags & FLAG_SUFFIX_VOWEL)))
      failed = 1;
     else {
      add_points = (20-distance_right);
      post_ptr += letter_xbytes;
     }
     break;
    case RULE_DIGIT:
```

```
    if (IsDigit(letter_w)) {
     add_points = (20-distance_right);
     post_ptr += letter_xbytes;
    } else if (tr->langopts.tone_numbers) {
     // also match if there is no digit
     add_points = (20-distance_right);
     post_ptr--;
    } else
     failed = 1;
    break;
   case RULE_NONALPHA:
    if (!iswalpha(letter_w)) {
     add_points = (21-distance_right);
     post_ptr += letter_xbytes;
    } else
     failed = 1;
    break;
   case RULE_DOUBLE:
    if (letter_w == last_letter_w)
     add_points = (21-distance_right);
    else
     failed = 1;
    break;
   case RULE_DOLLAR:
    command = *rule++;
    if (command == DOLLAR_UNPR)
     match.end_type = SUFX_UNPRON; // $unpron
    else if (command == DOLLAR_NOPREFIX) { // $noprefix
     if (word_flags & FLAG_PREFIX_REMOVED)
      failed = 1; // a prefix has been removed
     else
      add_points = 1;
    } else if ((command & 0xf0) == 0x10) {
     // $w_alt
     if (dict_flags & (1 << (BITNUM_FLAG_ALT + (command &
0xf))))
      add_points = 23;
```

```
      else
       failed = 1;
     } else if (((command & 0xf0) == 0x20) || (command ==
DOLLAR_LIST)) {
      // $list or $p_alt
      // make a copy of the word up to the post-match characters
      ix = *word - word_start + consumed + group_length + 1;
      memcpy(word_buf, word_start-1, ix);
      word_buf[ix] = ' ';
      word_buf[ix+1] = 0;
      LookupFlags(tr, &word_buf[1], &flags);

      if ((command == DOLLAR_LIST) && (flags[0] & FLAG_FOUND) &&
!(flags[1] & FLAG_ONLY))
       add_points = 23;
      else if (flags[0] & (1 << (BITNUM_FLAG_ALT + (command &
0xf))))
       add_points = 23;
      else
       failed = 1;
     }
     break;
    case '-':
     if ((letter == '-') || ((letter == ' ') && (word_flags &
FLAG_HYPHEN_AFTER)))
      add_points = (22-distance_right); // one point more than
match against space
     else
      failed = 1;
     break;
    case RULE_SYLLABLE:
    {
     // more than specified number of vowel letters to the right
     char *p = post_ptr + letter_xbytes;
     int vowel_count = 0;

     syllable_count = 1;
```

```
      while (*rule == RULE_SYLLABLE) {
       rule++;
       syllable_count += 1; // number of syllables to match
      }
      vowel = 0;
      while (letter_w != RULE_SPACE) {
       if ((vowel == 0) && IsLetter(tr, letter_w,
LETTERGP_VOWEL2)) {
        // this is counting vowels which are separated by non-
vowel letters
        vowel_count++;
       }
       vowel = IsLetter(tr, letter_w, LETTERGP_VOWEL2);
       p += utf8_in(&letter_w, p);
      }
      if (syllable_count <= vowel_count)
       add_points = (18+syllable_count-distance_right);
      else
       failed = 1;
     }
      break;
     case RULE_NOVOWELS:
     {
      char *p = post_ptr + letter_xbytes;
      while (letter_w != RULE_SPACE) {
       if (IsLetter(tr, letter_w, LETTERGP_VOWEL2)) {
        failed = 1;
        break;
       }
       p += utf8_in(&letter_w, p);
      }
      if (!failed)
       add_points = (19-distance_right);
     }
      break;
     case RULE_SKIPCHARS:
     {
```

```
     // '(Jxy'  means 'skip characters until xy'
     char *p = post_ptr - 1; // to allow empty jump (without
letter between), go one back
     char *p2 = p;  // pointer to the previous character in the
word
     int rule_w;  // first wide character of skip rule
     utf8_in(&rule_w, rule);
     int g_bytes = -1; // bytes of successfully found character
group
     while ((letter_w != rule_w) && (letter_w != RULE_SPACE) &&
(letter_w != 0) && (g_bytes == -1)) {
      if (rule_w == RULE_LETTERGP2)
       g_bytes = IsLetterGroup(tr, p, LetterGroupNo(rule + 1),
0);
      p2 = p;
      p += utf8_in(&letter_w, p);
     }
     if ((letter_w == rule_w) || (g_bytes >= 0))
      post_ptr = p2;
    }
     break;
    case RULE_INC_SCORE:
     add_points = 20; // force an increase in points
     break;
    case RULE_DEC_SCORE:
     add_points = -20; // force an decrease in points
     break;
    case RULE_DEL_FWD:
     // find the next 'e' in the word and replace by 'E'
     for (p = *word + group_length; p < post_ptr; p++) {
      if (*p == 'e') {
       match.del_fwd = p;
       break;
      }
     }
     break;
    case RULE_ENDING:
```

```
    {
     int end_type;
     // next 3 bytes are a (non-zero) ending type. 2 bytes of
flags + suffix length
     end_type = (rule[0] << 16) + ((rule[1] & 0x7f) << 8) +
(rule[2] & 0x7f);

     if ((tr->word_vowel_count == 0) && !(end_type & SUFX_P) &&
(tr->langopts.param[LOPT_SUFFIX] & 1))
      failed = 1; // don't match a suffix rule if there are no
previous syllables (needed for lang=tr).
     else {
      match.end_type = end_type;
      rule += 3;
     }
    }
     break;
    case RULE_NO_SUFFIX:
     if (word_flags & FLAG_SUFFIX_REMOVED)
      failed = 1; // a suffix has been removed
     else
      add_points = 1;
     break;
    default:
     if (letter == rb) {
      if ((letter & 0xc0) != 0x80) {
       // not for non-initial UTF-8 bytes
       add_points = (21-distance_right);
      }
     } else
      failed = 1;
     break;
    }
    break;
   case RULE_PRE:
    // match backwards from start of current group
    distance_left += 2;
```

```
    if (distance_left > 18)
     distance_left = 19;

    utf8_in(&last_letter_w, pre_ptr);
    pre_ptr--;
    letter_xbytes = utf8_in2(&letter_w, pre_ptr, 1)-1;
    letter = *pre_ptr;

    switch (rb)
    {
    case RULE_LETTERGP:
     letter_group = LetterGroupNo(rule++);
     if (IsLetter(tr, letter_w, letter_group)) {
      lg_pts = 20;
      if (letter_group == 2)
       lg_pts = 19; // fewer points for C, general consonant
      add_points = (lg_pts-distance_left);
      pre_ptr -= letter_xbytes;
     } else
      failed = 1;
     break;
    case RULE_LETTERGP2: // match against a list of utf-8 strings
     letter_group = LetterGroupNo(rule++);
     if ((n_bytes = IsLetterGroup(tr, pre_ptr, letter_group, 1))
>= 0) {
      add_points = (20-distance_right);
      if (n_bytes >= 0)  // move pointer, if group was found
       pre_ptr -= (n_bytes-1);
     } else
      failed = 1;
     break;
    case RULE_NOTVOWEL:
     if (!IsLetter(tr, letter_w, 0)) {
      add_points = (20-distance_left);
      pre_ptr -= letter_xbytes;
     } else
      failed = 1;
```

```
        break;
    case RULE_DOUBLE:
        if (letter_w == last_letter_w)
            add_points = (21-distance_left);
        else
            failed = 1;
        break;
    case RULE_DIGIT:
        if (IsDigit(letter_w)) {
            add_points = (21-distance_left);
            pre_ptr -= letter_xbytes;
        } else
            failed = 1;
        break;
    case RULE_NONALPHA:
        if (!iswalpha(letter_w)) {
            add_points = (21-distance_right);
            pre_ptr -= letter_xbytes;
        } else
            failed = 1;
        break;
    case RULE_DOLLAR:
        command = *rule++;
        if ((command == DOLLAR_LIST) || ((command & 0xf0) == 0x20))
{
            // $list or $p_alt
            // make a copy of the word up to the current character
            ix = *word - word_start + 1;
            memcpy(word_buf, word_start-1, ix);
            word_buf[ix] = ' ';
            word_buf[ix+1] = 0;
            LookupFlags(tr, &word_buf[1], &flags);

            if ((command == DOLLAR_LIST) && (flags[0] & FLAG_FOUND) &&
!(flags[1] & FLAG_ONLY))
                add_points = 23;
            else if (flags[0] & (1 << (BITNUM_FLAG_ALT + (command &
```

```
0xf))))
       add_points = 23;
      else
       failed = 1;
     }
     break;
    case RULE_SYLLABLE:
     // more than specified number of vowels to the left
     syllable_count = 1;
     while (*rule == RULE_SYLLABLE) {
      rule++;
      syllable_count++; // number of syllables to match
     }
     if (syllable_count <= tr->word_vowel_count)
      add_points = (18+syllable_count-distance_left);
     else
      failed = 1;
     break;
    case RULE_STRESSED:
     if (tr->word_stressed_count > 0)
      add_points = 19;
     else
      failed = 1;
     break;
    case RULE_NOVOWELS:
    {
     char *p = pre_ptr - letter_xbytes - 1;
     while (letter_w != RULE_SPACE) {
      if (IsLetter(tr, letter_w, LETTERGP_VOWEL2)) {
       failed = 1;
       break;
      }
      p -= utf8_in2(&letter_w, p, 1);
     }
     if (!failed)
      add_points = 3;
    }
```

```
 break;
case RULE_IFVERB:
 if (tr->expect_verb)
  add_points = 1;
 else
  failed = 1;
 break;
case RULE_CAPITAL:
 if (word_flags & FLAG_FIRST_UPPER)
  add_points = 1;
 else
  failed = 1;
 break;
case '.':
 // dot in pre- section, match on any dot before this point
in the word
 for (p = pre_ptr; *p != ' '; p--) {
  if (*p == '.') {
   add_points = 50;
   break;
  }
 }
 if (*p == ' ')
  failed = 1;
 break;
case '-':
 if ((letter == '-') || ((letter == ' ') && (word_flags &
FLAG_HYPHEN)))
  add_points = (22-distance_right); // one point more than
match against space
 else
  failed = 1;
 break;

case RULE_SKIPCHARS: {
 // 'xyJ)'  means 'skip characters backwards until xy'
 char *p = pre_ptr + 1; // to allow empty jump (without
```

```
letter between), go one forward
     char *p2 = p;  // pointer to previous character in word
     int g_bytes = -1; // bytes of successfully found character
group

     while ((*p != *rule) && (*p != RULE_SPACE) && (*p != 0) &&
(g_bytes == -1)) {
      p2 = p;
      p--;
      if (*rule == RULE_LETTERGP2)
       g_bytes = IsLetterGroup(tr, p2, LetterGroupNo(rule + 1),
1);
     }

     // if succeed, set pre_ptr to next character after 'xy' and
remaining
     // 'xy' part is checked as usual in following cycles of PRE
rule characters
     if (*p == *rule)
      pre_ptr = p2;
     if (g_bytes >= 0)
      pre_ptr = p2 + 1;

    }
     break;

    default:
     if (letter == rb) {
      if (letter == RULE_SPACE)
       add_points = 4;
      else if ((letter & 0xc0) != 0x80) {
       // not for non-initial UTF-8 bytes
       add_points = (21-distance_left);
      }
     } else
      failed = 1;
     break;
```

```
    }
    break;
   }

   if (failed == 0)
    match.points += add_points;
  }

  if ((failed == 2) && (unpron_ignore == 0)) {
   // do we also need to check for 'start of word' ?
   if ((check_atstart == false) || (pre_ptr[-1] == ' ')) {
    if (check_atstart)
     match.points += 4;

    // matched OK, is this better than the last best match ?
    if (match.points >= best.points) {
     memcpy(&best, &match, sizeof(match));
     total_consumed = consumed;
    }

    if ((option_phonemes & espeakPHONEMES_TRACE) && (match.points
> 0) && ((word_flags & FLAG_NO_TRACE) == 0)) {
     // show each rule that matches, and it's points score
     int pts;
     char decoded_phonemes[80];

     pts = match.points;
     if (group_length > 1)
      pts += 35; // to account for an extra letter matching
     DecodePhonemes(match.phonemes, decoded_phonemes);
     fprintf(f_trans, "%3d\t%s [%s]\n", pts,
DecodeRule(group_chars, group_length, rule_start, word_flags),
decoded_phonemes);
    }
   }
  }
```

```
  // skip phoneme string to reach start of next template
  while (*rule++ != 0) ;
 }

 // advance input data pointer
 total_consumed += group_length;
 if (total_consumed == 0)
  total_consumed = 1; // always advance over 1st letter

 if (best.points == 0)
  best.phonemes = "";
 memcpy(match_out, &best, sizeof(MatchRecord));
}

int TranslateRules(Translator *tr, char *p_start, char *phonemes,
int ph_size, char *end_phonemes, int word_flags, unsigned int
*dict_flags)
{
 /* Translate a word bounded by space characters
    Append the result to 'phonemes' and any standard
prefix/suffix in 'end_phonemes' */

 unsigned char c, c2;
 unsigned int c12;
 int wc = 0;
 int wc_bytes;
 char *p2;           // copy of p for use in double letter chain
match
 int found;
 int g;              // group chain number
 int g1;             // first group for this letter
 int n;
 int letter;
 int any_alpha = 0;
 int ix;
 unsigned int digit_count = 0;
 char *p;
```

```
ALPHABET *alphabet;
int dict_flags0 = 0;
MatchRecord match1;
MatchRecord match2;
char ph_buf[40];
char word_copy[N_WORD_BYTES];
static const char str_pause[2] = { phonPAUSE_NOLINK, 0 };

if (tr->data_dictrules == NULL)
 return 0;

if (dict_flags != NULL)
 dict_flags0 = dict_flags[0];

for (ix = 0; ix < (N_WORD_BYTES-1);) {
 c = p_start[ix];
 word_copy[ix++] = c;
 if (c == 0)
  break;
}
word_copy[ix] = 0;

if ((option_phonemes & espeakPHONEMES_TRACE) && ((word_flags &
FLAG_NO_TRACE) == 0)) {
 char wordbuf[120];
 unsigned int ix;

 for (ix = 0; ((c = p_start[ix]) != ' ') && (c != 0) && (ix <
(sizeof(wordbuf)-1)); ix++)
  wordbuf[ix] = c;
 wordbuf[ix] = 0;
 if (word_flags & FLAG_UNPRON_TEST)
  fprintf(f_trans, "Unpronouncable? '%s'\n", wordbuf);
 else
  fprintf(f_trans, "Translate '%s'\n", wordbuf);
}
```

```
 p = p_start;
 tr->word_vowel_count = 0;
 tr->word_stressed_count = 0;

 if (end_phonemes != NULL)
  end_phonemes[0] = 0;

 while (((c = *p) != ' ') && (c != 0)) {
  wc_bytes = utf8_in(&wc, p);
  if (IsAlpha(wc))
   any_alpha++;

  n = tr->groups2_count[c];
  if (IsDigit(wc) && ((tr->langopts.tone_numbers == 0) ||
!any_alpha)) {
   // lookup the number in *_list not *_rules
   char string[8];
   char buf[40];
   string[0] = '_';
   memcpy(&string[1], p, wc_bytes);
   string[1+wc_bytes] = 0;
   Lookup(tr, string, buf);
   if (++digit_count >= 2) {
    strcat(buf, str_pause);
    digit_count = 0;
   }
   AppendPhonemes(tr, phonemes, ph_size, buf);
   p += wc_bytes;
   continue;
  } else {
   digit_count = 0;
   found = 0;

   if (((ix = wc - tr->letter_bits_offset) >= 0) && (ix < 128)) {
    if (tr->groups3[ix] != NULL) {
     MatchRule(tr, &p, p_start, wc_bytes, tr->groups3[ix],
&match1, word_flags, dict_flags0);
```

```
     found = 1;
    }
   }

   if (!found && (n > 0)) {
    // there are some 2 byte chains for this initial letter
    c2 = p[1];
    c12 = c + (c2 << 8); // 2 characters

    g1 = tr->groups2_start[c];
    for (g = g1; g < (g1+n); g++) {
     if (tr->groups2_name[g] == c12) {
      found = 1;

      p2 = p;
      MatchRule(tr, &p2, p_start, 2, tr->groups2[g], &match2,
word_flags, dict_flags0);
      if (match2.points > 0)
       match2.points += 35; // to acount for 2 letters matching

      // now see whether single letter chain gives a better match
?
      MatchRule(tr, &p, p_start, 1, tr->groups1[c], &match1,
word_flags, dict_flags0);

      if (match2.points >= match1.points) {
       // use match from the 2-letter group
       memcpy(&match1, &match2, sizeof(MatchRecord));
       p = p2;
      }
     }
    }
   }

   if (!found) {
    // alphabetic, single letter chain
    if (tr->groups1[c] != NULL)
```

```
     MatchRule(tr, &p, p_start, 1, tr->groups1[c], &match1,
word_flags, dict_flags0);
    else {
     // no group for this letter, use default group
     MatchRule(tr, &p, p_start, 0, tr->groups1[0], &match1,
word_flags, dict_flags0);

     if ((match1.points == 0) && ((option_sayas & 0x10) == 0)) {
      n = utf8_in(&letter, p-1)-1;

      if (tr->letter_bits_offset > 0) {
       // not a Latin alphabet, switch to the default Latin
alphabet language
       if ((letter <= 0x241) && iswalpha(letter)) {
        sprintf(phonemes, "%cen", phonSWITCH);
        return 0;
       }
      }

      // is it a bracket ?
      if (letter == 0xe000+'(') {
       if (pre_pause < tr->langopts.param2[LOPT_BRACKET_PAUSE])
        pre_pause = tr->langopts.param2[LOPT_BRACKET_PAUSE]; // a
bracket, aleady spoken by AnnouncePunctuation()
      }
      if (IsBracket(letter)) {
       if (pre_pause < tr->langopts.param[LOPT_BRACKET_PAUSE])
        pre_pause = tr->langopts.param[LOPT_BRACKET_PAUSE];
      }

      // no match, try removing the accent and re-translating the
word
      if ((letter >= 0xc0) && (letter < N_REMOVE_ACCENT) && ((ix
= remove_accent[letter-0xc0]) != 0)) {
       // within range of the remove_accent table
       if ((p[-2] != ' ') || (p[n] != ' ')) {
        // not the only letter in the word
```

```
        p2 = p-1;
        p[-1] = ix;
        while ((p[0] = p[n]) != ' ')  p++;
        while (n-- > 0) *p++ = ' '; // replacement character must
be no longer than original

        if (tr->langopts.param[LOPT_DIERESES] &&
(lookupwchar(diereses_list, letter) > 0)) {
         // vowel with dieresis, replace and continue from this
point
         p = p2;
         continue;
        }

        phonemes[0] = 0; // delete any phonemes which have been
produced so far
        p = p_start;
        tr->word_vowel_count = 0;
        tr->word_stressed_count = 0;
        continue; // start again at the beginning of the word
       }
      }

      if (((alphabet = AlphabetFromChar(letter)) != NULL)  &&
(alphabet->offset != tr->letter_bits_offset)) {
       if (tr->langopts.alt_alphabet == alphabet->offset) {
        sprintf(phonemes, "%c%s", phonSWITCH,
WordToString2(tr->langopts.alt_alphabet_lang));
        return 0;
       }
       if (alphabet->flags & AL_WORDS) {
        // switch to the nominated language for this alphabet
        sprintf(phonemes, "%c%s", phonSWITCH,
WordToString2(alphabet->language));
        return 0;
       }
      }
```

```
     }
    }

    if (match1.points == 0) {
     if ((wc >= 0x300) && (wc <= 0x36f)) {
      // combining accent inside a word, ignore
     } else if (IsAlpha(wc)) {
      if ((any_alpha > 1) || (p[wc_bytes-1] > ' ')) {
       // an unrecognised character in a word, abort and then
spell the word
       phonemes[0] = 0;
       if (dict_flags != NULL)
        dict_flags[0] |= FLAG_SPELLWORD;
       break;
      }
     } else {
      LookupLetter(tr, wc, -1, ph_buf, 0);
      if (ph_buf[0]) {
       match1.phonemes = ph_buf;
       match1.points = 1;
      }
     }
     p += (wc_bytes-1);
    } else
     tr->phonemes_repeat_count = 0;
   }
  }

  if (match1.phonemes == NULL)
   match1.phonemes = "";

  if (match1.points > 0) {
   if (word_flags & FLAG_UNPRON_TEST)
    return match1.end_type | 1;

   if ((match1.phonemes[0] == phonSWITCH) && ((word_flags &
FLAG_DONT_SWITCH_TRANSLATOR) == 0)) {
```

```
    // an instruction to switch language, return immediately so
we can re-translate
    strcpy(phonemes, match1.phonemes);
    return 0;
   }

   if ((option_phonemes & espeakPHONEMES_TRACE) && ((word_flags &
FLAG_NO_TRACE) == 0))
    fprintf(f_trans, "\n");

   match1.end_type &= ~SUFX_UNPRON;

   if ((match1.end_type != 0) && (end_phonemes != NULL)) {
    // a standard ending has been found, re-translate the word
without it
    if ((match1.end_type & SUFX_P) && (word_flags &
FLAG_NO_PREFIX)) {
     // ignore the match on a prefix
    } else {
     if ((match1.end_type & SUFX_P) && ((match1.end_type & 0x7f)
== 0)) {
      // no prefix length specified
      match1.end_type |= p - p_start;
     }
     strcpy(end_phonemes, match1.phonemes);
     memcpy(p_start, word_copy, strlen(word_copy));
     return match1.end_type;
    }
   }
   if (match1.del_fwd != NULL)
    *match1.del_fwd = REPLACED_E;
   AppendPhonemes(tr, phonemes, ph_size, match1.phonemes);
  }
 }

 memcpy(p_start, word_copy, strlen(word_copy));
```

```
 return 0;
}

void ApplySpecialAttribute2(Translator *tr, char *phonemes, int
dict_flags)
{
 // apply after the translation is complete

 int ix;
 int len;
 char *p;

 len = strlen(phonemes);

 if (tr->langopts.param[LOPT_ALT] & 2) {
  for (ix = 0; ix < (len-1); ix++) {
   if (phonemes[ix] == phonSTRESS_P) {
    p = &phonemes[ix+1];
    if ((dict_flags & FLAG_ALT2_TRANS) != 0) {
     if (*p == PhonemeCode('E'))
      *p = PhonemeCode('e');
     if (*p == PhonemeCode('O'))
      *p = PhonemeCode('o');
    } else {
     if (*p == PhonemeCode('e'))
      *p = PhonemeCode('E');
     if (*p == PhonemeCode('o'))
      *p = PhonemeCode('O');
    }
    break;
   }
  }
 }
}

int TransposeAlphabet(Translator *tr, char *text)
{
```

```
 // transpose cyrillic alphabet (for example) into ascii (single
byte) character codes
 // return: number of bytes, bit 6: 1=used compression

 int c;
 int c2;
 int ix;
 int offset;
 int min;
 int max;
 const char *map;
 char *p = text;
 char *p2;
 bool all_alpha = true;
 int bits;
 int acc;
 int pairs_start;
 const short *pairs_list;
 int bufix;
 char buf[N_WORD_BYTES+1];

 offset = tr->transpose_min - 1;
 min = tr->transpose_min;
 max = tr->transpose_max;
 map = tr->transpose_map;

 pairs_start = max - min + 2;

 bufix = 0;
 do {
  p += utf8_in(&c, p);
  if (c != 0) {
   if ((c >= min) && (c <= max)) {
    if (map == NULL)
     buf[bufix++] = c - offset;
    else {
     // get the code from the transpose map
```

```
      if (map[c - min] > 0)
       buf[bufix++] = map[c - min];
      else {
       all_alpha = false;
       break;
      }
     }
    } else {
     all_alpha = false;
     break;
    }
   }
  } while ((c != 0) && (bufix < N_WORD_BYTES));
  buf[bufix] = 0;

  if (all_alpha) {
   // compress to 6 bits per character
   acc = 0;
   bits = 0;

   p = buf;
   p2 = buf;
   while ((c = *p++) != 0) {
    if ((pairs_list = tr->frequent_pairs) != NULL) {
     c2 = c + (*p << 8);
     for (ix = 0; c2 >= pairs_list[ix]; ix++) {
      if (c2 == pairs_list[ix]) {
       // found an encoding for a 2-character pair
       c = ix + pairs_start; // 2-character codes start after the
single letter codes
       p++;
       break;
      }
     }
    }
    acc = (acc << 6) + (c & 0x3f);
    bits += 6;
```

```
   if (bits >= 8) {
    bits -= 8;
    *p2++ = (acc >> bits);
   }
  }
  if (bits > 0)
   *p2++ = (acc << (8-bits));
  *p2 = 0;
  ix = p2 - buf;
  memcpy(text, buf, ix);
  return ix | 0x40; // bit 6 indicates compressed characters
 }
 return strlen(text);
}

   Returns NULL if no match, else returns 'word_end'

    word   zero terminated word to match
    word2  pointer to next word(s) in the input text (terminated
by space)

    flags:  returns dictionary flags which are associated with a
matched word

    end_flags:  indicates whether this is a retranslation after
removing a suffix

static const char *LookupDict2(Translator *tr, const char *word,
const char *word2,
                               char *phonetic, unsigned int
*flags, int end_flags, WORD_TAB *wtab)
{
 char *p;
 char *next;
 int hash;
 int phoneme_len;
```

```
 int wlen;
 unsigned char flag;
 unsigned int dictionary_flags;
 unsigned int dictionary_flags2;
 int condition_failed = 0;
 int n_chars;
 int no_phonemes;
 int skipwords;
 int ix;
 int c;
 const char *word_end;
 const char *word1;
 int wflags = 0;
 int lookup_symbol;
 char word_buf[N_WORD_BYTES+1];
 char dict_flags_buf[80];

 if (wtab != NULL)
  wflags = wtab->flags;

 lookup_symbol = flags[1] & FLAG_LOOKUP_SYMBOL;
 word1 = word;
 if (tr->transpose_min > 0) {
  strncpy0(word_buf, word, N_WORD_BYTES);
  wlen = TransposeAlphabet(tr, word_buf); // bit 6 indicates
compressed characters
  word = word_buf;
 } else
  wlen = strlen(word);

 hash = HashDictionary(word);
 p = tr->dict_hashtab[hash];

 if (p == NULL) {
  if (flags != NULL)
   *flags = 0;
  return 0;
```

```
 }

 // Find the first entry in the list for this hash value which
matches.
 // This corresponds to the last matching entry in the *_list
file.

 while (*p != 0) {
  next = p + (p[0] & 0xff);

  if (((p[1] & 0x7f) != wlen) || (memcmp(word, &p[2], wlen &
0x3f) != 0)) {
   // bit 6 of wlen indicates whether the word has been
compressed; so we need to match on this also.
   p = next;
   continue;
  }

  // found matching entry. Decode the phonetic string
  word_end = word2;

  dictionary_flags = 0;
  dictionary_flags2 = 0;
  no_phonemes = p[1] & 0x80;

  p += ((p[1] & 0x3f) + 2);

  if (no_phonemes) {
   phonetic[0] = 0;
   phoneme_len = 0;
  } else {
   strcpy(phonetic, p);
   phoneme_len = strlen(p);
   p += (phoneme_len + 1);
  }

  while (p < next) {
```

```
  // examine the flags which follow the phoneme string

  flag = *p++;
  if (flag >= 100) {
   // conditional rule
   if (flag >= 132) {
    // fail if this condition is set
    if ((tr->dict_condition & (1 << (flag-132))) != 0)
     condition_failed = 1;
   } else {
    // allow only if this condition is set
    if ((tr->dict_condition & (1 << (flag-100))) == 0)
     condition_failed = 1;
   }
  } else if (flag > 80) {
   // flags 81 to 90  match more than one word
   // This comes after the other flags
   n_chars = next - p;
   skipwords = flag - 80;

   // don't use the contraction if any of the words are
emphasized
   //  or has an embedded command, such as MARK
   if (wtab != NULL) {
    for (ix = 0; ix <= skipwords; ix++) {
     if (wtab[ix].flags & FLAG_EMPHASIZED2)
      condition_failed = 1;
    }
   }

   if (memcmp(word2, p, n_chars) != 0)
    condition_failed = 1;

   if (condition_failed) {
    p = next;
    break;
   }
```

```
    dictionary_flags |= FLAG_SKIPWORDS;
    dictionary_skipwords = skipwords;
    p = next;
    word_end = word2 + n_chars;
   } else if (flag > 64) {
    // stressed syllable information, put in bits 0-3
    dictionary_flags = (dictionary_flags & ~0xf) | (flag & 0xf);
    if ((flag & 0xc) == 0xc)
     dictionary_flags |= FLAG_STRESS_END;
   } else if (flag >= 32)
    dictionary_flags2 |= (1L << (flag-32));
   else
    dictionary_flags |= (1L << flag);
  }

  if (condition_failed) {
   condition_failed = 0;
   continue;
  }

  if ((end_flags & FLAG_SUFX) == 0) {
   // no suffix has been removed
   if (dictionary_flags2 & FLAG_STEM)
    continue; // this word must have a suffix
  }

  if ((end_flags & SUFX_P) && (dictionary_flags2 & (FLAG_ONLY |
FLAG_ONLY_S)))
   continue; // $only or $onlys, don't match if a prefix has been
removed

  if (end_flags & FLAG_SUFX) {
   // a suffix was removed from the word
   if (dictionary_flags2 & FLAG_ONLY)
    continue; // no match if any suffix
```

```
   if ((dictionary_flags2 & FLAG_ONLY_S) && ((end_flags &
FLAG_SUFX_S) == 0)) {
    // only a 's' suffix allowed, but the suffix wasn't 's'
    continue;
   }
  }

  if (dictionary_flags2 & FLAG_HYPHENATED) {
   if (!(wflags & FLAG_HYPHEN_AFTER))
    continue;
  }
  if (dictionary_flags2 & FLAG_CAPITAL) {
   if (!(wflags & FLAG_FIRST_UPPER))
    continue;
  }
  if (dictionary_flags2 & FLAG_ALLCAPS) {
   if (!(wflags & FLAG_ALL_UPPER))
    continue;
  }
  if (dictionary_flags & FLAG_NEEDS_DOT) {
   if (!(wflags & FLAG_HAS_DOT))
    continue;
  }

  if ((dictionary_flags2 & FLAG_ATEND) && (word_end <
translator->clause_end) && (lookup_symbol == 0)) {
   // only use this pronunciation if it's the last word of the
clause, or called from Lookup()
   continue;
  }

  if ((dictionary_flags2 & FLAG_ATSTART) && !(wflags &
FLAG_FIRST_WORD)) {
   // only use this pronunciation if it's the first word of a
clause
   continue;
  }
```

```
  if ((dictionary_flags2 & FLAG_SENTENCE) &&
!(translator->clause_terminator & CLAUSE_TYPE_SENTENCE)) {
   // only if this clause is a sentence , i.e. terminator is {. ?
!} not {, : :}
   continue;
  }

  if (dictionary_flags2 & FLAG_VERB) {
   // this is a verb-form pronunciation

   if (tr->expect_verb || (tr->expect_verb_s && (end_flags &
FLAG_SUFX_S))) {
    // OK, we are expecting a verb
    if ((tr->translator_name == L('e', 'n')) &&
(tr->prev_dict_flags[0] & FLAG_ALT7_TRANS) && (end_flags &
FLAG_SUFX_S)) {
     // lang=en, don't use verb form after 'to' if the word has
's' suffix
     continue;
    }
   } else {
    // don't use the 'verb' pronunciation unless we are expecting
a verb
    continue;
   }
  }
  if (dictionary_flags2 & FLAG_PAST) {
   if (!tr->expect_past) {
    // don't use the 'past' pronunciation unless we are expecting
past tense
    continue;
   }
  }
  if (dictionary_flags2 & FLAG_NOUN) {
   if ((!tr->expect_noun) || (end_flags & SUFX_V)) {
    // don't use the 'noun' pronunciation unless we are expecting
```

```
a noun
    continue;
   }
  }
  if (dictionary_flags2 & FLAG_NATIVE) {
   if (tr != translator)
    continue; // don't use if we've switched translators
  }
  if (dictionary_flags & FLAG_ALT2_TRANS) {
   // language specific
   if ((tr->translator_name == L('h', 'u')) &&
!(tr->prev_dict_flags[0] & FLAG_ALT_TRANS))
    continue;
  }

  if (flags != NULL) {
   flags[0] = dictionary_flags | FLAG_FOUND_ATTRIBUTES;
   flags[1] = dictionary_flags2;
  }

  if (phoneme_len == 0) {
   if (option_phonemes & espeakPHONEMES_TRACE) {
    print_dictionary_flags(flags, dict_flags_buf,
sizeof(dict_flags_buf));
    fprintf(f_trans, "Flags:  %s  %s\n", word1, dict_flags_buf);
   }
   return 0; // no phoneme translation found here, only flags. So
use rules
  }

  if (flags != NULL)
   flags[0] |= FLAG_FOUND; // this flag indicates word was found
in dictionary

  if (option_phonemes & espeakPHONEMES_TRACE) {
   char ph_decoded[N_WORD_PHONEMES];
   bool textmode;
```

```
   DecodePhonemes(phonetic, ph_decoded);

   if ((dictionary_flags & FLAG_TEXTMODE) == 0)
    textmode = false;
   else
    textmode = true;

   if (textmode == translator->langopts.textmode) {
    // only show this line if the word translates to phonemes,
not replacement text
    if ((dictionary_flags & FLAG_SKIPWORDS) && (wtab != NULL)) {
     // matched more than one word
     // (check for wtab prevents showing RULE_SPELLING byte when
speaking individual letters)
     memcpy(word_buf, word2, word_end-word2);
     word_buf[word_end-word2-1] = 0;
     fprintf(f_trans, "Found: '%s %s\n", word1, word_buf);
    } else
     fprintf(f_trans, "Found: '%s", word1);
    print_dictionary_flags(flags, dict_flags_buf,
sizeof(dict_flags_buf));
    fprintf(f_trans, "' [%s]  %s\n", ph_decoded, dict_flags_buf);
   }
  }

  ix = utf8_in(&c, word);
  if (flags != NULL && (word[ix] == 0) && !IsAlpha(c))
   flags[0] |= FLAG_MAX3;
  return word_end;

 }
 return 0;
}

   Returns phonetic data in 'phonetic' and bits in 'flags'
```

```
   end_flags:  indicates if a suffix has been removed

int LookupDictList(Translator *tr, char **wordptr, char *ph_out,
unsigned int *flags, int end_flags, WORD_TAB *wtab)
{
 int length;
 const char *found;
 const char *word1;
 const char *word2;
 unsigned char c;
 int nbytes;
 int len;
 char word[N_WORD_BYTES];
 static char word_replacement[N_WORD_BYTES];

 length = 0;
 word2 = word1 = *wordptr;

 while ((word2[nbytes = utf8_nbytes(word2)] == ' ') &&
(word2[nbytes+1] == '.')) {
  // look for an abbreviation of the form a.b.c
  // try removing the spaces between the dots and looking for a
match
  memcpy(&word[length], word2, nbytes);
  length += nbytes;
  word[length++] = '.';
  word2 += nbytes+3;
 }
 if (length > 0) {
  // found an abbreviation containing dots
  nbytes = 0;
  while (((c = word2[nbytes]) != 0) && (c != ' '))
   nbytes++;
  memcpy(&word[length], word2, nbytes);
  word[length+nbytes] = 0;
  found =  LookupDict2(tr, word, word2, ph_out, flags, end_flags,
wtab);
```

```
  if (found) {
   // set the skip words flag
   flags[0] |= FLAG_SKIPWORDS;
   dictionary_skipwords = length;
   return 1;
  }
 }

 for (length = 0; length < (N_WORD_BYTES-1); length++) {
  if (((c = *word1++) == 0) || (c == ' '))
   break;

  if ((c == '.') && (length > 0) && (IsDigit09(word[length-1])))
   break; // needed for lang=hu, eg. "december 2.-ig"

  word[length] = c;
 }
 word[length] = 0;

 found = LookupDict2(tr, word, word1, ph_out, flags, end_flags,
wtab);

 if (flags[0] & FLAG_MAX3) {
  if (strcmp(ph_out, tr->phonemes_repeat) == 0) {
   tr->phonemes_repeat_count++;
   if (tr->phonemes_repeat_count > 3)
    ph_out[0] = 0;
  } else {
   strncpy0(tr->phonemes_repeat, ph_out,
sizeof(tr->phonemes_repeat));
   tr->phonemes_repeat_count = 1;
  }
 } else
  tr->phonemes_repeat_count = 0;

 if ((found == 0) && (flags[1] & FLAG_ACCENT)) {
  int letter;
```

```
  word2 = word;
  if (*word2 == '_') word2++;
  len = utf8_in(&letter, word2);
  LookupAccentedLetter(tr, letter, ph_out);
  found = word2 + len;
 }

 if (found == 0 && length >= 2) {
  ph_out[0] = 0;

  // try modifications to find a recognised word

  if ((end_flags & FLAG_SUFX_E_ADDED) && (word[length-1] == 'e'))
{
   // try removing an 'e' which has been added by RemoveEnding
   word[length-1] = 0;
   found = LookupDict2(tr, word, word1, ph_out, flags, end_flags,
wtab);
  } else if ((end_flags & SUFX_D) && (word[length-1] ==
word[length-2])) {
   // try removing a double letter
   word[length-1] = 0;
   found = LookupDict2(tr, word, word1, ph_out, flags, end_flags,
wtab);
  }
 }

 if (found) {
  // if textmode is the default, then words which have phonemes
are marked.
  if (tr->langopts.textmode)
   *flags ^= FLAG_TEXTMODE;

  if (*flags & FLAG_TEXTMODE) {
   // the word translates to replacement text, not to phonemes

   if (end_flags & FLAG_ALLOW_TEXTMODE) {
```

```
    // only use replacement text if this is the original word,
not if a prefix or suffix has been removed
    word_replacement[0] = 0;
    word_replacement[1] = ' ';
    sprintf(&word_replacement[2], "%s ", ph_out); // replacement
word, preceded by zerochar and space

    word1 = *wordptr;
    *wordptr = &word_replacement[2];

    if (option_phonemes & espeakPHONEMES_TRACE) {
     len = found - word1;
     memcpy(word, word1, len); // include multiple matching words
     word[len] = 0;
     fprintf(f_trans, "Replace: %s  %s\n", word, *wordptr);
    }
   }

   ph_out[0] = 0;
   return 0;
  }

  return 1;
 }

 ph_out[0] = 0;
 return 0;
}

extern char word_phonemes[N_WORD_PHONEMES]; // a word translated
into phoneme codes

int Lookup(Translator *tr, const char *word, char *ph_out)
{
 // Look up in *_list, returns dictionary flags[0] and phonemes

 int flags0;
```

```
 unsigned int flags[2];
 int say_as;
 char *word1 = (char *)word;
 char text[80];

 flags[0] = 0;
 flags[1] = FLAG_LOOKUP_SYMBOL;
 if ((flags0 = LookupDictList(tr, &word1, ph_out, flags,
FLAG_ALLOW_TEXTMODE, NULL)) != 0)
  flags0 = flags[0];

 if (flags[0] & FLAG_TEXTMODE) {
  say_as = option_sayas;
  option_sayas = 0; // don't speak replacement word as letter
names
  // NOTE: TranslateRoman checks text[-2], so pad the start of
text to prevent
  // it reading data on the stack.
  text[0] = ' ';
  text[1] = ' ';
  strncpy0(text+2, word1, sizeof(text)-2);
  flags0 = TranslateWord(tr, text+2, NULL, NULL);
  strcpy(ph_out, word_phonemes);
  option_sayas = say_as;
 }
 return flags0;
}

int LookupFlags(Translator *tr, const char *word, unsigned int
**flags_out)
{
 char buf[100];
 static unsigned int flags[2];
 char *word1 = (char *)word;

 flags[0] = flags[1] = 0;
 LookupDictList(tr, &word1, buf, flags, 0, NULL);
```

```
 return flags[0];
}

int RemoveEnding(Translator *tr, char *word, int end_type, char
*word_copy)
{
 /* Removes a standard suffix from a word, once it has been
indicated by the dictionary rules.
    end_type: bits 0-6  number of letters
              bits 8-14  suffix flags

     word_copy: make a copy of the original word
     This routine is language specific.  In English it deals with
reversing y->i and e-dropping
     that were done when the suffix was added to the original
word.
  */

 int i;
 char *word_end;
 int len_ending;
 int end_flags;
 const char *p;
 int len;
 char ending[50] = {0};

 // these lists are language specific, but are only relevent if
the 'e' suffix flag is used
 static const char *add_e_exceptions[] = {
  "ion", NULL
 };

 static const char *add_e_additions[] = {
  "c", "rs", "ir", "ur", "ath", "ns", "u",
  "spong", // sponge
  "rang", // strange
```

```
  "larg", // large
  NULL
 };

 for (word_end = word; *word_end != ' '; word_end++) {
  // replace discarded 'e's
  if (*word_end == REPLACED_E)
   *word_end = 'e';
 }
 i = word_end - word;

 if (word_copy != NULL) {
  memcpy(word_copy, word, i);
  word_copy[i] = 0;
 }

 // look for multibyte characters to increase the number of bytes
to remove
 for (len_ending = i = (end_type & 0x3f); i > 0; i--) { // num.of
characters of the suffix
  word_end--;
  while ((*word_end & 0xc0) == 0x80) {
   word_end--; // for multibyte characters
   len_ending++;
  }
 }

 // remove bytes from the end of the word and replace them by
spaces
 for (i = 0; (i < len_ending) && (i < (int)sizeof(ending)-1);
i++) {
  ending[i] = word_end[i];
  word_end[i] = ' ';
 }
 ending[i] = 0;
 word_end--; // now pointing at last character of stem
```

```
 end_flags = (end_type & 0xfff0) | FLAG_SUFX;

 /* add an 'e' to the stem if appropriate,
     if  stem ends in vowel+consonant
     or  stem ends in 'c'  (add 'e' to soften it) */

 if (end_type & SUFX_I) {
  if (word_end[0] == 'i')
   word_end[0] = 'y';
 }

 if (end_type & SUFX_E) {
  if (tr->translator_name == L('n', 'l')) {
   if (((word_end[0] & 0x80) == 0) && ((word_end[-1] & 0x80) ==
0) && IsVowel(tr, word_end[-1]) && IsLetter(tr, word_end[0],
LETTERGP_C) && !IsVowel(tr, word_end[-2])) {
    // double the vowel before the (ascii) final consonant
    word_end[1] = word_end[0];
    word_end[0] = word_end[-1];
    word_end[2] = ' ';
   }
  } else if (tr->translator_name == L('e', 'n')) {
   // add 'e' to end of stem
   if (IsLetter(tr, word_end[-1], LETTERGP_VOWEL2) &&
IsLetter(tr, word_end[0], 1)) {
    // vowel(incl.'y') + hard.consonant

    for (i = 0; (p = add_e_exceptions[i]) != NULL; i++) {
     len = strlen(p);
     if (memcmp(p, &word_end[1-len], len) == 0)
      break;
    }
    if (p == NULL)
     end_flags |= FLAG_SUFX_E_ADDED; // no exception found
   } else {
    for (i = 0; (p = add_e_additions[i]) != NULL; i++) {
     len = strlen(p);
```

```
      if (memcmp(p, &word_end[1-len], len) == 0) {
       end_flags |= FLAG_SUFX_E_ADDED;
       break;
      }
     }
    }
   } else if (tr->langopts.suffix_add_e != 0)
    end_flags |= FLAG_SUFX_E_ADDED;

   if (end_flags & FLAG_SUFX_E_ADDED) {
    utf8_out(tr->langopts.suffix_add_e, &word_end[1]);

    if (option_phonemes & espeakPHONEMES_TRACE)
     fprintf(f_trans, "add e\n");
   }
  }

  if ((end_type & SUFX_V) && (tr->expect_verb == 0))
   tr->expect_verb = 1; // this suffix indicates the verb
pronunciation

  if ((strcmp(ending, "s") == 0) || (strcmp(ending, "es") == 0))
   end_flags |= FLAG_SUFX_S;

  if (ending[0] == '\'')
   end_flags &= ~FLAG_SUFX; // don't consider 's as an added
suffix

  return end_flags;
}
```

Chapter 62

./src/libespeak-ng/intonation.c

```
#include "config.h"

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "intonation.h"
#include "synthdata.h"

#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

   provide a more flexible intonation system.

// bits in SYLLABLE.flags
```

```
#define SYL_RISE        1
#define SYL_EMPHASIS    2
#define SYL_END_CLAUSE   4

typedef struct {
 char stress;
 char env;
 char flags; // bit 0=pitch rising, bit1=emnphasized, bit2=end of
clause
 char nextph_type;
 unsigned char pitch1;
 unsigned char pitch2;
} SYLLABLE;

static int tone_pitch_env; // used to return pitch envelope

#define PITCHfall     0
#define PITCHrise     2
#define PITCHfrise    4 // and 3 must be for the variant preceded
by 'r'
#define PITCHfrise2   6 // and 5 must be the 'r' variant

unsigned char env_fall[128] = {
 0xff, 0xfd, 0xfa, 0xf8, 0xf6, 0xf4, 0xf2, 0xf0, 0xee, 0xec,
0xea, 0xe8, 0xe6, 0xe4, 0xe2, 0xe0,
 0xde, 0xdc, 0xda, 0xd8, 0xd6, 0xd4, 0xd2, 0xd0, 0xce, 0xcc,
0xca, 0xc8, 0xc6, 0xc4, 0xc2, 0xc0,
 0xbe, 0xbc, 0xba, 0xb8, 0xb6, 0xb4, 0xb2, 0xb0, 0xae, 0xac,
0xaa, 0xa8, 0xa6, 0xa4, 0xa2, 0xa0,
 0x9e, 0x9c, 0x9a, 0x98, 0x96, 0x94, 0x92, 0x90, 0x8e, 0x8c,
0x8a, 0x88, 0x86, 0x84, 0x82, 0x80,
 0x7e, 0x7c, 0x7a, 0x78, 0x76, 0x74, 0x72, 0x70, 0x6e, 0x6c,
0x6a, 0x68, 0x66, 0x64, 0x62, 0x60,
 0x5e, 0x5c, 0x5a, 0x58, 0x56, 0x54, 0x52, 0x50, 0x4e, 0x4c,
0x4a, 0x48, 0x46, 0x44, 0x42, 0x40,
 0x3e, 0x3c, 0x3a, 0x38, 0x36, 0x34, 0x32, 0x30, 0x2e, 0x2c,
0x2a, 0x28, 0x26, 0x24, 0x22, 0x20,
```

```
 0x1e, 0x1c, 0x1a, 0x18, 0x16, 0x14, 0x12, 0x10, 0x0e, 0x0c,
0x0a, 0x08, 0x06, 0x04, 0x02, 0x00
};

unsigned char env_rise[128] = {
  // Table removed from listning (similar to previous one)
};

unsigned char env_frise[128] = {
  // Table removed from listning (similar to previous one)

};

static unsigned char env_r_frise[128] = {

};

static unsigned char env_frise2[128] = {

};

static unsigned char env_r_frise2[128] = {

};

static unsigned char env_risefall[128] = {

};

static unsigned char env_rise2[128] = {

};

static unsigned char env_fall2[128] = {

};
```

```
static unsigned char env_fallrise3[128] = {

};

static unsigned char env_fallrise4[128] = {

};

static unsigned char env_risefallrise[128] = {

};

unsigned char *envelope_data[N_ENVELOPE_DATA] = {
 env_fall,  env_fall,
 env_rise,  env_rise,
 env_frise,  env_r_frise,
 env_frise2, env_r_frise2,
 env_risefall, env_risefall,

 env_fallrise3, env_fallrise3,
 env_fallrise4, env_fallrise4,
 env_fall2, env_fall2,
 env_rise2, env_rise2,
 env_risefallrise, env_risefallrise
};

// indexed by stress
static int min_drop[] =  { 6, 7, 9, 9, 20, 20, 20, 25 };

// pitch change during the main part of the clause
static int drops_0[8] = { 9, 9, 16, 16, 16, 23, 55, 32 };

// overflow table values are 64ths of the body pitch range
(between body_start and body_end)
static signed char oflow[] = { 0, 40, 24, 8, 0 };
static signed char oflow_emf[] = { 10, 52, 32, 20, 10 };
static signed char oflow_less[] = { 6, 38, 24, 14, 4 };
```

```
#define N_TONE_HEAD_TABLE    13
#define N_TONE_NUCLEUS_TABLE 13

typedef struct {
 unsigned char pre_start;
 unsigned char pre_end;

 unsigned char body_start;
 unsigned char body_end;

 int  *body_drops;
 unsigned char body_max_steps;
 char body_lower_u;

 unsigned char n_overflow;
 signed char *overflow;
} TONE_HEAD;

typedef struct {
 unsigned char pitch_env0; // pitch envelope, tonic syllable at
end
 unsigned char tonic_max0;
 unsigned char tonic_min0;

 unsigned char pitch_env1; // followed by unstressed
 unsigned char tonic_max1;
 unsigned char tonic_min1;

 short *backwards;

 unsigned char tail_start;
 unsigned char tail_end;
 unsigned char flags;
} TONE_NUCLEUS;

#define T_EMPH  1
```

```
static TONE_HEAD tone_head_table[N_TONE_HEAD_TABLE] = {
 { 46, 57,   78, 50,  drops_0, 3, 7,   5, oflow },      // 0
statement
 { 46, 57,   78, 46,  drops_0, 3, 7,   5, oflow },      // 1
comma
 { 46, 57,   78, 46,  drops_0, 3, 7,   5, oflow },      // 2
question
 { 46, 57,   90, 50,  drops_0, 3, 9,   5, oflow_emf },  // 3
exclamation
 { 46, 57,   78, 50,  drops_0, 3, 7,   5, oflow },      // 4
statement, emphatic
 { 46, 57,   74, 55,  drops_0, 4, 7,   5, oflow_less }, // 5
statement, less intonation
 { 46, 57,   74, 55,  drops_0, 4, 7,   5, oflow_less }, // 6
comma, less intonation
 { 46, 57,   74, 55,  drops_0, 4, 7,   5, oflow_less }, // 7
comma, less intonation, less rise
 { 46, 57,   78, 50,  drops_0, 3, 7,   5, oflow },      // 8
pitch raises at end of sentence
 { 46, 57,   78, 46,  drops_0, 3, 7,   5, oflow },      // 9
comma
 { 46, 57,   78, 50,  drops_0, 3, 7,   5, oflow },      // 10
question
 { 34, 41,   41, 32,  drops_0, 3, 7,   5, oflow_less }, // 11
test
 { 46, 57,   55, 50,  drops_0, 3, 7,   5, oflow_less }, // 12
test
};

static TONE_NUCLEUS tone_nucleus_table[N_TONE_NUCLEUS_TABLE] = {
 { PITCHfall,   64,  8, PITCHfall,   70, 18, NULL, 24, 12, 0 },
//  0 statement
 { PITCHfrise,  80, 18, PITCHfrise2, 78, 22, NULL, 34, 52, 0 },
//  1 comma
 { PITCHfrise,  88, 22, PITCHfrise2, 82, 22, NULL, 34, 64, 0 },
//  2 question
```

```
 { PITCHfall,   92,  8, PITCHfall,   92, 80, NULL, 76,  8, T_EMPH
}, //  3 exclamation
 { PITCHfall,   86,  4, PITCHfall,   94, 66, NULL, 34, 10, 0 },
//  4 statement, emphatic
 { PITCHfall,   62, 10, PITCHfall,   62, 20, NULL, 28, 16, 0 },
//  5 statement, less intonation
 { PITCHfrise,  68, 18, PITCHfrise2, 68, 22, NULL, 30, 44, 0 },
//  6 comma, less intonation
 { PITCHfrise2, 64, 16, PITCHfall,   66, 32, NULL, 32, 18, 0 },
//  7 comma, less intonation, less rise
 { PITCHrise,   68, 46, PITCHfall,   42, 32, NULL, 46, 58, 0 },
//  8 pitch raises at end of sentence
 { PITCHfrise,  78, 24, PITCHfrise2, 72, 22, NULL, 42, 52, 0 },
//  9 comma
 { PITCHfrise,  88, 34, PITCHfall,   64, 32, NULL, 46, 82, 0 },
// 10 question
 { PITCHfall,   56, 12, PITCHfall,   56, 20, NULL, 24, 12, 0 },
// 11 test
 { PITCHfall,   70, 18, PITCHfall,   70, 24, NULL, 32, 20, 0 },
// 12 test
};

int n_tunes = 0;
TUNE *tunes = NULL;

#define SECONDARY        3
#define PRIMARY          4
#define PRIMARY_STRESSED 6
#define PRIMARY_LAST     7

static int number_pre;
static int number_body;
static int number_tail;
static int last_primary;
static int tone_posn;
static int tone_posn2;
static int no_tonic;
```

```
static void count_pitch_vowels(SYLLABLE *syllable_tab, int start,
int end, int clause_end)
{
 int ix;
 int stress;
 int max_stress = 0;
 int max_stress_posn = 0;  // last syllable ot the highest stress
 int max_stress_posn2 = 0; // penuntimate syllable of the highest
stress

 number_pre = -1; // number of vowels before 1st primary stress
 number_body = 0;
 number_tail = 0; // number between tonic syllable and next
primary
 last_primary = -1;

 for (ix = start; ix < end; ix++) {
  stress = syllable_tab[ix].stress; // marked stress level

  if (stress >= max_stress) {
   if (stress > max_stress)
    max_stress_posn2 = ix;
   else
    max_stress_posn2 = max_stress_posn;
   max_stress_posn = ix;
   max_stress = stress;
  }
  if (stress >= PRIMARY) {
   if (number_pre < 0)
    number_pre = ix - start;

   last_primary = ix;
  }
 }

 if (number_pre < 0)
```

```
  number_pre = end;

 number_tail = end - max_stress_posn - 1;
 tone_posn = max_stress_posn;
 tone_posn2 = max_stress_posn2;

 if (no_tonic)
  tone_posn = tone_posn2 = end; // next position after the end of
the truncated clause
 else if (last_primary >= 0) {
  if (end == clause_end)
   syllable_tab[last_primary].stress = PRIMARY_LAST;
 } else {
  // no primary stress. Use the highest stress
  syllable_tab[tone_posn].stress = PRIMARY_LAST;
 }
}

// Count number of primary stresses up to tonic syllable or
body_reset
static int count_increments(SYLLABLE *syllable_tab, int ix, int
end_ix, int min_stress)
{
 int count = 0;
 int stress;

 while (ix < end_ix) {
  stress = syllable_tab[ix++].stress;
  if (stress >= PRIMARY_LAST)
   break;

  if (stress >= min_stress)
   count++;
 }
 return count;
}
```

```
// Set the pitch of a vowel in syllable_tab
static void set_pitch(SYLLABLE *syl, int base, int drop)
{
 int pitch1, pitch2;
 int flags = 0;

 if (base < 0)  base = 0;

 pitch2 = base;

 if (drop < 0) {
  flags = SYL_RISE;
  drop = -drop;
 }

 pitch1 = pitch2 + drop;
 if (pitch1 < 0)
  pitch1 = 0;

 if (pitch1 > 254) pitch1 = 254;
 if (pitch2 > 254) pitch2 = 254;

 syl->pitch1 = pitch1;
 syl->pitch2 = pitch2;
 syl->flags |= flags;
}

static int CountUnstressed(SYLLABLE *syllable_tab, int start, int
end, int limit)
{
 int ix;

 for (ix = start; ix <= end; ix++) {
  if (syllable_tab[ix].stress >= limit)
   break;
 }
 return ix - start;
```

```
}

static int SetHeadIntonation(SYLLABLE *syllable_tab, TUNE *tune,
int syl_ix, int end_ix)
{
 int stress;
 SYLLABLE *syl;
 int ix;
 int pitch = 0;
 int increment = 0;
 int n_steps = 0;
 int stage; // onset, head, last
 bool initial;
 int overflow_ix = 0;
 int pitch_range;
 int pitch_range_abs;
 int *drops;
 int n_unstressed = 0;
 int unstressed_ix = 0;
 int unstressed_inc;
 bool used_onset = false;
 int head_final = end_ix;
 int secondary = 2;

 pitch_range = (tune->head_end - tune->head_start) << 8;
 pitch_range_abs = abs(pitch_range);
 drops = drops_0; // this should be controled by tune->head_drops
 initial = true;

 stage = 0;
 if (tune->onset == 255)
  stage = 1; // no onset specified

 if (tune->head_last != 255) {
  // find the last primary stress in the body
  for (ix = end_ix-1; ix >= syl_ix; ix--) {
   if (syllable_tab[ix].stress >= 4) {
```

```
   head_final = ix;
   break;
  }
 }
}

while (syl_ix < end_ix) {
 syl = &syllable_tab[syl_ix];
 stress = syl->stress;

 if (initial || (stress >= 4)) {
  // a primary stress

  if ((initial) || (stress == 5)) {
   initial = false;
   overflow_ix = 0;

   if (tune->onset == 255) {
    n_steps = count_increments(syllable_tab, syl_ix, head_final,
4);
    pitch = tune->head_start << 8;
   } else {
    // a pitch has been specified for the onset syllable, don't
include it in the pitch incrementing
    n_steps = count_increments(syllable_tab, syl_ix+1,
head_final, 4);
    pitch = tune->onset << 8;
    used_onset = true;
   }

   if (n_steps > tune->head_max_steps)
    n_steps = tune->head_max_steps;

   if (n_steps > 1)
    increment = pitch_range / (n_steps -1);
   else
    increment = 0;
```

```
   } else if (syl_ix == head_final) {
    // a pitch has been specified for the last primary stress
before the nucleus
    pitch = tune->head_last << 8;
    stage = 2;
   } else {
    if (used_onset) {
     stage = 1;
     used_onset = false;
     pitch = tune->head_start << 8;
     n_steps++;
    } else if (n_steps > 0)
     pitch += increment;
    else {
     pitch = (tune->head_end << 8) + (pitch_range_abs *
tune->head_extend[overflow_ix++])/64;
     if (overflow_ix >= tune->n_head_extend)
      overflow_ix = 0;
    }
   }

   n_steps--;
  }

  if (stress >= PRIMARY) {
   n_unstressed = CountUnstressed(syllable_tab, syl_ix+1, end_ix,
secondary);
   unstressed_ix = 0;
   syl->stress = PRIMARY_STRESSED;
   syl->env = tune->stressed_env;
   set_pitch(syl, (pitch >> 8), tune->stressed_drop);
  } else if (stress >= secondary) {
   n_unstressed = CountUnstressed(syllable_tab, syl_ix+1, end_ix,
secondary);
   unstressed_ix = 0;
   set_pitch(syl, (pitch >> 8), drops[stress]);
  } else {
```

```
   if (n_unstressed > 1)
    unstressed_inc = (tune->unstr_end[stage] -
tune->unstr_start[stage]) / (n_unstressed - 1);
   else
    unstressed_inc = 0;

   set_pitch(syl, (pitch >> 8) + tune->unstr_start[stage] +
(unstressed_inc * unstressed_ix), drops[stress]);
   unstressed_ix++;
  }

  syl_ix++;
 }
 return syl_ix;
}

    Increment pitch if stress is >= min_stress.
    Used for tonic segment */
static int calc_pitch_segment(SYLLABLE *syllable_tab, int ix, int
end_ix, TONE_HEAD *th, TONE_NUCLEUS *tn, int min_stress, bool
continuing)
{
 int stress;
 int pitch = 0;
 int increment = 0;
 int n_primary = 0;
 int n_steps = 0;
 bool initial;
 int overflow = 0;
 int n_overflow;
 int pitch_range;
 int pitch_range_abs;
 int *drops;
 signed char *overflow_tab;
 SYLLABLE *syl;

 static signed char continue_tab[5] = { -26, 32, 20, 8, 0 };
```

```
drops = th->body_drops;
pitch_range = (th->body_end - th->body_start) << 8;
pitch_range_abs = abs(pitch_range);

if (continuing) {
 initial = false;
 overflow = 0;
 n_overflow = 5;
 overflow_tab = continue_tab;
 increment = pitch_range / (th->body_max_steps -1);
} else {
 n_overflow = th->n_overflow;
 overflow_tab = th->overflow;
 initial = true;
}

while (ix < end_ix) {
 syl = &syllable_tab[ix];
 stress = syl->stress;

 if (initial || (stress >= min_stress)) {
  // a primary stress

  if ((initial) || (stress == 5)) {
   initial = false;
   overflow = 0;
   n_steps = n_primary = count_increments(syllable_tab, ix,
end_ix, min_stress);

   if (n_steps > th->body_max_steps)
    n_steps = th->body_max_steps;

   if (n_steps > 1)
    increment = pitch_range / (n_steps -1);
   else
    increment = 0;
```

```
    pitch = th->body_start << 8;
   } else {
    if (n_steps > 0)
     pitch += increment;
    else {
     pitch = (th->body_end << 8) + (pitch_range_abs *
overflow_tab[overflow++])/64;
     if (overflow >= n_overflow) {
      overflow = 0;
      overflow_tab = th->overflow;
     }
    }
   }

   n_steps--;

   n_primary--;
   if ((tn->backwards) && (n_primary < 2))
    pitch = tn->backwards[n_primary] << 8;
  }

  if (stress >= PRIMARY) {
   syl->stress = PRIMARY_STRESSED;
   set_pitch(syl, (pitch >> 8), drops[stress]);
  } else if (stress >= SECONDARY)
   set_pitch(syl, (pitch >> 8), drops[stress]);
  else {
   // unstressed, drop pitch if preceded by PRIMARY
   if ((syllable_tab[ix-1].stress & 0x3f) >= SECONDARY)
    set_pitch(syl, (pitch >> 8) - th->body_lower_u,
drops[stress]);
   else
    set_pitch(syl, (pitch >> 8), drops[stress]);
  }

  ix++;
```

```
 }
 return ix;
}

static void SetPitchGradient(SYLLABLE *syllable_tab, int
start_ix, int end_ix, int start_pitch, int end_pitch)
{
 // Set a linear pitch change over a number of syllables.
 // Used for pre-head, unstressed syllables in the body, and the
tail

 int ix;
 int stress;
 int pitch;
 int increment;
 int n_increments;
 int drop;
 SYLLABLE *syl;

 increment = (end_pitch - start_pitch) << 8;
 n_increments = end_ix - start_ix;

 if (n_increments <= 0)
  return;

 if (n_increments > 1)
  increment = increment / n_increments;

 pitch = start_pitch << 8;

 for (ix = start_ix; ix < end_ix; ix++) {
  syl = &syllable_tab[ix];
  stress = syl->stress;

  if (increment > 0) {
   set_pitch(syl, (pitch >> 8), -(increment >> 8));
   pitch += increment;
```

```
  } else {
   drop = -(increment >> 8);
   if (drop < min_drop[stress])
    drop = min_drop[stress];

   pitch += increment;

   if (drop > 18)
    drop = 18;
   set_pitch(syl, (pitch >> 8), drop);
  }
 }
}

// Calculate pitch values for the vowels in this tone group
static int calc_pitches2(SYLLABLE *syllable_tab, int start, int
end,  int tune_number)
{
 int ix;
 TUNE *tune;
 int drop;

 tune = &tunes[tune_number];
 ix = start;

 // vowels before the first primary stress

 SetPitchGradient(syllable_tab, ix, ix+number_pre,
tune->prehead_start, tune->prehead_end);
 ix += number_pre;

 // body of tonic segment

 if (option_tone_flags & OPTION_EMPHASIZE_PENULTIMATE)
  tone_posn = tone_posn2; // put tone on the penultimate stressed
word
 ix = SetHeadIntonation(syllable_tab, tune, ix, tone_posn);
```

```
 if (no_tonic)
  return 0;

 // tonic syllable

 if (number_tail == 0) {
  tone_pitch_env = tune->nucleus0_env;
  drop = tune->nucleus0_max - tune->nucleus0_min;
  set_pitch(&syllable_tab[ix++], tune->nucleus0_min, drop);
 } else {
  tone_pitch_env = tune->nucleus1_env;
  drop = tune->nucleus1_max - tune->nucleus1_min;
  set_pitch(&syllable_tab[ix++], tune->nucleus1_min, drop);
 }

 syllable_tab[tone_posn].env = tone_pitch_env;
 if (syllable_tab[tone_posn].stress == PRIMARY)
  syllable_tab[tone_posn].stress = PRIMARY_STRESSED;

 // tail, after the tonic syllable

 SetPitchGradient(syllable_tab, ix, end, tune->tail_start,
tune->tail_end);

 return tone_pitch_env;
}

// Calculate pitch values for the vowels in this tone group
static int calc_pitches(SYLLABLE *syllable_tab, int control, int
start, int end,  int tune_number)
{
 int ix;
 TONE_HEAD *th;
 TONE_NUCLEUS *tn;
 int drop;
 bool continuing = false;
```

```
 if (control == 0)
  return calc_pitches2(syllable_tab, start, end, tune_number);

 if (start > 0)
  continuing = true;

 th = &tone_head_table[tune_number];
 tn = &tone_nucleus_table[tune_number];
 ix = start;

 // vowels before the first primary stress

 SetPitchGradient(syllable_tab, ix, ix+number_pre, th->pre_start,
th->pre_end);
 ix += number_pre;

 // body of tonic segment

 if (option_tone_flags & OPTION_EMPHASIZE_PENULTIMATE)
  tone_posn = tone_posn2; // put tone on the penultimate stressed
word
 ix = calc_pitch_segment(syllable_tab, ix, tone_posn, th, tn,
PRIMARY, continuing);

 if (no_tonic)
  return 0;

 // tonic syllable

 if (tn->flags & T_EMPH)
  syllable_tab[ix].flags |= SYL_EMPHASIS;

 if (number_tail == 0) {
  tone_pitch_env = tn->pitch_env0;
  drop = tn->tonic_max0 - tn->tonic_min0;
  set_pitch(&syllable_tab[ix++], tn->tonic_min0, drop);
```

```
 } else {
  tone_pitch_env = tn->pitch_env1;
  drop = tn->tonic_max1 - tn->tonic_min1;
  set_pitch(&syllable_tab[ix++], tn->tonic_min1, drop);
 }

 syllable_tab[tone_posn].env = tone_pitch_env;
 if (syllable_tab[tone_posn].stress == PRIMARY)
  syllable_tab[tone_posn].stress = PRIMARY_STRESSED;

 // tail, after the tonic syllable

 SetPitchGradient(syllable_tab, ix, end, tn->tail_start,
tn->tail_end);

 return tone_pitch_env;
}

static void CalcPitches_Tone(Translator *tr)
{
 PHONEME_LIST *p;
 int ix;
 int count_stressed = 0;
 int final_stressed = 0;

 int tone_ph;
 bool pause;
 bool tone_promoted;
 PHONEME_TAB *tph;
 PHONEME_TAB *prev_tph; // forget across word boundary
 PHONEME_TAB *prevw_tph; // remember across word boundary
 PHONEME_LIST *prev_p;

 int pitch_adjust = 0;    // pitch gradient through the clause -
inital value
 int pitch_decrement = 0; // decrease by this for each stressed
syllable
```

```
 int pitch_low = 0;       // until it drops to this
 int pitch_high = 0;      // then reset to this

 // count number of stressed syllables
 p = &phoneme_list[0];
 for (ix = 0; ix < n_phoneme_list; ix++, p++) {
  if ((p->type == phVOWEL) && (p->stresslevel >= 4)) {
   if (count_stressed == 0)
    final_stressed = ix;

   if (p->stresslevel >= 4) {
    final_stressed = ix;
    count_stressed++;
   }
  }
 }

 phoneme_list[final_stressed].stresslevel = 7;

 // language specific, changes to tones
 if (tr->translator_name == L('v', 'i')) {
  // LANG=vi
  p = &phoneme_list[final_stressed];
  if (p->tone_ph == 0)
   p->tone_ph = PhonemeCode('7'); // change default tone (tone 1)
to falling tone at end of clause
 }

 pause = true;
 tone_promoted = false;

 prev_p = p = &phoneme_list[0];
 prev_tph = prevw_tph = phoneme_tab[phonPAUSE];

 // perform tone sandhi
 for (ix = 0; ix < n_phoneme_list; ix++, p++) {
  if ((p->type == phPAUSE) && (p->ph->std_length > 50)) {
```

```
   pause = true; // there is a pause since the previous vowel
   prevw_tph = phoneme_tab[phonPAUSE]; // forget previous tone
  }

  if (p->newword)
   prev_tph = phoneme_tab[phonPAUSE]; // forget across word
boundaries

  if (p->synthflags & SFLAG_SYLLABLE) {
   tone_ph = p->tone_ph;
   tph = phoneme_tab[tone_ph];

   /* Hakka
   ref.:https://en.wikipedia.org/wiki/Sixian_dialect#Tone_sandhi
*/
   if (tr->translator_name == L3('h','a','k')){
    if (prev_tph->mnemonic == 0x31){ // [previous one is 1st
tone]
      // [this one is 1st, 4th, or 6th tone]
      if (tph->mnemonic == 0x31 || tph->mnemonic == 0x34 ||
       tph->mnemonic == 0x36){
     /* trigger the tone sandhi of the prev. syllable
        from 1st tone ->2nd tone */
     prev_p->tone_ph = PhonemeCode('2');
      }
    }
     }
   // Mandarin
   if (tr->translator_name == L('z', 'h')) {
    if (tone_ph == 0) {
     if (pause || tone_promoted) {
      tone_ph = PhonemeCode2('5', '5'); // no previous vowel, use
tone 1
      tone_promoted = true;
     } else
      tone_ph = PhonemeCode2('1', '1'); // default tone 5
```

```
     p->tone_ph = tone_ph;
     tph = phoneme_tab[tone_ph];
    } else
     tone_promoted = false;

    if (ix == final_stressed) {
     if ((tph->mnemonic == 0x3535 ) || (tph->mnemonic == 0x3135))
{
      // change sentence final tone 1 or 4 to stress 6, not 7
      phoneme_list[final_stressed].stresslevel = 6;
     }
    }

    if (prevw_tph->mnemonic == 0x343132) { // [214]
     if (tph->mnemonic == 0x343132) // [214]
      prev_p->tone_ph = PhonemeCode2('3', '5');
     else
      prev_p->tone_ph = PhonemeCode2('2', '1');
    }
    if ((prev_tph->mnemonic == 0x3135)  && (tph->mnemonic ==
0x3135)) // [51] + [51]
     prev_p->tone_ph = PhonemeCode2('5', '3');

    if (tph->mnemonic == 0x3131) { // [11] Tone 5
     // tone 5, change its level depending on the previous tone
(across word boundaries)
     if (prevw_tph->mnemonic == 0x3535)
      p->tone_ph = PhonemeCode2('2', '2');
     if (prevw_tph->mnemonic == 0x3533)
      p->tone_ph = PhonemeCode2('3', '3');
     if (prevw_tph->mnemonic == 0x343132)
      p->tone_ph = PhonemeCode2('4', '4');

     // tone 5 is unstressed (shorter)
     p->stresslevel = 0; // diminished stress
    }
   }
```

```
   prev_p = p;
   prevw_tph = prev_tph = tph;
   pause = false;
  }
 }

 // convert tone numbers to pitch
 p = &phoneme_list[0];
 for (ix = 0; ix < n_phoneme_list; ix++, p++) {
  if (p->synthflags & SFLAG_SYLLABLE) {
   tone_ph = p->tone_ph;

   if (p->stresslevel != 0) { // TEST, consider all syllables as
stressed
    if (ix == final_stressed) {
     // the last stressed syllable
     pitch_adjust = pitch_low;
    } else {
     pitch_adjust -= pitch_decrement;
     if (pitch_adjust <= pitch_low)
      pitch_adjust = pitch_high;
    }
   }

   if (tone_ph == 0) {
    tone_ph = phonDEFAULTTONE; // no tone specified, use default
tone 1
    p->tone_ph = tone_ph;
   }
   p->pitch1 = pitch_adjust + phoneme_tab[tone_ph]->start_type;
   p->pitch2 = pitch_adjust + phoneme_tab[tone_ph]->end_type;
  }
 }
}

void CalcPitches(Translator *tr, int clause_type)
```

```
{
 // clause_type: 0=. 1=, 2=?, 3=! 4=none

 PHONEME_LIST *p;
 SYLLABLE *syl;
 int ix;
 int x;
 int st_ix;
 int n_st;
 int option;
 int group_tone;
 int group_tone_comma;
 int ph_start = 0;
 int st_start;
 int st_clause_end;
 int count;
 int n_primary;
 int count_primary;
 PHONEME_TAB *ph;
 int ph_end = n_phoneme_list;

 SYLLABLE syllable_tab[N_PHONEME_LIST];
 n_st = 0;
 n_primary = 0;
 for (ix = 0; ix < (n_phoneme_list-1); ix++) {
  p = &phoneme_list[ix];
  syllable_tab[ix].flags = 0;
  if (p->synthflags & SFLAG_SYLLABLE) {
   syllable_tab[n_st].env = PITCHfall;
   syllable_tab[n_st].nextph_type = phoneme_list[ix+1].type;
   syllable_tab[n_st++].stress = p->stresslevel;

   if (p->stresslevel >= 4)
    n_primary++;
  } else if ((p->ph->code == phonPAUSE_CLAUSE) && (n_st > 0))
   syllable_tab[n_st-1].flags |= SYL_END_CLAUSE;
 }
```

```
 syllable_tab[n_st].stress = 0; // extra 0 entry at the end

 if (n_st == 0)
  return; // nothing to do

 if (tr->langopts.tone_language == 1) {
  CalcPitches_Tone(tr);
  return;
 }

 option = tr->langopts.intonation_group;
 if (option >= INTONATION_TYPES)
  option = 1;

 if (option == 0) {
  group_tone = tr->langopts.tunes[clause_type];
  group_tone_comma = tr->langopts.tunes[1];
 } else {
  group_tone = tr->punct_to_tone[option][clause_type];
  group_tone_comma = tr->punct_to_tone[option][1]; // emphatic
form of statement
 }

 if (clause_type == 4)
  no_tonic = 1; // incomplete clause, used for abbreviations such
as Mr. Dr. Mrs.
 else
  no_tonic = 0;

 st_start = 0;
 count_primary = 0;
 for (st_ix = 0; st_ix < n_st; st_ix++) {
  syl = &syllable_tab[st_ix];

  if (syl->stress >= 4)
   count_primary++;
```

```
  if (syl->stress == 6) {
   // reduce the stress of the previous stressed syllable (review
only the previous few syllables)
   for (ix = st_ix-1; ix >= st_start && ix >= (st_ix-3); ix--) {
    if (syllable_tab[ix].stress == 6)
     break;
    if (syllable_tab[ix].stress == 4) {
     syllable_tab[ix].stress = 3;
     break;
    }
   }

   // are the next primary syllables also emphasized ?
   for (ix = st_ix+1; ix < n_st; ix++) {
    if (syllable_tab[ix].stress == 4)
     break;
    if (syllable_tab[ix].stress == 6) {
     // emphasize this syllable, but don't end the current tone
group
     syllable_tab[st_ix].flags = SYL_EMPHASIS;
     syl->stress = 5;
     break;
    }
   }
  }

  if (syl->stress == 6) {
   // an emphasized syllable, end the tone group after the next
primary stress
   syllable_tab[st_ix].flags = SYL_EMPHASIS;

   count = 0;
   if ((n_primary - count_primary) > 1)
    count = 1;

   for (ix = st_ix+1; ix < n_st; ix++) {
    if (syllable_tab[ix].stress > 4)
```

```
     break;
    if (syllable_tab[ix].stress == 4) {
     count++;
     if (count > 1)
      break;
    }
   }

   count_pitch_vowels(syllable_tab, st_start, ix, n_st);
   if ((ix < n_st) || (clause_type == 0)) {
    calc_pitches(syllable_tab, option, st_start, ix, group_tone);
// split into > 1 tone groups

    if ((clause_type == 1) || (clause_type == 2))
     group_tone = tr->langopts.tunes[1]; // , or ?  remainder has
comma-tone
    else
     group_tone = tr->langopts.tunes[0]; // . or !  remainder has
statement tone
   } else
    calc_pitches(syllable_tab, option, st_start, ix, group_tone);

   st_start = ix;
  }
  if ((st_start < st_ix) && (syl->flags & SYL_END_CLAUSE)) {
   // end of clause after this syllable, indicated by a
phonPAUSE_CLAUSE phoneme
   st_clause_end = st_ix+1;
   count_pitch_vowels(syllable_tab, st_start, st_clause_end,
st_clause_end);
   calc_pitches(syllable_tab, option, st_start, st_clause_end,
group_tone_comma);
   st_start = st_clause_end;
  }
 }

 if (st_start < st_ix) {
```

```
  count_pitch_vowels(syllable_tab, st_start, st_ix, n_st);
  calc_pitches(syllable_tab, option, st_start, st_ix,
group_tone);
 }

 // unpack pitch data
 st_ix = 0;
 for (ix = ph_start; ix < ph_end; ix++) {
  p = &phoneme_list[ix];
  p->stresslevel = syllable_tab[st_ix].stress;

  if (p->synthflags & SFLAG_SYLLABLE) {
   syl = &syllable_tab[st_ix];

   p->pitch1 = syl->pitch1;
   p->pitch2 = syl->pitch2;

   p->env = PITCHfall;
   if (syl->flags & SYL_RISE)
    p->env = PITCHrise;
   else if (p->stresslevel > 5)
    p->env = syl->env;

   if (p->pitch1 > p->pitch2) {
    // swap so that pitch2 is the higher
    x = p->pitch1;
    p->pitch1 = p->pitch2;
    p->pitch2 = x;
   }

   if (p->tone_ph) {
    ph = phoneme_tab[p->tone_ph];
    x = (p->pitch1 + p->pitch2)/2;
    p->pitch2 = x + ph->end_type;
    p->pitch1 = x + ph->start_type;
   }
```

```
   if (syl->flags & SYL_EMPHASIS)
    p->stresslevel |= 8; // emphasized

   st_ix++;
  }
 }
}
```

Chapter 63

./src/libespeak-ng/encoding.c

```
#include "config.h"

#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include <wchar.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/encoding.h>

#include "speech.h"
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

// http://www.iana.org/assignments/character-sets/character-
sets.xhtml
MNEM_TAB mnem_encoding[] = {
 { "ANSI_X3.4-1968",   ESPEAKNG_ENCODING_US_ASCII },
 { "ANSI_X3.4-1986",   ESPEAKNG_ENCODING_US_ASCII },
 { "ASMO-708",         ESPEAKNG_ENCODING_ISO_8859_6 },
```

```
{ "ECMA-114",          ESPEAKNG_ENCODING_ISO_8859_6 },
{ "ECMA-118",          ESPEAKNG_ENCODING_ISO_8859_7 },
{ "ELOT_928",          ESPEAKNG_ENCODING_ISO_8859_7 },
{ "IBM367",            ESPEAKNG_ENCODING_US_ASCII },
{ "IBM819",            ESPEAKNG_ENCODING_ISO_8859_1 },
{ "ISCII",             ESPEAKNG_ENCODING_ISCII },
{ "ISO_646.irv:1991", ESPEAKNG_ENCODING_US_ASCII },
{ "ISO_8859-1",        ESPEAKNG_ENCODING_ISO_8859_1 },
{ "ISO_8859-1:1987",  ESPEAKNG_ENCODING_ISO_8859_1 },
{ "ISO_8859-2",        ESPEAKNG_ENCODING_ISO_8859_2 },
{ "ISO_8859-2:1987",  ESPEAKNG_ENCODING_ISO_8859_2 },
{ "ISO_8859-3",        ESPEAKNG_ENCODING_ISO_8859_3 },
{ "ISO_8859-3:1988",  ESPEAKNG_ENCODING_ISO_8859_3 },
{ "ISO_8859-4",        ESPEAKNG_ENCODING_ISO_8859_4 },
{ "ISO_8859-4:1988",  ESPEAKNG_ENCODING_ISO_8859_4 },
{ "ISO_8859-5",        ESPEAKNG_ENCODING_ISO_8859_5 },
{ "ISO_8859-5:1988",  ESPEAKNG_ENCODING_ISO_8859_5 },
{ "ISO_8859-6",        ESPEAKNG_ENCODING_ISO_8859_6 },
{ "ISO_8859-6:1987",  ESPEAKNG_ENCODING_ISO_8859_6 },
{ "ISO_8859-7",        ESPEAKNG_ENCODING_ISO_8859_7 },
{ "ISO_8859-7:1987",  ESPEAKNG_ENCODING_ISO_8859_7 },
{ "ISO_8859-8",        ESPEAKNG_ENCODING_ISO_8859_8 },
{ "ISO_8859-8:1988",  ESPEAKNG_ENCODING_ISO_8859_8 },
{ "ISO_8859-9",        ESPEAKNG_ENCODING_ISO_8859_9 },
{ "ISO_8859-9:1989",  ESPEAKNG_ENCODING_ISO_8859_9 },
{ "ISO_8859-10",       ESPEAKNG_ENCODING_ISO_8859_10 },
{ "ISO_8859-10:1992", ESPEAKNG_ENCODING_ISO_8859_10 },
{ "ISO_8859-14",       ESPEAKNG_ENCODING_ISO_8859_14 },
{ "ISO_8859-14:1998", ESPEAKNG_ENCODING_ISO_8859_14 },
{ "ISO_8859-15",       ESPEAKNG_ENCODING_ISO_8859_15 },
{ "ISO_8859-16",       ESPEAKNG_ENCODING_ISO_8859_16 },
{ "ISO_8859-16:2001", ESPEAKNG_ENCODING_ISO_8859_16 },
{ "ISO646-US",         ESPEAKNG_ENCODING_US_ASCII },
{ "ISO-10646-UCS-2",  ESPEAKNG_ENCODING_ISO_10646_UCS_2 },
{ "ISO-8859-1",        ESPEAKNG_ENCODING_ISO_8859_1 },
{ "ISO-8859-2",        ESPEAKNG_ENCODING_ISO_8859_2 },
{ "ISO-8859-3",        ESPEAKNG_ENCODING_ISO_8859_3 },
```

```
{ "ISO-8859-4",        ESPEAKNG_ENCODING_ISO_8859_4 },
{ "ISO-8859-5",        ESPEAKNG_ENCODING_ISO_8859_5 },
{ "ISO-8859-6",        ESPEAKNG_ENCODING_ISO_8859_6 },
{ "ISO-8859-7",        ESPEAKNG_ENCODING_ISO_8859_7 },
{ "ISO-8859-8",        ESPEAKNG_ENCODING_ISO_8859_8 },
{ "ISO-8859-9",        ESPEAKNG_ENCODING_ISO_8859_9 },
{ "ISO-8859-10",       ESPEAKNG_ENCODING_ISO_8859_10 },
{ "ISO-8859-11",       ESPEAKNG_ENCODING_ISO_8859_11 },
{ "ISO-8859-13",       ESPEAKNG_ENCODING_ISO_8859_13 },
{ "ISO-8859-14",       ESPEAKNG_ENCODING_ISO_8859_14 },
{ "ISO-8859-15",       ESPEAKNG_ENCODING_ISO_8859_15 },
{ "ISO-8859-16",       ESPEAKNG_ENCODING_ISO_8859_16 },
{ "KOI8-R",            ESPEAKNG_ENCODING_KOI8_R },
{ "Latin-9",           ESPEAKNG_ENCODING_ISO_8859_15 },
{ "TIS-620",           ESPEAKNG_ENCODING_ISO_8859_11 },
{ "US-ASCII",          ESPEAKNG_ENCODING_US_ASCII },
{ "UTF-8",             ESPEAKNG_ENCODING_UTF_8 },
{ "cp367",             ESPEAKNG_ENCODING_US_ASCII },
{ "cp819",             ESPEAKNG_ENCODING_ISO_8859_1 },
{ "csASCII",           ESPEAKNG_ENCODING_US_ASCII },
{ "csISO885913",       ESPEAKNG_ENCODING_ISO_8859_13 },
{ "csISO885914",       ESPEAKNG_ENCODING_ISO_8859_14 },
{ "csISO885915",       ESPEAKNG_ENCODING_ISO_8859_15 },
{ "csISO885916",       ESPEAKNG_ENCODING_ISO_8859_16 },
{ "csISOLatin1",       ESPEAKNG_ENCODING_ISO_8859_1 },
{ "csISOLatin2",       ESPEAKNG_ENCODING_ISO_8859_2 },
{ "csISOLatin3",       ESPEAKNG_ENCODING_ISO_8859_3 },
{ "csISOLatin4",       ESPEAKNG_ENCODING_ISO_8859_4 },
{ "csISOLatin5",       ESPEAKNG_ENCODING_ISO_8859_9 },
{ "csISOLatin6",       ESPEAKNG_ENCODING_ISO_8859_10 },
{ "csISOLatinArabic",  ESPEAKNG_ENCODING_ISO_8859_6 },
{ "csISOLatinCyrillic",ESPEAKNG_ENCODING_ISO_8859_5 },
{ "csISOLatinGreek",   ESPEAKNG_ENCODING_ISO_8859_7 },
{ "csISOLatinHebrew",  ESPEAKNG_ENCODING_ISO_8859_8 },
{ "csKOI8R",           ESPEAKNG_ENCODING_KOI8_R },
{ "csTIS620",          ESPEAKNG_ENCODING_ISO_8859_11 },
{ "csUTF8",            ESPEAKNG_ENCODING_UTF_8 },
```

```
{ "csUnicode",         ESPEAKNG_ENCODING_ISO_10646_UCS_2 },
{ "arabic",            ESPEAKNG_ENCODING_ISO_8859_6 },
{ "cyrillic",          ESPEAKNG_ENCODING_ISO_8859_5 },
{ "greek",             ESPEAKNG_ENCODING_ISO_8859_7 },
{ "greek8",            ESPEAKNG_ENCODING_ISO_8859_7 },
{ "hebrew",            ESPEAKNG_ENCODING_ISO_8859_8 },
{ "iso-celtic",        ESPEAKNG_ENCODING_ISO_8859_14 },
{ "iso-ir-6",          ESPEAKNG_ENCODING_US_ASCII },
{ "iso-ir-100",        ESPEAKNG_ENCODING_ISO_8859_1 },
{ "iso-ir-101",        ESPEAKNG_ENCODING_ISO_8859_2 },
{ "iso-ir-109",        ESPEAKNG_ENCODING_ISO_8859_3 },
{ "iso-ir-110",        ESPEAKNG_ENCODING_ISO_8859_4 },
{ "iso-ir-126",        ESPEAKNG_ENCODING_ISO_8859_7 },
{ "iso-ir-127",        ESPEAKNG_ENCODING_ISO_8859_6 },
{ "iso-ir-138",        ESPEAKNG_ENCODING_ISO_8859_8 },
{ "iso-ir-144",        ESPEAKNG_ENCODING_ISO_8859_5 },
{ "iso-ir-148",        ESPEAKNG_ENCODING_ISO_8859_9 },
{ "iso-ir-157",        ESPEAKNG_ENCODING_ISO_8859_10 },
{ "iso-ir-199",        ESPEAKNG_ENCODING_ISO_8859_14 },
{ "iso-ir-226",        ESPEAKNG_ENCODING_ISO_8859_16 },
{ "latin1",            ESPEAKNG_ENCODING_ISO_8859_1 },
{ "latin2",            ESPEAKNG_ENCODING_ISO_8859_2 },
{ "latin3",            ESPEAKNG_ENCODING_ISO_8859_3 },
{ "latin4",            ESPEAKNG_ENCODING_ISO_8859_4 },
{ "latin5",            ESPEAKNG_ENCODING_ISO_8859_9 },
{ "latin6",            ESPEAKNG_ENCODING_ISO_8859_10 },
{ "latin8",            ESPEAKNG_ENCODING_ISO_8859_14 },
{ "latin10",           ESPEAKNG_ENCODING_ISO_8859_16 },
{ "l1",                ESPEAKNG_ENCODING_ISO_8859_1 },
{ "l2",                ESPEAKNG_ENCODING_ISO_8859_2 },
{ "l3",                ESPEAKNG_ENCODING_ISO_8859_3 },
{ "l4",                ESPEAKNG_ENCODING_ISO_8859_4 },
{ "l5",                ESPEAKNG_ENCODING_ISO_8859_9 },
{ "l6",                ESPEAKNG_ENCODING_ISO_8859_10 },
{ "l8",                ESPEAKNG_ENCODING_ISO_8859_14 },
{ "l10",               ESPEAKNG_ENCODING_ISO_8859_16 },
{ "us",                ESPEAKNG_ENCODING_US_ASCII },
```

```
 { NULL,                ESPEAKNG_ENCODING_UNKNOWN }
};

#pragma GCC visibility push(default)

espeak_ng_ENCODING
espeak_ng_EncodingFromName(const char *encoding)
{
 return LookupMnem(mnem_encoding, encoding);
}

#pragma GCC visibility pop

struct espeak_ng_TEXT_DECODER_
{
 const uint8_t *current;
 const uint8_t *end;

 uint32_t (*get)(espeak_ng_TEXT_DECODER *decoder);
 const uint16_t *codepage;
};

// Reference: http://www.iana.org/go/rfc1345
// Reference:
http://www.unicode.org/Public/MAPPINGS/ISO8859/8859-1.TXT
static const uint16_t ISO_8859_1[0x80] = {
 0x0080, 0x0081, 0x0082, 0x0083, 0x0084, 0x0085, 0x0086, 0x0087,

// Table removed from listning... (100+ lines)

static uint32_t
string_decoder_getc_us_ascii(espeak_ng_TEXT_DECODER *decoder)
{
 uint8_t c = *decoder->current++;
 return (c >= 0x80) ? 0xFFFD : c;
}
```

```
static uint32_t
string_decoder_getc_codepage(espeak_ng_TEXT_DECODER *decoder)
{
 uint8_t c = *decoder->current++;
 return (c >= 0x80) ? decoder->codepage[c - 0x80] : c;
}

static uint32_t
string_decoder_getc_utf_8(espeak_ng_TEXT_DECODER *decoder)
{
 uint8_t  c = *decoder->current++;
 uint32_t ret;
 switch (c & 0xF0)
 {
 // 1-byte UTF-8 sequence
 case 0x00: case 0x10: case 0x20: case 0x30:
 case 0x40: case 0x50: case 0x60: case 0x70:
  return c;
 // UTF-8 tail byte -- invalid in isolation
 case 0x80: case 0x90: case 0xA0: case 0xB0:
  return 0xFFFD;
 // 2-byte UTF-8 sequence
 case 0xC0: case 0xD0:
  if (decoder->current + 1 >= decoder->end) goto eof;
  ret = c & 0x1F;
  if (((c = *decoder->current++) & LEADING_2_BITS) !=
UTF8_TAIL_BITS) goto error;
  ret = (ret << 6) + (c & 0x3F);
  return ret;
 // 3-byte UTF-8 sequence
 case 0xE0:
  if (decoder->current + 2 >= decoder->end) goto eof;
  ret = c & 0x0F;
  if (((c = *decoder->current++) & LEADING_2_BITS) !=
UTF8_TAIL_BITS) goto error;
  ret = (ret << 6) + (c & 0x3F);
  if (((c = *decoder->current++) & LEADING_2_BITS) !=
```

```
UTF8_TAIL_BITS) goto error;
  ret = (ret << 6) + (c & 0x3F);
  return ret;
 // 4-byte UTF-8 sequence
 case 0xF0:
  if (decoder->current + 3 >= decoder->end) goto eof;
  ret = c & 0x0F;
  if (((c = *decoder->current++) & LEADING_2_BITS) !=
UTF8_TAIL_BITS) goto error;
  ret = (ret << 6) + (c & 0x3F);
  if (((c = *decoder->current++) & LEADING_2_BITS) !=
UTF8_TAIL_BITS) goto error;
  ret = (ret << 6) + (c & 0x3F);
  if (((c = *decoder->current++) & LEADING_2_BITS) !=
UTF8_TAIL_BITS) goto error;
  ret = (ret << 6) + (c & 0x3F);
  return (ret <= 0x10FFFF) ? ret : 0xFFFD;
 }
error:
 --decoder->current;
 return 0xFFFD;
eof:
 decoder->current = decoder->end;
 return 0xFFFD;
}

static uint32_t
string_decoder_getc_iso_10646_ucs_2(espeak_ng_TEXT_DECODER
*decoder)
{
 if (decoder->current + 1 >= decoder->end) {
  decoder->current = decoder->end;
  return 0xFFFD;
 }

 uint8_t c1 = *decoder->current++;
 uint8_t c2 = *decoder->current++;
```

```
 return c1 + (c2 << 8);
}

static uint32_t
string_decoder_getc_wchar(espeak_ng_TEXT_DECODER *decoder)
{
 wchar_t c = *(const wchar_t *)decoder->current;
 decoder->current += sizeof(wchar_t);
 return c;
}

static uint32_t
string_decoder_getc_auto(espeak_ng_TEXT_DECODER *decoder)
{
 const uint8_t *ptr = decoder->current;
 uint32_t c = string_decoder_getc_utf_8(decoder);
 if (c == 0xFFFD) {
  decoder->get = string_decoder_getc_codepage;
  decoder->current = ptr;
  c = decoder->get(decoder);
 }
 return c;
}

static uint32_t
null_decoder_getc(espeak_ng_TEXT_DECODER *decoder)
{
 (void)decoder; // unused parameter
 return 0;
}

typedef struct
{
 uint32_t (*get)(espeak_ng_TEXT_DECODER *decoder);
 const uint16_t *codepage;
} encoding_t;
```

```
static const encoding_t string_decoders[] = {
 { NULL, NULL },
 { string_decoder_getc_us_ascii, NULL },
 { string_decoder_getc_codepage, ISO_8859_1 },
 { string_decoder_getc_codepage, ISO_8859_2 },
 { string_decoder_getc_codepage, ISO_8859_3 },
 { string_decoder_getc_codepage, ISO_8859_4 },
 { string_decoder_getc_codepage, ISO_8859_5 },
 { string_decoder_getc_codepage, ISO_8859_6 },
 { string_decoder_getc_codepage, ISO_8859_7 },
 { string_decoder_getc_codepage, ISO_8859_8 },
 { string_decoder_getc_codepage, ISO_8859_9 },
 { string_decoder_getc_codepage, ISO_8859_10 },
 { string_decoder_getc_codepage, ISO_8859_11 },
 // ISO-8859-12 is not a valid encoding.
 { string_decoder_getc_codepage, ISO_8859_13 },
 { string_decoder_getc_codepage, ISO_8859_14 },
 { string_decoder_getc_codepage, ISO_8859_15 },
 { string_decoder_getc_codepage, ISO_8859_16 },
 { string_decoder_getc_codepage, KOI8_R },
 { string_decoder_getc_codepage, ISCII },
 { string_decoder_getc_utf_8, NULL },
 { string_decoder_getc_iso_10646_ucs_2, NULL },
};

#pragma GCC visibility push(default)

espeak_ng_TEXT_DECODER *
create_text_decoder(void)
{
 espeak_ng_TEXT_DECODER *decoder =
malloc(sizeof(espeak_ng_TEXT_DECODER));
 if (!decoder) return NULL;

 decoder->current = NULL;
 decoder->end = NULL;
 decoder->get = NULL;
```

```
 decoder->codepage = NULL;
 return decoder;
}

void
destroy_text_decoder(espeak_ng_TEXT_DECODER *decoder)
{
 if (decoder) free(decoder);
}

espeak_ng_STATUS
text_decoder_decode_string(espeak_ng_TEXT_DECODER *decoder,
                           const char *string,
                           int length,
                           espeak_ng_ENCODING encoding)
{
 if (encoding > ESPEAKNG_ENCODING_ISO_10646_UCS_2)
  return ENS_UNKNOWN_TEXT_ENCODING;

 const encoding_t *enc = string_decoders + encoding;
 if (enc->get == NULL)
  return ENS_UNKNOWN_TEXT_ENCODING;

 if (length < 0) length = string ? strlen(string) + 1 : 0;

 decoder->get = string ? enc->get : null_decoder_getc;
 decoder->codepage = enc->codepage;
 decoder->current = (const uint8_t *)string;
 decoder->end = (const uint8_t *)(string ? string + length :
string);
 return ENS_OK;
}

espeak_ng_STATUS
text_decoder_decode_string_auto(espeak_ng_TEXT_DECODER *decoder,
                                const char *string,
                                int length,
```

```
                                espeak_ng_ENCODING encoding)
{
 if (encoding > ESPEAKNG_ENCODING_ISO_10646_UCS_2)
  return ENS_UNKNOWN_TEXT_ENCODING;

 const encoding_t *enc = string_decoders + encoding;
 if (enc->get == NULL)
  return ENS_UNKNOWN_TEXT_ENCODING;

 if (length < 0) length = string ? strlen(string) + 1 : 0;

 decoder->get = string ? string_decoder_getc_auto :
null_decoder_getc;
 decoder->codepage = enc->codepage;
 decoder->current = (const uint8_t *)string;
 decoder->end = (const uint8_t *)(string ? string + length :
string);
 return ENS_OK;
}

espeak_ng_STATUS
text_decoder_decode_wstring(espeak_ng_TEXT_DECODER *decoder,
                            const wchar_t *string,
                            int length)
{
 if (length < 0) length = string ? wcslen(string) + 1 : 0;

 decoder->get = string ? string_decoder_getc_wchar :
null_decoder_getc;
 decoder->codepage = NULL;
 decoder->current = (const uint8_t *)string;
 decoder->end = (const uint8_t *)(string ? string + length :
string);
 return ENS_OK;
}

espeak_ng_STATUS
```

```
text_decoder_decode_string_multibyte(espeak_ng_TEXT_DECODER
*decoder,
                                     const void *input,
                                     espeak_ng_ENCODING encoding,
                                     int flags)
{
 switch (flags & 7)
 {
 case espeakCHARS_WCHAR:
  return text_decoder_decode_wstring(decoder, (const wchar_t
*)input, -1);
 case espeakCHARS_AUTO:
  return text_decoder_decode_string_auto(decoder, (const char
*)input, -1, encoding);
 case espeakCHARS_UTF8:
  return text_decoder_decode_string(decoder, (const char *)input,
-1, ESPEAKNG_ENCODING_UTF_8);
 case espeakCHARS_8BIT:
  return text_decoder_decode_string(decoder, (const char *)input,
-1, encoding);
 case espeakCHARS_16BIT:
  return text_decoder_decode_string(decoder, (const char *)input,
-1, ESPEAKNG_ENCODING_ISO_10646_UCS_2);
 default:
  return ENS_UNKNOWN_TEXT_ENCODING;
 }
}

int
text_decoder_eof(espeak_ng_TEXT_DECODER *decoder)
{
 return decoder->current == decoder->end;
}

uint32_t
text_decoder_getc(espeak_ng_TEXT_DECODER *decoder)
{
```

```
 return decoder->get(decoder);
}

uint32_t
text_decoder_peekc(espeak_ng_TEXT_DECODER *decoder)
{
 if (decoder->current == decoder->end) return 0;

 const uint8_t *current = decoder->current;
 uint32_t c = decoder->get(decoder);
 decoder->current = current;
 return c;
}

const void *
text_decoder_get_buffer(espeak_ng_TEXT_DECODER *decoder)
{
 if (text_decoder_eof(decoder))
  return NULL;
 return decoder->current;
}

#pragma GCC visibility pop
```

Chapter 64

./src/libespeak-ng/setlengths.c

```
#include "config.h"

#include <errno.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "readclause.h"
#include "setlengths.h"
#include "synthdata.h"
#include "wavegen.h"

#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

extern int saved_parameters[];
```

```
// convert from words-per-minute to internal speed factor
// Use this to calibrate speed for wpm 80-450 (espeakRATE_MINIMUM
- espeakRATE_MAXIMUM)
static unsigned char speed_lookup[] = {
 255, 255, 255, 255, 255, //  80
 253, 249, 245, 242, 238, //  85
 235, 232, 228, 225, 222, //  90
 218, 216, 213, 210, 207, //  95
 204, 201, 198, 196, 193, // 100
 191, 188, 186, 183, 181, // 105
 179, 176, 174, 172, 169, // 110
 168, 165, 163, 161, 159, // 115
 158, 155, 153, 152, 150, // 120
 148, 146, 145, 143, 141, // 125
 139, 137, 136, 135, 133, // 130
 131, 130, 129, 127, 126, // 135
 124, 123, 122, 120, 119, // 140
 118, 117, 115, 114, 113, // 145
 112, 111, 110, 109, 107, // 150
 106, 105, 104, 103, 102, // 155
 101, 100,  99,  98,  97, // 160
  96,  95,  94,  93,  92, // 165
  91,  90,  89,  89,  88, // 170
  87,  86,  85,  84,  83, // 175
  82,  82,  81,  80,  80, // 180
  79,  78,  77,  76,  76, // 185
  75,  75,  74,  73,  72, // 190
  71,  71,  70,  69,  69, // 195
  68,  67,  67,  66,  66, // 200
  65,  64,  64,  63,  62, // 205
  62,  61,  61,  60,  59, // 210
  59,  58,  58,  57,  57, // 215
  56,  56,  55,  54,  54, // 220
  53,  53,  52,  52,  52, // 225
  51,  50,  50,  49,  49, // 230
  48,  48,  47,  47,  46, // 235
```

```
  46,  46,  45,  45,  44, // 240
  44,  44,  43,  43,  42, // 245
  41,  40,  40,  40,  39, // 250
  39,  39,  38,  38,  38, // 255
  37,  37,  37,  36,  36, // 260
  35,  35,  35,  35,  34, // 265
  34,  34,  33,  33,  33, // 270
  32,  32,  31,  31,  31, // 275
  30,  30,  30,  29,  29, // 280
  29,  29,  28,  28,  27, // 285
  27,  27,  27,  26,  26, // 290
  26,  26,  25,  25,  25, // 295
  24,  24,  24,  24,  23, // 300
  23,  23,  23,  22,  22, // 305
  22,  21,  21,  21,  21, // 310
  20,  20,  20,  20,  19, // 315
  19,  19,  18,  18,  17, // 320
  17,  17,  16,  16,  16, // 325
  16,  16,  16,  15,  15, // 330
  15,  15,  14,  14,  14, // 335
  13,  13,  13,  12,  12, // 340
  12,  12,  11,  11,  11, // 345
  11,  10,  10,  10,   9, // 350
   9,   9,   8,   8,   8, // 355
};

// speed_factor1 adjustments for speeds 350 to 374: pauses
static unsigned char pause_factor_350[] = {
 22, 22, 22, 22, 22, 22, 22, 21, 21, 21, // 350
 21, 20, 20, 19, 19, 18, 17, 16, 15, 15, // 360
 15, 15, 15, 15, 15                      // 370
};

// wav_factor adjustments for speeds 350 to 450
// Use this to calibrate speed for wpm 350-450
static unsigned char wav_factor_350[] = {
 120, 121, 120, 119, 119, // 350
```

```
 118, 118, 117, 116, 116, // 355
 115, 114, 113, 112, 112, // 360
 111, 111, 110, 109, 108, // 365
 107, 106, 106, 104, 103, // 370
 103, 102, 102, 102, 101, // 375
 101,  99,  98,  98,  97, // 380
  96,  96,  95,  94,  93, // 385
  91,  90,  91,  90,  89, // 390
  88,  86,  85,  86,  85, // 395
  85,  84,  82,  81,  80, // 400
  79,  77,  78,  78,  76, // 405
  77,  75,  75,  74,  73, // 410
  71,  72,  70,  69,  69, // 415
  69,  67,  65,  64,  63, // 420
  63,  63,  61,  61,  59, // 425
  59,  59,  58,  56,  57, // 430
  58,  56,  54,  53,  52, // 435
  52,  53,  52,  52,  50, // 440
  48,  47,  47,  45,  46, // 445
  45                      // 450
};

static int speed1 = 130;
static int speed2 = 121;
static int speed3 = 118;

#if HAVE_SONIC_H

void SetSpeed(int control)
{
 int x;
 int s1;
 int wpm;
 int wpm2;
 int wpm_value;
 double sonic;
```

```
 speed.loud_consonants = 0;
 speed.min_sample_len = espeakRATE_MAXIMUM;
 speed.lenmod_factor = 110; // controls the effect of
FRFLAG_LEN_MOD reduce length change
 speed.lenmod2_factor = 100;
 speed.min_pause = 5;

 wpm = embedded_value[EMBED_S];
 if (control == 2)
  wpm = embedded_value[EMBED_S2];

 wpm_value = wpm;

 if (voice->speed_percent > 0)
  wpm = (wpm * voice->speed_percent)/100;

 if (control & 2)
  DoSonicSpeed(1 * 1024);
 if ((wpm_value >= espeakRATE_MAXIMUM) || ((wpm_value >
speed.fast_settings[0]) && (wpm > 350))) {
  wpm2 = wpm;
  wpm = espeakRATE_NORMAL;

  // set special eSpeak speed parameters for Sonic use
  // The eSpeak output will be speeded up by at least x2
  x = 73;
  if (control & 1) {
   speed1 = (x * voice->speedf1)/256;
   speed2 = (x * voice->speedf2)/256;
   speed3 = (x * voice->speedf3)/256;
  }
  if (control & 2) {
   sonic = ((double)wpm2)/wpm;
   DoSonicSpeed((int)(sonic * 1024));
   speed.pause_factor = 85;
   speed.clause_pause_factor = espeakRATE_MINIMUM;
   speed.min_pause = 22;
```

```
   speed.min_sample_len = espeakRATE_MAXIMUM*2;
   speed.wav_factor = 211;
   speed.lenmod_factor = 210;
   speed.lenmod2_factor = 170;
  }
  return;
 }

 if (wpm > espeakRATE_MAXIMUM)
  wpm = espeakRATE_MAXIMUM;

 if (wpm > 360)
  speed.loud_consonants = (wpm - 360) / 8;

 wpm2 = wpm;
 if (wpm > 359) wpm2 = 359;
 if (wpm < espeakRATE_MINIMUM) wpm2 = espeakRATE_MINIMUM;
 x = speed_lookup[wpm2-espeakRATE_MINIMUM];

 if (wpm >= 380)
  x = 7;
 if (wpm >= 400)
  x = 6;

 if (control & 1) {
  // set speed factors for different syllable positions within a
word
  // these are used in CalcLengths()
  speed1 = (x * voice->speedf1)/256;
  speed2 = (x * voice->speedf2)/256;
  speed3 = (x * voice->speedf3)/256;

  if (x <= 7) {
   speed1 = x;
   speed2 = speed3 = x - 1;
  }
 }
```

```
 if (control & 2) {
  // these are used in synthesis file

  if (wpm > 350) {
   speed.lenmod_factor = 85 - (wpm - 350) / 3;
   speed.lenmod2_factor = 60 - (wpm - 350) / 8;
  } else if (wpm > 250) {
   speed.lenmod_factor = 110 - (wpm - 250)/4;
   speed.lenmod2_factor = 110 - (wpm - 250)/2;
  }

  s1 = (x * voice->speedf1)/256;

  if (wpm >= 170)
   speed.wav_factor = 110 + (150*s1)/128; // reduced speed
adjustment, used for playing recorded sounds
  else
   speed.wav_factor = 128 + (128*s1)/130; // = 215 at 170 wpm

  if (wpm >= 350)
   speed.wav_factor = wav_factor_350[wpm-350];

  if (wpm >= 390) {
   speed.min_sample_len = espeakRATE_MAXIMUM - (wpm - 400)/2;
   if (wpm > 440)
    speed.min_sample_len = 420 - (wpm - 440);
  }

  // adjust for different sample rates
  speed.min_sample_len = (speed.min_sample_len *
samplerate_native) / 22050;

  speed.pause_factor = (256 * s1)/115; // full speed adjustment,
used for pause length
  speed.clause_pause_factor = 0;
```

```
  if (wpm > 430)
   speed.pause_factor = 12;
  else if (wpm > 400)
   speed.pause_factor = 13;
  else if (wpm > 374)
   speed.pause_factor = 14;
  else if (wpm > 350)
   speed.pause_factor = pause_factor_350[wpm - 350];

  if (speed.clause_pause_factor == 0) {
   // restrict the reduction of pauses between clauses
   if ((speed.clause_pause_factor = speed.pause_factor) < 16)
    speed.clause_pause_factor = 16;
  }
 }
}

#else

void SetSpeed(int control)
{
 // This is the earlier version of SetSpeed() before sonic speed-
up was added
 int x;
 int s1;
 int wpm;
 int wpm2;

 speed.loud_consonants = 0;
 speed.min_sample_len = espeakRATE_MAXIMUM;
 speed.lenmod_factor = 110; // controls the effect of
FRFLAG_LEN_MOD reduce length change
 speed.lenmod2_factor = 100;

 wpm = embedded_value[EMBED_S];
 if (control == 2)
  wpm = embedded_value[EMBED_S2];
```

```
 if (voice->speed_percent > 0)
  wpm = (wpm * voice->speed_percent)/100;
 if (wpm > espeakRATE_MAXIMUM)
  wpm = espeakRATE_MAXIMUM;

 if (wpm > 360)
  speed.loud_consonants = (wpm - 360) / 8;

 wpm2 = wpm;
 if (wpm > 359) wpm2 = 359;
 if (wpm < espeakRATE_MINIMUM) wpm2 = espeakRATE_MINIMUM;
 x = speed_lookup[wpm2-espeakRATE_MINIMUM];

 if (wpm >= 380)
  x = 7;
 if (wpm >= 400)
  x = 6;

 if (control & 1) {
  // set speed factors for different syllable positions within a
word
  // these are used in CalcLengths()
  speed1 = (x * voice->speedf1)/256;
  speed2 = (x * voice->speedf2)/256;
  speed3 = (x * voice->speedf3)/256;

  if (x <= 7) {
   speed1 = x;
   speed2 = speed3 = x - 1;
  }
 }

 if (control & 2) {
  // these are used in synthesis file

  if (wpm > 350) {
```

```
    speed.lenmod_factor = 85 - (wpm - 350) / 3;
    speed.lenmod2_factor = 60 - (wpm - 350) / 8;
  } else if (wpm > 250) {
    speed.lenmod_factor = 110 - (wpm - 250)/4;
    speed.lenmod2_factor = 110 - (wpm - 250)/2;
  }

  s1 = (x * voice->speedf1)/256;

  if (wpm >= 170)
    speed.wav_factor = 110 + (150*s1)/128; // reduced speed
adjustment, used for playing recorded sounds
  else
    speed.wav_factor = 128 + (128*s1)/130; // = 215 at 170 wpm

  if (wpm >= 350)
    speed.wav_factor = wav_factor_350[wpm-350];

  if (wpm >= 390) {
    speed.min_sample_len = espeakRATE_MAXIMUM - (wpm - 400)/2;
    if (wpm > 440)
      speed.min_sample_len = 420 - (wpm - 440);
  }

  speed.pause_factor = (256 * s1)/115; // full speed adjustment,
used for pause length
  speed.clause_pause_factor = 0;

  if (wpm > 430)
    speed.pause_factor = 12;
  else if (wpm > 400)
    speed.pause_factor = 13;
  else if (wpm > 374)
    speed.pause_factor = 14;
  else if (wpm > 350)
    speed.pause_factor = pause_factor_350[wpm - 350];
```

```
  if (speed.clause_pause_factor == 0) {
   // restrict the reduction of pauses between clauses
   if ((speed.clause_pause_factor = speed.pause_factor) < 16)
    speed.clause_pause_factor = 16;
  }
 }
}

#endif

espeak_ng_STATUS SetParameter(int parameter, int value, int
relative)
{
 // parameter: reset-all, amp, pitch, speed, linelength,
expression, capitals, number grouping
 // relative 0=absolute  1=relative

 int new_value = value;
 int default_value;
 extern const int param_defaults[N_SPEECH_PARAM];

 if (relative) {
  if (parameter < 5) {
   default_value = param_defaults[parameter];
   new_value = default_value + (default_value * value)/100;
  }
 }
 param_stack[0].parameter[parameter] = new_value;
 saved_parameters[parameter] = new_value;

 switch (parameter)
 {
 case espeakRATE:
  embedded_value[EMBED_S] = new_value;
  embedded_value[EMBED_S2] = new_value;
  SetSpeed(3);
  break;
```

```
 case espeakVOLUME:
  embedded_value[EMBED_A] = new_value;
  GetAmplitude();
  break;
 case espeakPITCH:
  if (new_value > 99) new_value = 99;
  if (new_value < 0) new_value = 0;
  embedded_value[EMBED_P] = new_value;
  break;
 case espeakRANGE:
  if (new_value > 99) new_value = 99;
  embedded_value[EMBED_R] = new_value;
  break;
 case espeakLINELENGTH:
  option_linelength = new_value;
  break;
 case espeakWORDGAP:
  option_wordgap = new_value;
  break;
 case espeakINTONATION:
  if ((new_value & 0xff) != 0)
   translator->langopts.intonation_group = new_value & 0xff;
  option_tone_flags = new_value;
  break;
 default:
  return EINVAL;
 }
 return ENS_OK;
}

static void DoEmbedded2(int *embix)
{
 // There were embedded commands in the text at this point

 unsigned int word;

 do {
```

```
  word = embedded_list[(*embix)++];

  if ((word & 0x1f) == EMBED_S) {
   // speed
   SetEmbedded(word & 0x7f, word >> 8); // adjusts
embedded_value[EMBED_S]
   SetSpeed(1);
  }
 } while ((word & 0x80) == 0);
}

void CalcLengths(Translator *tr)
{
 int ix;
 int ix2;
 PHONEME_LIST *prev;
 PHONEME_LIST *next;
 PHONEME_LIST *next2;
 PHONEME_LIST *next3;
 PHONEME_LIST *p;
 PHONEME_LIST *p2;

 int stress;
 int type;
 static int more_syllables = 0;
 bool pre_sonorant = false;
 bool pre_voiced = false;
 int last_pitch = 0;
 int pitch_start;
 int length_mod;
 int next2type;
 int len;
 int env2;
 int end_of_clause;
 int embedded_ix = 0;
 int min_drop;
 int pitch1;
```

```
int emphasized;
int tone_mod;
unsigned char *pitch_env = NULL;
PHONEME_DATA phdata_tone;

for (ix = 1; ix < n_phoneme_list; ix++) {
 prev = &phoneme_list[ix-1];
 p = &phoneme_list[ix];
 stress = p->stresslevel & 0x7;
 emphasized = p->stresslevel & 0x8;

 next = &phoneme_list[ix+1];

 if (p->synthflags & SFLAG_EMBEDDED)
  DoEmbedded2(&embedded_ix);

 type = p->type;
 if (p->synthflags & SFLAG_SYLLABLE)
  type = phVOWEL;

 switch (type)
 {
 case phPAUSE:
  last_pitch = 0;
  break;
 case phSTOP:
  last_pitch = 0;
  if (prev->type == phFRICATIVE)
   p->prepause = 25;
  else if ((more_syllables > 0) || (stress < 4))
   p->prepause = 48;
  else
   p->prepause = 60;

  if (prev->type == phSTOP)
   p->prepause = 60;
```

```
   if ((tr->langopts.word_gap & 0x10) && (p->newword))
    p->prepause = 60;

   if (p->ph->phflags & phLENGTHENSTOP)
    p->prepause += 30;

   if (p->synthflags & SFLAG_LENGTHEN)
    p->prepause += tr->langopts.long_stop;
   break;
  case phVFRICATIVE:
  case phFRICATIVE:
   if (p->newword) {
    if ((prev->type == phVOWEL) && (p->ph->phflags & phNOPAUSE))
{
    } else
     p->prepause = 15;
   }

   if (next->type == phPAUSE && prev->type == phNASAL &&
!(p->ph->phflags&phVOICELESS))
    p->prepause = 25;

   if (prev->ph->phflags & phBRKAFTER)
    p->prepause = 30;

   if ((tr->langopts.word_gap & 0x10) && (p->newword))
    p->prepause = 30;

   if ((p->ph->phflags & phSIBILANT) && next->type == phSTOP &&
!next->newword) {
    if (prev->type == phVOWEL)
     p->length = 200; // ?? should do this if it's from a prefix
    else
     p->length = 150;
   } else
    p->length = 256;
```

```
   if (type == phVFRICATIVE) {
    if (next->type == phVOWEL)
     pre_voiced = true;
    if ((prev->type == phVOWEL) || (prev->type == phLIQUID))
     p->length = (255 + prev->length)/2;
   }
   break;
  case phVSTOP:
   if (prev->type == phVFRICATIVE || prev->type == phFRICATIVE ||
(prev->ph->phflags & phSIBILANT) || (prev->type == phLIQUID))
    p->prepause = 30;

   if (next->type == phVOWEL || next->type == phLIQUID) {
    if ((next->type == phVOWEL) || !next->newword)
     pre_voiced = true;

    p->prepause = 40;

    if (prev->type == phVOWEL) {
     p->prepause = 0; // use murmur instead to link from the
preceding vowel
    } else if (prev->type == phPAUSE) {
     // reduce by the length of the preceding pause
     if (prev->length < p->prepause)
      p->prepause -= prev->length;
     else
      p->prepause = 0;
    } else if (p->newword == 0) {
     if (prev->type == phLIQUID)
      p->prepause = 20;
     if (prev->type == phNASAL)
      p->prepause = 12;

     if (prev->type == phSTOP && !(prev->ph->phflags &
phVOICELESS))
      p->prepause = 0;
    }
```

```
   }
   if ((tr->langopts.word_gap & 0x10) && (p->newword) &&
(p->prepause < 20))
    p->prepause = 20;
   break;
  case phLIQUID:
  case phNASAL:
   p->amp = tr->stress_amps[0]; // unless changed later
   p->length = 256; //  TEMPORARY

   if (p->newword) {
    if (prev->type == phLIQUID)
     p->prepause = 25;
    if (prev->type == phVOWEL) {
     if (!(p->ph->phflags & phNOPAUSE))
      p->prepause = 12;
    }
   }

   if (next->type == phVOWEL)
    pre_sonorant = true;
   else {
    p->pitch2 = last_pitch;

    if ((prev->type == phVOWEL) || (prev->type == phLIQUID)) {
     p->length = prev->length;

     if (p->type == phLIQUID)
      p->length = speed1;

     if (next->type == phVSTOP)
      p->length = (p->length * 160)/100;
     if (next->type == phVFRICATIVE)
      p->length = (p->length * 120)/100;
    } else {
     for (ix2 = ix; ix2 < n_phoneme_list; ix2++) {
      if (phoneme_list[ix2].type == phVOWEL) {
```

```
        p->pitch2 = phoneme_list[ix2].pitch2;
        break;
       }
      }
     }

     p->pitch1 = p->pitch2-16;
     if (p->pitch2 < 16)
      p->pitch1 = 0;
     p->env = PITCHfall;
     pre_voiced = false;
    }
    break;
   case phVOWEL:
    min_drop = 0;
    next2 = &phoneme_list[ix+2];
    next3 = &phoneme_list[ix+3];

    if (stress > 7) stress = 7;

    if (stress <= 1)
     stress = stress ^ 1; // swap diminished and unstressed (until
we swap stress_amps,stress_lengths in tr_languages)
    if (pre_sonorant)
     p->amp = tr->stress_amps[stress]-1;
    else
     p->amp = tr->stress_amps[stress];

    if (emphasized)
     p->amp = 25;

    if (ix >= (n_phoneme_list-3)) {
     // last phoneme of a clause, limit its amplitude
     if (p->amp > tr->langopts.param[LOPT_MAXAMP_EOC])
      p->amp = tr->langopts.param[LOPT_MAXAMP_EOC];
    }
```

```
   // is the last syllable of a word ?
   more_syllables = 0;
   end_of_clause = 0;
   for (p2 = p+1; p2->newword == 0; p2++) {
    if ((p2->type == phVOWEL) && !(p2->ph->phflags &
phNONSYLLABIC))
     more_syllables++;

    if (p2->ph->code == phonPAUSE_CLAUSE)
     end_of_clause = 2;
   }
   if (p2->ph->code == phonPAUSE_CLAUSE)
    end_of_clause = 2;

   if ((p2->newword & PHLIST_END_OF_CLAUSE) && (more_syllables ==
0))
    end_of_clause = 2;

   // calc length modifier
   if ((next->ph->code == phonPAUSE_VSHORT) && (next2->type ==
phPAUSE)) {
    // if PAUSE_VSHORT is followed by a pause, then use that
    next = next2;
    next2 = next3;
    next3 = &phoneme_list[ix+4];
   }

   next2type = next2->ph->length_mod;
   if (more_syllables == 0) {
    if (next->newword || next2->newword) {
     // don't use 2nd phoneme over a word boundary, unless it's a
pause
     if (next2type != 1)
      next2type = 0;
    }

    len = tr->langopts.length_mods0[next2type *10+
```

```
next->ph->length_mod];

    if ((next->newword) && (tr->langopts.word_gap & 0x20)) {
     // consider as a pause + first phoneme of the next word
     length_mod = (len +
tr->langopts.length_mods0[next->ph->length_mod *10+ 1])/2;
    } else
     length_mod = len;
   } else {
    length_mod = tr->langopts.length_mods[next2type *10+
next->ph->length_mod];

    if ((next->type == phNASAL) && (next2->type == phSTOP ||
next2->type == phVSTOP) && (next3->ph->phflags & phVOICELESS))
     length_mod -= 15;
   }

   if (more_syllables == 0)
    length_mod *= speed1;
   else if (more_syllables == 1)
    length_mod *= speed2;
   else
    length_mod *= speed3;

   length_mod = length_mod / 128;

   if (length_mod < 8)
    length_mod = 8; // restrict how much lengths can be reduced

   if (stress >= 7) {
    // tonic syllable, include a constant component so it doesn't
decrease directly with speed
    length_mod += tr->langopts.lengthen_tonic;
    if (emphasized)
     length_mod += (tr->langopts.lengthen_tonic/2);
   } else if (emphasized)
    length_mod += tr->langopts.lengthen_tonic;
```

```
   if ((len = tr->stress_lengths[stress]) == 0)
    len = tr->stress_lengths[6];

   length_mod = length_mod * len;

   if (p->tone_ph != 0) {
    if ((tone_mod = phoneme_tab[p->tone_ph]->std_length) > 0) {
     // a tone phoneme specifies a percentage change to the
length
     length_mod = (length_mod * tone_mod) / 100;
    }
   }

   if ((end_of_clause == 2) && !(tr->langopts.stress_flags &
S_NO_EOC_LENGTHEN)) {
    // this is the last syllable in the clause, lengthen it -
more for short vowels
    len = (p->ph->std_length * 2);
    if (tr->langopts.stress_flags & S_EO_CLAUSE1)
     len = 200; // don't lengthen short vowels more than long
vowels at end-of-clause
    length_mod = length_mod * (256 + (280 - len)/3)/256;
   }

   if (length_mod > tr->langopts.max_lengthmod*speed1) {
    // limit the vowel length adjustment for some languages
    length_mod = (tr->langopts.max_lengthmod*speed1);
   }

   length_mod = length_mod / 128;

   if (p->type != phVOWEL) {
    length_mod = 256; // syllabic consonant
    min_drop = 16;
   }
   p->length = length_mod;
```

```
   if (p->env >= (N_ENVELOPE_DATA-1)) {
    fprintf(stderr, "espeak: Bad intonation data\n");
    p->env = 0;
   }

   // pre-vocalic part
   // set last-pitch
   env2 = p->env + 1; // version for use with preceding semi-
vowel

   if (p->tone_ph != 0) {
    InterpretPhoneme2(p->tone_ph, &phdata_tone);
    pitch_env = GetEnvelope(phdata_tone.pitch_env);
   } else
    pitch_env = envelope_data[env2];

   pitch_start = p->pitch1 +
((p->pitch2-p->pitch1)*pitch_env[0])/256;

   if (pre_sonorant || pre_voiced) {
    // set pitch for pre-vocalic part
    if (pitch_start == 255)
     last_pitch = pitch_start; // pitch is not set

    if (pitch_start - last_pitch > 16)
     last_pitch = pitch_start - 16;

    prev->pitch1 = last_pitch;
    prev->pitch2 = pitch_start;
    if (last_pitch < pitch_start) {
     prev->env = PITCHrise;
     p->env = env2;
    } else
     prev->env = PITCHfall;

    prev->length = length_mod;
```

```
    prev->amp = p->amp;
    if ((prev->type != phLIQUID) && (prev->amp > 18))
     prev->amp = 18;
   }

   // vowel & post-vocalic part
   next->synthflags &= ~SFLAG_SEQCONTINUE;
   if (next->type == phNASAL && next2->type != phVOWEL)
    next->synthflags |= SFLAG_SEQCONTINUE;

   if (next->type == phLIQUID) {
    next->synthflags |= SFLAG_SEQCONTINUE;

    if (next2->type == phVOWEL)
     next->synthflags &= ~SFLAG_SEQCONTINUE;

    if (next2->type != phVOWEL) {
     if (next->ph->mnemonic == ('/'*256+'r'))
      next->synthflags &= ~SFLAG_SEQCONTINUE;
    }
   }

   if ((min_drop > 0) && ((p->pitch2 - p->pitch1) < min_drop)) {
    pitch1 = p->pitch2 - min_drop;
    if (pitch1 < 0)
     pitch1 = 0;
    p->pitch1 = pitch1;
   }

   last_pitch = p->pitch1 +
((p->pitch2-p->pitch1)*envelope_data[p->env][127])/256;
   pre_sonorant = false;
   pre_voiced = false;
   break;
  }
 }
```

```
}
```

Chapter 65

./src/libespeak-ng/fifo.c

```
// This source file is only used for asynchronious modes

#include "config.h"

#include <assert.h>
#include <errno.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

#include <espeak-ng/espeak_ng.h>

#include "speech.h"
#include "espeak_command.h"
#include "fifo.h"
#include "event.h"
```

```
#ifdef USE_ASYNC

// my_mutex: protects my_thread_is_talking,
// my_stop_is_required, and the command fifo
static pthread_mutex_t my_mutex;
static bool my_command_is_running = false;
static pthread_cond_t my_cond_command_is_running;
static bool my_stop_is_required = false;
static bool my_terminate_is_required = 0;

// my_thread: reads commands from the fifo, and runs them.
static pthread_t my_thread;

static pthread_cond_t my_cond_start_is_required;
static bool my_start_is_required = false;

static pthread_cond_t my_cond_stop_is_acknowledged;
static bool my_stop_is_acknowledged = false;

static void *say_thread(void *);

static espeak_ng_STATUS push(t_espeak_command *the_command);
static t_espeak_command *pop(void);
static void init(int process_parameters);
static int node_counter = 0;

enum {
 MAX_NODE_COUNTER = 400,
 INACTIVITY_TIMEOUT = 50, // in ms, check that the stream is
inactive
 MAX_INACTIVITY_CHECK = 2
};

void fifo_init()
{
 // security
 pthread_mutex_init(&my_mutex, (const pthread_mutexattr_t
```

```
*)NULL);
 init(0);

 assert(-1 != pthread_cond_init(&my_cond_command_is_running,
NULL));
 assert(-1 != pthread_cond_init(&my_cond_start_is_required,
NULL));
 assert(-1 != pthread_cond_init(&my_cond_stop_is_acknowledged,
NULL));

 pthread_attr_t a_attrib;
 if (pthread_attr_init(&a_attrib)
     || pthread_attr_setdetachstate(&a_attrib,
PTHREAD_CREATE_JOINABLE)
     || pthread_create(&my_thread,
                       &a_attrib,
                       say_thread,
                       (void *)NULL)) {
  assert(0);
 }

 pthread_attr_destroy(&a_attrib);

 // leave once the thread is actually started
 assert(-1 != pthread_mutex_lock(&my_mutex));
 while (my_stop_is_acknowledged == false) {
  while ((pthread_cond_wait(&my_cond_stop_is_acknowledged,
&my_mutex) == -1) && errno == EINTR)
   ;
 }
 my_stop_is_acknowledged = false;
 pthread_mutex_unlock(&my_mutex);
}

espeak_ng_STATUS fifo_add_command(t_espeak_command *the_command)
{
 espeak_ng_STATUS status;
```

```
 if ((status = pthread_mutex_lock(&my_mutex)) != ENS_OK)
  return status;

 if ((status = push(the_command)) != ENS_OK) {
  pthread_mutex_unlock(&my_mutex);
  return status;
 }

 my_start_is_required = true;
 pthread_cond_signal(&my_cond_start_is_required);

 while (my_start_is_required && !my_command_is_running) {
  if((status = pthread_cond_wait(&my_cond_command_is_running,
&my_mutex)) != ENS_OK && errno != EINTR) {
   pthread_mutex_unlock(&my_mutex);
   return status;
  }
 }
 if ((status = pthread_mutex_unlock(&my_mutex)) != ENS_OK)
  return status;

 return ENS_OK;
}

espeak_ng_STATUS fifo_add_commands(t_espeak_command *command1,
t_espeak_command *command2)
{
 espeak_ng_STATUS status;
 if ((status = pthread_mutex_lock(&my_mutex)) != ENS_OK)
  return status;

 if (node_counter+1 >= MAX_NODE_COUNTER) {
  pthread_mutex_unlock(&my_mutex);
  return ENS_FIFO_BUFFER_FULL;
 }

 if ((status = push(command1)) != ENS_OK) {
```

```
  pthread_mutex_unlock(&my_mutex);
  return status;
 }

 if ((status = push(command2)) != ENS_OK) {
  pthread_mutex_unlock(&my_mutex);
  return status;
 }

 my_start_is_required = true;
 pthread_cond_signal(&my_cond_start_is_required);

 while (my_start_is_required && !my_command_is_running) {
  if((status = pthread_cond_wait(&my_cond_command_is_running,
&my_mutex)) != ENS_OK && errno != EINTR) {
   pthread_mutex_unlock(&my_mutex);
   return status;
  }
 }
 if ((status = pthread_mutex_unlock(&my_mutex)) != ENS_OK)
  return status;

 return ENS_OK;
}

espeak_ng_STATUS fifo_stop()
{
 espeak_ng_STATUS status;
 if ((status = pthread_mutex_lock(&my_mutex)) != ENS_OK)
  return status;

 bool a_command_is_running = false;
 if (my_command_is_running) {
  a_command_is_running = true;
  my_stop_is_required = true;
  my_stop_is_acknowledged = false;
 }
```

```
 if (a_command_is_running) {
  while (my_stop_is_acknowledged == false) {
   while ((pthread_cond_wait(&my_cond_stop_is_acknowledged,
&my_mutex) == -1) && errno == EINTR)
    continue; // Restart when interrupted by handler
  }
 }

 my_stop_is_required = false;
 if ((status = pthread_mutex_unlock(&my_mutex)) != ENS_OK)
  return status;

 return ENS_OK;
}

int fifo_is_busy()
{
 return my_command_is_running;
}

static int sleep_until_start_request_or_inactivity()
{
 int a_start_is_required = false;

 // Wait for the start request (my_cond_start_is_required).
 // Besides this, if the audio stream is still busy,
 // check from time to time its end.
 // The end of the stream is confirmed by several checks
 // for filtering underflow.
 //
 int i = 0;
 int err = pthread_mutex_lock(&my_mutex);
 assert(err != -1);
 while ((i <= MAX_INACTIVITY_CHECK) && !a_start_is_required) {
  i++;
```

```
  struct timespec ts;
  struct timeval tv;

  clock_gettime2(&ts);

  add_time_in_ms(&ts, INACTIVITY_TIMEOUT);

  while ((err =
pthread_cond_timedwait(&my_cond_start_is_required, &my_mutex,
&ts)) == -1
         && errno == EINTR)
   continue;

  assert(gettimeofday(&tv, NULL) != -1);

  if (err == 0)
   a_start_is_required = true;
 }
 pthread_mutex_unlock(&my_mutex);
 return a_start_is_required;
}

static espeak_ng_STATUS close_stream()
{
 espeak_ng_STATUS status = pthread_mutex_lock(&my_mutex);
 if (status != ENS_OK)
  return status;

 bool a_stop_is_required = my_stop_is_required;
 if (!a_stop_is_required)
  my_command_is_running = true;

 status = pthread_mutex_unlock(&my_mutex);

 if (!a_stop_is_required) {
  int a_status = pthread_mutex_lock(&my_mutex);
  if (status == ENS_OK)
```

```
   status = a_status;

  my_command_is_running = false;
  a_stop_is_required = my_stop_is_required;

  a_status = pthread_mutex_unlock(&my_mutex);
  if (status == ENS_OK)
   status = a_status;

  if (a_stop_is_required) {
   // cancel the audio early, to be more responsive when using
eSpeak NG
   // for audio.
   cancel_audio();

   // acknowledge the stop request
   if((a_status = pthread_mutex_lock(&my_mutex)) != ENS_OK)
    return a_status;

   my_stop_is_acknowledged = true;
   a_status = pthread_cond_signal(&my_cond_stop_is_acknowledged);
   if(a_status != ENS_OK)
    return a_status;
   a_status = pthread_mutex_unlock(&my_mutex);
   if (status == ENS_OK)
    status = a_status;

  }
 }

 return status;
}

static void *say_thread(void *p)
{
 (void)p; // unused
```

```
 // announce that thread is started
 assert(-1 != pthread_mutex_lock(&my_mutex));
 my_stop_is_acknowledged = true;
 assert(-1 !=
pthread_cond_signal(&my_cond_stop_is_acknowledged));
 assert(-1 != pthread_mutex_unlock(&my_mutex));

 bool look_for_inactivity = false;

 while (!my_terminate_is_required) {
  bool a_start_is_required = false;
  if (look_for_inactivity) {
   a_start_is_required =
sleep_until_start_request_or_inactivity();
   if (!a_start_is_required)
    close_stream();
  }
  look_for_inactivity = true;

  int a_status = pthread_mutex_lock(&my_mutex);
  assert(!a_status);

  if (!a_start_is_required) {
   while (my_start_is_required == false &&
my_terminate_is_required == false) {
    while ((pthread_cond_wait(&my_cond_start_is_required,
&my_mutex) == -1) && errno == EINTR)
     continue; // Restart when interrupted by handler
   }
  }

  my_command_is_running = true;

  assert(-1 !=
pthread_cond_broadcast(&my_cond_command_is_running));
  assert(-1 != pthread_mutex_unlock(&my_mutex));
```

```
  while (my_command_is_running && !my_terminate_is_required) {
   int a_status = pthread_mutex_lock(&my_mutex);
   assert(!a_status);
   t_espeak_command *a_command = (t_espeak_command *)pop();

   if (a_command == NULL) {
    my_command_is_running = false;
    a_status = pthread_mutex_unlock(&my_mutex);
   } else {
    my_start_is_required = false;

    if (my_stop_is_required)
     my_command_is_running = false;
    a_status = pthread_mutex_unlock(&my_mutex);

    if (my_command_is_running)
     process_espeak_command(a_command);
    delete_espeak_command(a_command);
   }
  }

  if (my_stop_is_required || my_terminate_is_required) {
   // no mutex required since the stop command is synchronous
   // and waiting for my_cond_stop_is_acknowledged
   init(1);

   assert(-1 != pthread_mutex_lock(&my_mutex));
   my_start_is_required = false;

   // acknowledge the stop request
   my_stop_is_acknowledged = true;
   int a_status =
pthread_cond_signal(&my_cond_stop_is_acknowledged);
   assert(a_status != -1);
   pthread_mutex_unlock(&my_mutex);

  }
```

```
  // and wait for the next start
 }

 return NULL;
}

int fifo_is_command_enabled()
{
 return 0 == my_stop_is_required;
}

typedef struct t_node {
 t_espeak_command *data;
 struct t_node *next;
} node;

static node *head = NULL;
static node *tail = NULL;

static espeak_ng_STATUS push(t_espeak_command *the_command)
{
 assert((!head && !tail) || (head && tail));

 if (the_command == NULL)
  return EINVAL;

 if (node_counter >= MAX_NODE_COUNTER)
  return ENS_FIFO_BUFFER_FULL;

 node *n = (node *)malloc(sizeof(node));
 if (n == NULL)
  return ENOMEM;

 if (head == NULL) {
  head = n;
  tail = n;
 } else {
```

```
  tail->next = n;
  tail = n;
 }

 tail->next = NULL;
 tail->data = the_command;

 node_counter++;

 the_command->state = CS_PENDING;

 return ENS_OK;
}

static t_espeak_command *pop()
{
 t_espeak_command *the_command = NULL;

 assert((!head && !tail) || (head && tail));

 if (head != NULL) {
  node *n = head;
  the_command = n->data;
  head = n->next;
  free(n);
  node_counter--;
 }

 if (head == NULL)
  tail = NULL;

 return the_command;
}

static void init(int process_parameters)
{
 t_espeak_command *c = NULL;
```

```
 c = pop();
 while (c != NULL) {
  if (process_parameters && (c->type == ET_PARAMETER || c->type
== ET_VOICE_NAME || c->type == ET_VOICE_SPEC))
   process_espeak_command(c);
  delete_espeak_command(c);
  c = pop();
 }
 node_counter = 0;
}

void fifo_terminate()
{
 my_terminate_is_required = true;
 pthread_cond_signal(&my_cond_start_is_required);
 pthread_join(my_thread, NULL);
 my_terminate_is_required = false;

 pthread_mutex_destroy(&my_mutex);
 pthread_cond_destroy(&my_cond_start_is_required);
 pthread_cond_destroy(&my_cond_stop_is_acknowledged);

 init(0); // purge fifo
}

#endif
```

Chapter 66

./src/libespeak-ng/tr_languages.c

```
#include "config.h"

#include <ctype.h>
#include <locale.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

// start of unicode pages for character sets
#define OFFSET_GREEK    0x380
#define OFFSET_CYRILLIC 0x420
#define OFFSET_ARMENIAN 0x530
```

```
#define OFFSET_HEBREW   0x590
#define OFFSET_ARABIC   0x600
#define OFFSET_SYRIAC   0x700
#define OFFSET_THAANA   0x780 // Divehi/Maldives
#define OFFSET_DEVANAGARI  0x900
#define OFFSET_BENGALI  0x980
#define OFFSET_GURMUKHI 0xa00
#define OFFSET_GUJARATI 0xa80
#define OFFSET_ORIYA    0xb00
#define OFFSET_TAMIL    0xb80
#define OFFSET_TELUGU   0xc00
#define OFFSET_KANNADA  0xc80
#define OFFSET_MALAYALAM 0xd00
#define OFFSET_SINHALA  0x0d80
#define OFFSET_THAI     0x0e00
#define OFFSET_LAO      0x0e80
#define OFFSET_TIBET    0x0f00
#define OFFSET_MYANMAR  0x1000
#define OFFSET_GEORGIAN 0x10a0
#define OFFSET_KOREAN   0x1100
#define OFFSET_ETHIOPIC 0x1200

// character ranges must be listed in ascending unicode order
ALPHABET alphabets[] = {
 { "_el",    OFFSET_GREEK,    0x380, 0x3ff,  L('e', 'l'),
AL_DONT_NAME | AL_NOT_LETTERS | AL_WORDS },
 { "_cyr",   OFFSET_CYRILLIC, 0x400, 0x52f,  0, 0 },
 { "_hy",    OFFSET_ARMENIAN, 0x530, 0x58f,  L('h', 'y'),
AL_WORDS },
 { "_he",    OFFSET_HEBREW,   0x590, 0x5ff,  0, 0 },
 { "_ar",    OFFSET_ARABIC,   0x600, 0x6ff,  0, 0 },
 { "_syc",   OFFSET_SYRIAC,   0x700, 0x74f,  0, 0 },
 { "_dv",    OFFSET_THAANA,   0x780, 0x7bf,  0, 0 },
 { "_hi",    OFFSET_DEVANAGARI, 0x900, 0x97f, L('h', 'i'),
AL_WORDS },
 { "_bn",    OFFSET_BENGALI,  0x0980, 0x9ff, L('b', 'n'),
AL_WORDS },
```

```
 { "_gur",   OFFSET_GURMUKHI, 0xa00, 0xa7f,  L('p', 'a'),
AL_WORDS },
 { "_gu",    OFFSET_GUJARATI, 0xa80, 0xaff,  L('g', 'u'),
AL_WORDS },
 { "_or",    OFFSET_ORIYA,    0xb00, 0xb7f,  0, 0 },
 { "_ta",    OFFSET_TAMIL,    0xb80, 0xbff,  L('t', 'a'),
AL_WORDS },
 { "_te",    OFFSET_TELUGU,   0xc00, 0xc7f,  L('t', 'e'), 0 },
 { "_kn",    OFFSET_KANNADA,  0xc80, 0xcff,  L('k', 'n'),
AL_WORDS },
 { "_ml",    OFFSET_MALAYALAM, 0xd00, 0xd7f,  L('m', 'l'),
AL_WORDS },
 { "_si",    OFFSET_SINHALA,  0xd80, 0xdff,  L('s', 'i'),
AL_WORDS },
 { "_th",    OFFSET_THAI,     0xe00, 0xe7f,  0, 0 },
 { "_lo",    OFFSET_LAO,      0xe80, 0xeff,  0, 0 },
 { "_ti",    OFFSET_TIBET,    0xf00, 0xfff,  0, 0 },
 { "_my",    OFFSET_MYANMAR,  0x1000, 0x109f, 0, 0 },
 { "_ka",    OFFSET_GEORGIAN, 0x10a0, 0x10ff, L('k', 'a'),
AL_WORDS },
 { "_ko",    OFFSET_KOREAN,   0x1100, 0x11ff, L('k', 'o'),
AL_WORDS },
 { "_eth",   OFFSET_ETHIOPIC, 0x1200, 0x139f, 0, 0 },
 { "_braille", 0x2800,        0x2800, 0x28ff, 0, AL_NO_SYMBOL },
 { "_ja",    0x3040,          0x3040, 0x30ff, 0, AL_NOT_CODE },
 { "_zh",    0x3100,          0x3100, 0x9fff, 0, AL_NOT_CODE },
 { "_ko",    0xa700,          0xa700, 0xd7ff, L('k', 'o'),
AL_NOT_CODE | AL_WORDS },
 { NULL, 0, 0, 0, 0, 0 }
};

ALPHABET *AlphabetFromChar(int c)
{
 // Find the alphabet from a character.
 ALPHABET *alphabet = alphabets;

 while (alphabet->name != NULL) {
```

```
  if (c <= alphabet->range_max) {
   if (c >= alphabet->range_min)
    return alphabet;
   else
    break;
  }
  alphabet++;
 }
 return NULL;
}

static void Translator_Russian(Translator *tr);

static void SetLetterVowel(Translator *tr, int c)
{
 tr->letter_bits[c] = (tr->letter_bits[c] & 0x40) | 0x81; // keep
value for group 6 (front vowels e,i,y)
}

static void ResetLetterBits(Translator *tr, int groups)
{
 // Clear all the specified groups
 unsigned int ix;
 unsigned int mask;

 mask = ~groups;

 for (ix = 0; ix < sizeof(tr->letter_bits); ix++)
  tr->letter_bits[ix] &= mask;
}

static void SetLetterBits(Translator *tr, int group, const char
*string)
{
 int bits;
 unsigned char c;
```

```
 bits = (1L << group);
 while ((c = *string++) != 0)
  tr->letter_bits[c] |= bits;
}

static void SetLetterBitsRange(Translator *tr, int group, int
first, int last)
{
 int bits;
 int ix;

 bits = (1L << group);
 for (ix = first; ix <= last; ix++)
  tr->letter_bits[ix] |= bits;
}

static void SetLetterBitsUTF8(Translator *tr, int group, const
char *letters, int offset)
{
 // Add the letters to the specified letter group.
 const char *p = letters;
 int code = -1;
 while (code != 0) {
  int bytes = utf8_in(&code, p);
  if (code > 0x20)
   tr->letter_bits[code - offset] |= (1L << group);
  p += bytes;
 }
}

// ignore these characters
static const unsigned short chars_ignore_default[] = {
 // U+00AD SOFT HYPHEN
 //     Used to mark hyphenation points in words for where to
split a
 //     word at the end of a line to provide readable justified
text.
```

```
 0xad,   1,
 // U+200C ZERO WIDTH NON-JOINER
 //     Used to prevent combined ligatures being displayed in
their
 //     combined form.
 0x200c, 1,
 // U+200D ZERO WIDTH JOINER
 //     Used to indicate an alternative connected form made up of
the
 //     characters surrounding the ZWJ in Devanagari, Kannada,
Malayalam
 //     and Emoji.
// 0x200d, 1, // Not ignored.
 // End of the ignored character list.
 0,      0
};

// alternatively, ignore characters but allow zero-width-non-
joiner (lang-fa)
static const unsigned short chars_ignore_zwnj_hyphen[] = {
 // U+00AD SOFT HYPHEN
 //     Used to mark hyphenation points in words for where to
split a
 //     word at the end of a line to provide readable justified
text.
 0xad,   1,
 // U+0640 TATWEEL (KASHIDA)
 //     Used in Arabic scripts to stretch characters for
justifying
 //     the text.
 0x640,  1,
 // U+200C ZERO WIDTH NON-JOINER
 //     Used to prevent combined ligatures being displayed in
their
 //     combined form.
 0x200c, '-',
 // U+200D ZERO WIDTH JOINER
```

```
 //     Used to indicate an alternative connected form made up of
the
 //     characters surrounding the ZWJ in Devanagari, Kannada,
Malayalam
 //     and Emoji.
// 0x200d, 1, // Not ignored.
 // End of the ignored character list.
 0,      0
};

const unsigned char utf8_ordinal[] = { 0xc2, 0xba, 0 }; //
masculine ordinal character, UTF-8
const unsigned char utf8_null[] = { 0 }; // null string, UTF-8

static Translator *NewTranslator(void)
{
 Translator *tr;
 int ix;
 static const unsigned char stress_amps2[] = { 18, 18, 20, 20,
20, 22, 22, 20 };
 static const short stress_lengths2[8] = { 182, 140, 220, 220,
220, 240, 260, 280 };
 static const wchar_t empty_wstring[1] = { 0 };
 static const wchar_t punct_in_word[2] = { '\'', 0 };  // allow
hyphen within words
 static const unsigned char default_tunes[6] = { 0, 1, 2, 3, 0, 0
};

 // Translates character codes in the range transpose_min to
transpose_max to
 // a number in the range 1 to 63.  0 indicates there is no
translation.
 // Used up to 57 (max of 63)
 static const char transpose_map_latin[] = {
   0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
// 0x60
  16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,  0,  0,  0,  0,  0,
```

```
// 0x70
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
// 0x80
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
// 0x90
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
// 0xa0
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
// 0xb0
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
// 0xc0
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
// 0xd0
  27, 28, 29,  0,  0, 30, 31, 32, 33, 34, 35, 36,  0, 37, 38,  0,
// 0xe0
   0,  0,  0, 39,  0,  0, 40,  0, 41,  0, 42,  0, 43,  0,  0,  0,
// 0xf0
   0,  0,  0, 44,  0, 45,  0, 46,  0,  0,  0,  0,  0, 47,  0,  0,
// 0x100
   0, 48,  0,  0,  0,  0,  0,  0,  0, 49,  0,  0,  0,  0,  0,  0,
// 0x110
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
// 0x120
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
// 0x130
   0,  0, 50,  0, 51,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
// 0x140
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 52,  0,  0,  0,  0,
// 0x150
   0, 53,  0, 54,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
// 0x160
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 55,  0, 56,  0, 57,  0,
// 0x170
 };

 if ((tr = (Translator *)malloc(sizeof(Translator))) == NULL)
  return NULL;
```

```
 tr->encoding = ESPEAKNG_ENCODING_ISO_8859_1;
 dictionary_name[0] = 0;
 tr->dictionary_name[0] = 0;
 tr->dict_condition = 0;
 tr->dict_min_size = 0;
 tr->data_dictrules = NULL; // language_1   translation rules
file
 tr->data_dictlist = NULL;  // language_2   dictionary lookup
file

 tr->transpose_min = 0x60;
 tr->transpose_max = 0x17f;
 tr->transpose_map = transpose_map_latin;
 tr->frequent_pairs = NULL;

 // only need lower case
 tr->letter_bits_offset = 0;
 memset(tr->letter_bits, 0, sizeof(tr->letter_bits));
 memset(tr->letter_groups, 0, sizeof(tr->letter_groups));

 // 0-6 sets of characters matched by A B C H F G Y  in
pronunciation rules
 // these may be set differently for different languages
 SetLetterBits(tr, 0, "aeiou"); // A  vowels, except y
 SetLetterBits(tr, 1, "bcdfgjklmnpqstvxz"); // B  hard
consonants, excluding h,r,w
 SetLetterBits(tr, 2, "bcdfghjklmnpqrstvwxz"); // C  all
consonants
 SetLetterBits(tr, 3, "hlmnr"); // H  'soft' consonants
 SetLetterBits(tr, 4, "cfhkpqstx"); // F  voiceless consonants
 SetLetterBits(tr, 5, "bdgjlmnrvwyz"); // G voiced
 SetLetterBits(tr, 6, "eiy"); // Letter group Y, front vowels
 SetLetterBits(tr, 7, "aeiouy"); // vowels, including y

 tr->char_plus_apostrophe = empty_wstring;
 tr->punct_within_word = punct_in_word;
```

```
 tr->chars_ignore = chars_ignore_default;

 for (ix = 0; ix < 8; ix++) {
  tr->stress_amps[ix] = stress_amps2[ix];
  tr->stress_amps_r[ix] = stress_amps2[ix] - 1;
  tr->stress_lengths[ix] = stress_lengths2[ix];
 }
 memset(&(tr->langopts), 0, sizeof(tr->langopts));
 tr->langopts.max_lengthmod = 500;
 tr->langopts.lengthen_tonic = 20;

 tr->langopts.stress_rule = STRESSPOSN_2R;
 tr->langopts.unstressed_wd1 = 1;
 tr->langopts.unstressed_wd2 = 3;
 tr->langopts.param[LOPT_SONORANT_MIN] = 95;
 tr->langopts.param[LOPT_LONG_VOWEL_THRESHOLD] = 190/2;
 tr->langopts.param[LOPT_MAXAMP_EOC] = 19;
 tr->langopts.param[LOPT_UNPRONOUNCABLE] = 's'; // don't count
this character at start of word
 tr->langopts.param[LOPT_BRACKET_PAUSE] = 4; // pause at bracket
 tr->langopts.param2[LOPT_BRACKET_PAUSE] = 2; // pauses when
announcing bracket names
 tr->langopts.max_initial_consonants = 3;
 tr->langopts.replace_chars = NULL;
 tr->langopts.alt_alphabet_lang = L('e', 'n');
 tr->langopts.roman_suffix = utf8_null;

 SetLengthMods(tr, 201);

 tr->langopts.long_stop = 100;

 tr->langopts.max_roman = 49;
 tr->langopts.min_roman = 2;
 tr->langopts.thousands_sep = ',';
 tr->langopts.decimal_sep = '.';
 tr->langopts.break_numbers = BREAK_THOUSANDS;
 tr->langopts.max_digits = 14;
```

```
 // index by 0=. 1=, 2=?, 3=! 4=none, 5=emphasized
 unsigned char
punctuation_to_tone[INTONATION_TYPES][PUNCT_INTONATIONS] = {
  {  0,  1,  2,  3, 0, 4 },
  {  5,  6,  2,  3, 0, 4 },
  {  5,  7,  1,  3, 0, 4 },
  {  8,  9, 10,  3, 0, 0 },
  {  8,  8, 10,  3, 0, 0 },
  { 11, 11, 11, 11, 0, 0 }, // 6 test
  { 12, 12, 12, 12, 0, 0 }
 };

 memcpy(tr->punct_to_tone, punctuation_to_tone,
sizeof(tr->punct_to_tone));

 memcpy(tr->langopts.tunes, default_tunes,
sizeof(tr->langopts.tunes));

 return tr;
}

// common letter pairs, encode these as a single byte
//  2 bytes, using the transposed character codes
static const short pairs_ru[] = {
 0x010c, //      21052  0x23
 0x010e, //      18400
 0x0113, //      14254
 0x0301, //      31083
 0x030f, //      13420
 0x060e, //      21798
 0x0611, //      19458
 0x0903, //      16226
 0x0b01, //      14456
 0x0b0f, //      17836
 0x0c01, //      13324
 0x0c09, //      16877
```

```
 0x0e01, //      15359
 0x0e06, //      13543  0x30
 0x0e09, //      17168
 0x0e0e, //      15973
 0x0e0f, //      22373
 0x0e1c, //      15052
 0x0f03, //      24947
 0x0f11, //      13552
 0x0f12, //      16368
 0x100f, //      19054
 0x1011, //      17067
 0x1101, //      23967
 0x1106, //      18795
 0x1109, //      13797
 0x110f, //      21737
 0x1213, //      25076
 0x1220, //      14310
 0x7fff
};

static const unsigned char ru_vowels[] = { // (also kazakh)
offset by 0x420 --
 0x10, 0x15, 0x31, 0x18, 0x1e, 0x23, 0x2b, 0x2d, 0x2e, 0x2f,
0xb9, 0xc9, 0x91, 0x8f, 0x36, 0
};
static const unsigned char ru_consonants[] = { //

 0x11, 0x12, 0x13, 0x14, 0x16, 0x17, 0x19, 0x1a, 0x1b, 0x1c,
0x1d, 0x1f, 0x20, 0x21, 0x22, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29,
0x2a, 0x2c, 0x73, 0x7b, 0x83, 0x9b, 0
};

static void SetArabicLetters(Translator *tr)
{
 const char *arab_vowel_letters = "    ";
 const char *arab_consonant_vowel_letters =  "   ;"
 const char *arab_consonant_letters =  "
```

```
                              ;"
 const char *arab_thick_letters =  "     ;"
 const char *arab_shadda_letter = "  ";
 const char *arab_hamza_letter = "  ";
 const char *arab_sukun_letter = "  ";

 SetLetterBitsUTF8(tr, LETTERGP_A, arab_vowel_letters,
OFFSET_ARABIC);
 SetLetterBitsUTF8(tr, LETTERGP_B, arab_consonant_vowel_letters,
OFFSET_ARABIC);
 SetLetterBitsUTF8(tr, LETTERGP_C, arab_consonant_letters,
OFFSET_ARABIC);
 SetLetterBitsUTF8(tr, LETTERGP_F, arab_thick_letters,
OFFSET_ARABIC);
 SetLetterBitsUTF8(tr, LETTERGP_G, arab_shadda_letter,
OFFSET_ARABIC);
 SetLetterBitsUTF8(tr, LETTERGP_H, arab_hamza_letter,
OFFSET_ARABIC);
 SetLetterBitsUTF8(tr, LETTERGP_Y, arab_sukun_letter,
OFFSET_ARABIC);
}

static void SetCyrillicLetters(Translator *tr)
{
 // Set letter types for Cyrillic script languages: bg
(Bulgarian), ru (Russian), tt (Tatar), uk (Ukranian).

 // character codes offset by 0x420
 static const char cyrl_soft[] = { 0x2c, 0x19, 0x27, 0x29, 0 };
// letter group B  [k ts; s;] --
 static const char cyrl_hard[] = { 0x2a, 0x16, 0x26, 0x28, 0 };
// letter group H  [S Z ts] --
 static const char cyrl_nothard[] = { 0x11, 0x12, 0x13, 0x14,
0x17, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1f, 0x20, 0x21, 0x22, 0x24,
0x25, 0x27, 0x29, 0x2c, 0 }; //

 static const char cyrl_voiced[] = { 0x11, 0x12, 0x13, 0x14,
```

```
0x16, 0x17, 0 };    // letter group G  (voiced obstruents) --

 static const char cyrl_ivowels[] = { 0x2c, 0x2e, 0x2f, 0x31, 0
};   // letter group Y  (iotated vowels & soft-sign) --
 tr->encoding = ESPEAKNG_ENCODING_KOI8_R;
 tr->transpose_min = 0x430;  // convert cyrillic from unicode
into range 0x01 to 0x22
 tr->transpose_max = 0x451;
 tr->transpose_map = NULL;
 tr->frequent_pairs = pairs_ru;

 tr->letter_bits_offset = OFFSET_CYRILLIC;
 memset(tr->letter_bits, 0, sizeof(tr->letter_bits));
 SetLetterBits(tr, LETTERGP_A, (char *)ru_vowels);
 SetLetterBits(tr, LETTERGP_B, cyrl_soft);
 SetLetterBits(tr, LETTERGP_C, (char *)ru_consonants);
 SetLetterBits(tr, LETTERGP_H, cyrl_hard);
 SetLetterBits(tr, LETTERGP_F, cyrl_nothard);
 SetLetterBits(tr, LETTERGP_G, cyrl_voiced);
 SetLetterBits(tr, LETTERGP_Y, cyrl_ivowels);
 SetLetterBits(tr, LETTERGP_VOWEL2, (char *)ru_vowels);
}

static void SetIndicLetters(Translator *tr)
{
 // Set letter types for Devanagari (Indic) script languages:
Devanagari, Tamill, etc.

 static const char deva_consonants2[] = { 0x02, 0x03, 0x58, 0x59,
0x5a, 0x5b, 0x5c, 0x5d, 0x5e, 0x5f, 0x7b, 0x7c, 0x7e, 0x7f, 0 };
 static const char deva_vowels2[] = { 0x60, 0x61, 0x55, 0x56,
0x57, 0x62, 0x63, 0 };  // non-consecutive vowels and vowel-signs

 memset(tr->letter_bits, 0, sizeof(tr->letter_bits));
 SetLetterBitsRange(tr, LETTERGP_A, 0x04, 0x14); // vowel letters
 SetLetterBitsRange(tr, LETTERGP_A, 0x3e, 0x4d); // + vowel
signs, and virama
```

```
 SetLetterBits(tr, LETTERGP_A, deva_vowels2);     // + extra
vowels and vowel signs

 SetLetterBitsRange(tr, LETTERGP_B, 0x3e, 0x4d); // vowel signs,
and virama
 SetLetterBits(tr, LETTERGP_B, deva_vowels2);     // + extra
vowels and vowel signs

 SetLetterBitsRange(tr, LETTERGP_C, 0x15, 0x39); // the main
consonant range
 SetLetterBits(tr, LETTERGP_C, deva_consonants2); // + additional
consonants

 SetLetterBitsRange(tr, LETTERGP_Y, 0x04, 0x14); // vowel letters
 SetLetterBitsRange(tr, LETTERGP_Y, 0x3e, 0x4c); // + vowel signs
 SetLetterBits(tr, LETTERGP_Y, deva_vowels2);     // + extra
vowels and vowel signs

 tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1;    // disable check
for unpronouncable words
 tr->langopts.suffix_add_e = tr->letter_bits_offset + 0x4d; //
virama
}

static void SetupTranslator(Translator *tr, const short *lengths,
const unsigned char *amps)
{
 if (lengths != NULL)
  memcpy(tr->stress_lengths, lengths,
sizeof(tr->stress_lengths));
 if (amps != NULL)
  memcpy(tr->stress_amps, amps, sizeof(tr->stress_amps));
}

Translator *SelectTranslator(const char *name)
{
 int name2 = 0;
```

```
 Translator *tr;

 static const short stress_lengths_equal[8] = { 230, 230,  230,
230,  0, 0,  230, 230 };
 static const unsigned char stress_amps_equal[8] = { 19, 19, 19,
19, 19, 19, 19, 19 };

 static const short stress_lengths_fr[8] = { 190, 170,  190, 200,
0, 0,  190, 240 };
 static const unsigned char stress_amps_fr[8] = { 18, 16, 18, 18,
18, 18, 18, 18 };

 static const unsigned char stress_amps_sk[8] = { 17, 16, 20, 20,
20, 22, 22, 21 };
 static const short stress_lengths_sk[8] = { 190, 190, 210, 210,
0, 0, 210, 210 };

 static const short stress_lengths_ta[8] = { 200, 200,  210, 210,
0, 0,  230, 230 };
 static const short stress_lengths_ta2[8] = { 230, 230,  240,
240,  0, 0,  260, 260 };
 static const unsigned char stress_amps_ta[8] = { 18, 18, 18, 18,
20, 20, 22, 22 };

 tr = NewTranslator();
 strcpy(tr->dictionary_name, name);

 // convert name string into a word of up to 4 characters, for
the switch()
 while (*name != 0)
  name2 = (name2 << 8) + *name++;

 switch (name2)
 {
 case L('a', 'f'):
 {
  static const short stress_lengths_af[8] = { 170, 140, 220, 220,
```

```
0, 0, 250, 270 };
  SetupTranslator(tr, stress_lengths_af, NULL);

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.vowel_pause = 0x30;
  tr->langopts.param[LOPT_DIERESES] = 1;
  tr->langopts.param[LOPT_PREFIXES] = 1;
  SetLetterVowel(tr, 'y'); // add 'y' to vowels

  tr->langopts.numbers = NUM_SWAP_TENS | NUM_HUNDRED_AND |
NUM_SINGLE_AND | NUM_ROMAN | NUM_1900;
  tr->langopts.accents = 1;
 }
  break;
 case L('a', 'm'): // Amharic, Ethiopia
 {
  SetupTranslator(tr, stress_lengths_fr, stress_amps_fr);
  tr->letter_bits_offset = OFFSET_ETHIOPIC;
  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.stress_flags = S_NO_AUTO_2 | S_FINAL_DIM; // don't
use secondary stress
  tr->langopts.length_mods0 = tr->langopts.length_mods;  // don't
lengthen vowels in the last syllable
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1;           //
disable check for unpronouncable words
  tr->langopts.numbers = NUM_OMIT_1_HUNDRED;
 }
  break;
 case L('a', 'r'): // Arabic
  tr->transpose_min = OFFSET_ARABIC; // for ar_list, use 6-bit
character codes
  tr->transpose_max = 0x65f;
  tr->transpose_map = NULL;
  tr->letter_bits_offset = OFFSET_ARABIC;
  tr->langopts.numbers = NUM_SWAP_TENS | NUM_AND_UNITS |
NUM_HUNDRED_AND | NUM_OMIT_1_HUNDRED | NUM_AND_HUNDRED |
NUM_THOUSAND_AND | NUM_OMIT_1_THOUSAND;
```

```
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1; // disable check
for unpronouncable words
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_6;
  SetArabicLetters(tr);
  break;
 case L('b', 'g'): // Bulgarian
 {
  SetCyrillicLetters(tr);
  SetLetterVowel(tr, 0x2a);
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_5;
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 0x432; // [v]  don't
count this character at start of word
  tr->langopts.param[LOPT_REGRESSIVE_VOICING] = 0x107; // devoice
at end of word, and change voicing to match a following consonant
(except v)
  tr->langopts.param[LOPT_REDUCE] = 2;
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_ALLOW_SPACE |
NUM_OMIT_1_HUNDRED | NUM_HUNDRED_AND | NUM_AND_UNITS |
NUM_SINGLE_AND | NUM_ROMAN | NUM_ROMAN_ORDINAL |
NUM_ROMAN_CAPITALS;
  tr->langopts.thousands_sep = ' '; // don't allow dot as
thousands separator
 }
  break;
 case L('b', 'n'): // Bengali
 case L('a', 's'): // Assamese
 case L3('b', 'p', 'y'): // Manipuri  (temporary placement - it's
not indo-european)
 {
  static const short stress_lengths_bn[8] = { 180, 180,  210,
210,  0, 0,  230, 240 };
  static const unsigned char stress_amps_bn[8] = { 18, 18, 18,
18, 20, 20, 22, 22 };
  static const char bn_consonants2[3] = { 0x70, 0x71, 0 };

  SetupTranslator(tr, stress_lengths_bn, stress_amps_bn);
```

```
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.stress_flags =  S_MID_DIM | S_FINAL_DIM; // use
'diminished' for unstressed final syllable
  tr->letter_bits_offset = OFFSET_BENGALI;
  SetIndicLetters(tr); // call this after setting OFFSET_BENGALI
  SetLetterBitsRange(tr, LETTERGP_B, 0x01, 0x01); // candranindu
  SetLetterBitsRange(tr, LETTERGP_F, 0x3e, 0x4c); // vowel signs,
but not virama
  SetLetterBits(tr, LETTERGP_C, bn_consonants2);

  tr->langopts.numbers = NUM_SWAP_TENS;
  tr->langopts.break_numbers = BREAK_LAKH_BN;

  if (name2 == L3('b', 'p', 'y')) {
   tr->langopts.numbers = 1;
   tr->langopts.numbers2 = NUM2_SWAP_THOUSANDS;
  }

 }
  break;
 case L('b', 'o'): // Tibet
 {
  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->letter_bits_offset = OFFSET_TIBET;
  SetLetterBitsRange(tr, LETTERGP_A, 0x71, 0x7d); // vowel signs
  SetLetterBitsRange(tr, LETTERGP_B, 0x71, 0x81); // vowel signs
and subjoined letters
  SetLetterBitsRange(tr, LETTERGP_B, 0x90, 0xbc);
  SetLetterBitsRange(tr, LETTERGP_C, 0x40, 0x6c); // consonant
letters (not subjoined)
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1;    // disable
check for unpronouncable words
  tr->langopts.numbers = 1;
 }
```

```
  break;
 case L('c', 'y'): // Welsh
 {
  static const short stress_lengths_cy[8] = { 170, 220, 180, 180,
0, 0, 250, 270 };
  static const unsigned char stress_amps_cy[8] = { 17, 15, 18,
18, 0, 0, 22, 20 }; // 'diminished' is used to mark a quieter,
final unstressed syllable

  SetupTranslator(tr, stress_lengths_cy, stress_amps_cy);

  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_14;
  tr->langopts.stress_rule = STRESSPOSN_2R;

  // 'diminished' is an unstressed final syllable
  tr->langopts.stress_flags =  S_FINAL_DIM_ONLY | S_FINAL_NO_2;
  tr->langopts.unstressed_wd1 = 0;
  tr->langopts.unstressed_wd2 = 2;
  tr->langopts.param[LOPT_SONORANT_MIN] = 120; // limit the
shortening of sonorants before short vowels

  tr->langopts.numbers = NUM_OMIT_1_HUNDRED;

  SetLetterVowel(tr, 'w'); // add letter to vowels and remove
from consonants
  SetLetterVowel(tr, 'y');
 }
  break;
 case L('d', 'a'): // Danish
 {
  static const short stress_lengths_da[8] = { 160, 140, 200, 200,
0, 0, 220, 230 };
  SetupTranslator(tr, stress_lengths_da, NULL);

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.param[LOPT_PREFIXES] = 1;
  SetLetterVowel(tr, 'y');
```

```
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_SWAP_TENS |
NUM_HUNDRED_AND | NUM_OMIT_1_HUNDRED | NUM_ORDINAL_DOT | NUM_1900
| NUM_ROMAN | NUM_ROMAN_CAPITALS | NUM_ROMAN_ORDINAL;
 }
  break;
 case L('d', 'e'):
 {
  static const short stress_lengths_de[8] = { 150, 130, 200, 200,
0, 0, 270, 270 };
  static const unsigned char stress_amps_de[] = { 20, 20, 20, 20,
20, 22, 22, 20 };
  SetupTranslator(tr, stress_lengths_de, stress_amps_de);
  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.word_gap = 0x8; // don't use linking phonemes
  tr->langopts.vowel_pause = 0x30;
  tr->langopts.param[LOPT_PREFIXES] = 1;
  tr->langopts.param[LOPT_REGRESSIVE_VOICING] = 0x100; // devoice
at end of word
  tr->langopts.param[LOPT_LONG_VOWEL_THRESHOLD] = 175/2;

  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_SWAP_TENS |
NUM_ALLOW_SPACE | NUM_ORDINAL_DOT | NUM_ROMAN;
  SetLetterVowel(tr, 'y');
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 2; // use de_rules
for unpronouncable rules
 }
  break;
 case L('d', 'v'): // Divehi (Maldives) FIXME: this language code
is actually never used
 {
  SetupTranslator(tr, stress_lengths_ta, stress_amps_ta);
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1; // disable check
for unpronouncable words
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable
  tr->letter_bits_offset = OFFSET_THAANA;
  tr->langopts.stress_rule = STRESSPOSN_1L;
```

```
  tr->langopts.stress_flags =  S_MID_DIM | S_FINAL_DIM; // use
'diminished' for unstressed final syllable
  SetLetterBitsRange(tr, LETTERGP_B, 0x26, 0x30); // vowel signs,
and virama
  tr->langopts.break_numbers = BREAK_LAKH_DV;
  tr->langopts.numbers = 1;
 }
  break;
 case L('e', 'n'):
 {
  static const short stress_lengths_en[8] = { 182, 140, 220, 220,
0, 0, 248, 275 };
  SetupTranslator(tr, stress_lengths_en, NULL);

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.stress_flags = 0x08;
  tr->langopts.numbers = NUM_HUNDRED_AND | NUM_ROMAN | NUM_1900;
  tr->langopts.max_digits = 33;
  tr->langopts.param[LOPT_COMBINE_WORDS] = 2; // allow "mc" to
cmbine with the following word
  tr->langopts.suffix_add_e = 'e';
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 2; // use en_rules
for unpronouncable rules
  SetLetterBits(tr, 6, "aeiouy"); // Group Y: vowels, including y
 }
  break;
 case L('e', 'l'): // Greek
 case L3('g', 'r', 'c'): // Ancient Greek
 {
  static const short stress_lengths_el[8] = { 155, 180,  210,
210,  0, 0,  270, 300 };
  static const unsigned char stress_amps_el[8] = { 15, 12, 20,
20, 20, 22, 22, 21 }; // 'diminished' is used to mark a quieter,
final unstressed syllable

  // character codes offset by 0x380
  static const char el_vowels[] = { 0x10, 0x2c, 0x2d, 0x2e, 0x2f,
```

```
0x30, 0x31, 0x35, 0x37, 0x39, 0x3f, 0x45, 0x49, 0x4a, 0x4b, 0x4c,
0x4d, 0x4e, 0x4f, 0 };
  static const char el_fvowels[] = { 0x2d, 0x2e, 0x2f, 0x35,
0x37, 0x39, 0x45, 0x4d, 0 }; //                    _
  static const char el_voiceless[] = { 0x38, 0x3a, 0x3e, 0x40,
0x42, 0x43, 0x44, 0x46, 0x47, 0 }; //                    _
  static const char el_consonants[] = { 0x32, 0x33, 0x34, 0x36,
0x38, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x40, 0x41, 0x42, 0x43, 0x44,
0x46, 0x47, 0x48, 0 };
  static const wchar_t el_char_apostrophe[] = { 0x3c3, 0 }; //
-

  SetupTranslator(tr, stress_lengths_el, stress_amps_el);

  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_7;
  tr->char_plus_apostrophe = el_char_apostrophe;

  tr->letter_bits_offset = OFFSET_GREEK;
  memset(tr->letter_bits, 0, sizeof(tr->letter_bits));
  SetLetterBits(tr, LETTERGP_A, el_vowels);
  SetLetterBits(tr, LETTERGP_VOWEL2, el_vowels);
  SetLetterBits(tr, LETTERGP_B, el_voiceless);
  SetLetterBits(tr, LETTERGP_C, el_consonants);
  SetLetterBits(tr, LETTERGP_Y, el_fvowels); // front vowels:
   -

  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.stress_flags = S_FINAL_DIM_ONLY; // mark
unstressed final syllables as diminished
  tr->langopts.unstressed_wd1 = 0;
  tr->langopts.unstressed_wd2 = 2;
  tr->langopts.param[LOPT_SONORANT_MIN] = 130; // limit the
shortening of sonorants before short vowels

  tr->langopts.numbers = NUM_SINGLE_STRESS | NUM_DECIMAL_COMMA;
```

```
  tr->langopts.numbers2 = 0x2 | NUM2_MULTIPLE_ORDINAL |
NUM2_ORDINAL_NO_AND; // variant form of numbers before thousands

  if (name2 == L3('g', 'r', 'c')) {
   // ancient greek
   tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1;
  }
 }
  break;
 case L('e', 'o'):
 {
  static const short stress_lengths_eo[8] = { 150, 140,  180,
180,    0,   0,  200, 200 };
  static const unsigned char stress_amps_eo[] = { 16, 14, 20, 20,
20, 22, 22, 21 };
  static const wchar_t eo_char_apostrophe[2] = { 'l', 0 };

  SetupTranslator(tr, stress_lengths_eo, stress_amps_eo);

  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_3;
  tr->char_plus_apostrophe = eo_char_apostrophe;

  tr->langopts.vowel_pause = 2;
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.stress_flags =  S_FINAL_DIM_ONLY | S_FINAL_NO_2;
  tr->langopts.unstressed_wd2 = 2;

  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_OMIT_1_HUNDRED |
NUM_ALLOW_SPACE | NUM_ROMAN;
 }
  break;
 case L('e', 's'): // Spanish
 case L('a', 'n'): // Aragonese
 case L('c', 'a'): // Catalan
 case L('i', 'a'): // Interlingua
 case L3('p', 'a', 'p'): // Papiamento
 {
```

```
  static const short stress_lengths_es[8] = { 160, 145,  155,
150,  0, 0,  200, 245 };
  static const unsigned char stress_amps_es[8] = { 16, 14, 15,
16, 20, 20, 22, 22 }; // 'diminished' is used to mark a quieter,
final unstressed syllable
  static const wchar_t ca_punct_within_word[] = { '\'', 0xb7, 0
}; // ca: allow middle-dot within word

  SetupTranslator(tr, stress_lengths_es, stress_amps_es);

  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable
  tr->langopts.stress_rule = STRESSPOSN_2R;

  // stress last syllable if it doesn't end in vowel or "s" or
"n"
  // 'diminished' is an unstressed final syllable
  tr->langopts.stress_flags = S_FINAL_SPANISH | S_FINAL_DIM_ONLY
| S_FINAL_NO_2;
  tr->langopts.unstressed_wd1 = 0;
  tr->langopts.unstressed_wd2 = 2;
  tr->langopts.param[LOPT_SONORANT_MIN] = 120; // limit the
shortening of sonorants before short vowels

  tr->langopts.numbers = NUM_SINGLE_STRESS | NUM_DECIMAL_COMMA |
NUM_AND_UNITS | NUM_OMIT_1_HUNDRED | NUM_OMIT_1_THOUSAND |
NUM_ROMAN | NUM_ROMAN_AFTER | NUM_DFRACTION_4;
  tr->langopts.numbers2 = NUM2_MULTIPLE_ORDINAL |
NUM2_ORDINAL_NO_AND;

  if (name2 == L('c', 'a')) {
   // stress last syllable unless word ends with a vowel
   tr->punct_within_word = ca_punct_within_word;
   tr->langopts.stress_flags = S_FINAL_SPANISH | S_FINAL_DIM_ONLY
| S_FINAL_NO_2 | S_NO_AUTO_2;
  } else if (name2 == L('i', 'a')) {
   tr->langopts.stress_flags = S_FINAL_SPANISH | S_FINAL_DIM_ONLY
```

```
| S_FINAL_NO_2;
   tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_OMIT_1_HUNDRED
| NUM_OMIT_1_THOUSAND | NUM_ROMAN | NUM_ROMAN_AFTER;
  } else if (name2 == L('a', 'n')) {
   tr->langopts.stress_flags = S_FINAL_SPANISH | S_FINAL_DIM_ONLY
| S_FINAL_NO_2;
   tr->langopts.numbers = NUM_SINGLE_STRESS | NUM_DECIMAL_COMMA |
NUM_AND_UNITS | NUM_OMIT_1_HUNDRED | NUM_OMIT_1_THOUSAND |
NUM_ROMAN | NUM_ROMAN_ORDINAL;
   tr->langopts.numbers2 = NUM2_ORDINAL_NO_AND;
   tr->langopts.roman_suffix = utf8_ordinal;
  } else if (name2 == L3('p', 'a', 'p')) {
   // stress last syllable unless word ends with a vowel
   tr->langopts.stress_rule = STRESSPOSN_1R;
   tr->langopts.stress_flags = S_FINAL_VOWEL_UNSTRESSED |
S_FINAL_DIM_ONLY | S_FINAL_NO_2 | S_NO_AUTO_2;
  } else
   tr->langopts.param[LOPT_UNPRONOUNCABLE] = 2; // use es_rules
for unpronouncable rules
 }
  break;
 case L('e', 'u'): // basque
 {
  static const short stress_lengths_eu[8] = { 200, 200,  200,
200,  0, 0,  210, 230 }; // very weak stress
  static const unsigned char stress_amps_eu[8] = { 16, 16, 18,
18, 18, 18, 18, 18 };
  SetupTranslator(tr, stress_lengths_eu, stress_amps_eu);
  tr->langopts.stress_rule = STRESSPOSN_2L; // ?? second
syllable, but not on a word-final vowel
  tr->langopts.stress_flags = S_FINAL_VOWEL_UNSTRESSED;
  tr->langopts.param[LOPT_SUFFIX] = 1;
  tr->langopts.numbers = NUM_SINGLE_STRESS | NUM_DECIMAL_COMMA |
NUM_HUNDRED_AND | NUM_OMIT_1_HUNDRED | NUM_OMIT_1_THOUSAND |
NUM_VIGESIMAL;
 }
  break;
```

```
 case L('f', 'a'): // Farsi
 {
  // Convert characters in the range 0x620 to 0x6cc to the range
1 to 63.
  // 0 indicates no translation for this character
  static const char transpose_map_fa[] = {
    0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
15, // 0x620
   16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,  0,  0,  0,  0,
0, // 0x630
    0, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, // 0x640
   42, 43,  0,  0, 44,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0, // 0x650
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0, // 0x660
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 45,
0, // 0x670
    0,  0,  0,  0,  0,  0, 46,  0,  0,  0,  0,  0,  0,  0,  0,
0, // 0x680
    0,  0,  0,  0,  0,  0,  0,  0, 47,  0,  0,  0,  0,  0,  0,
0, // 0x690
    0,  0,  0,  0,  0,  0,  0,  0,  0, 48,  0,  0,  0,  0,  0,
49, // 0x6a0
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0, // 0x6b0
   50,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 51
// 0x6c0
  };
  tr->transpose_min = 0x620;
  tr->transpose_max = 0x6cc;
  tr->transpose_map = transpose_map_fa;
  tr->letter_bits_offset = OFFSET_ARABIC;

  tr->langopts.numbers = NUM_AND_UNITS | NUM_HUNDRED_AND;
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1; // disable check
for unpronouncable words
```

```
  tr->chars_ignore = chars_ignore_zwnj_hyphen; // replace ZWNJ by
hyphen
 }
  break;
 case L('e', 't'): // Estonian
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_4;
  // fallthrough:
 case L('f', 'i'): // Finnish
 {
  static const unsigned char stress_amps_fi[8] = { 18, 16, 22,
22, 20, 22, 22, 22 };
  static const short stress_lengths_fi[8] = { 150, 180, 200, 200,
0, 0, 210, 250 };

  SetupTranslator(tr, stress_lengths_fi, stress_amps_fi);

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.stress_flags = S_FINAL_DIM_ONLY | S_FINAL_NO_2 |
S_2_TO_HEAVY; // move secondary stress from light to a following
heavy syllable
  tr->langopts.param[LOPT_IT_DOUBLING] = 1;
  tr->langopts.long_stop = 130;

  tr->langopts.numbers = NUM_DECIMAL_COMMA + NUM_ALLOW_SPACE;
  SetLetterVowel(tr, 'y');
  tr->langopts.spelling_stress = 1;
  tr->langopts.intonation_group = 3; // less intonation, don't
raise pitch at comma
 }
  break;
 case L('f', 'r'): // french
 {
  SetupTranslator(tr, stress_lengths_fr, stress_amps_fr);
  tr->langopts.stress_rule = STRESSPOSN_1R; // stress on final
syllable
  tr->langopts.stress_flags = S_NO_AUTO_2 | S_FINAL_DIM; // don't
```

```
use secondary stress
  tr->langopts.param[LOPT_IT_LENGTHEN] = 1; // remove lengthen
indicator from unstressed syllables
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable
  tr->langopts.accents = 2; // Say "Capital" after the letter.

  tr->langopts.numbers = NUM_SINGLE_STRESS | NUM_DECIMAL_COMMA |
NUM_ALLOW_SPACE | NUM_OMIT_1_HUNDRED | NUM_NOPAUSE | NUM_ROMAN |
NUM_ROMAN_CAPITALS | NUM_ROMAN_AFTER | NUM_VIGESIMAL |
NUM_DFRACTION_4;
  SetLetterVowel(tr, 'y');
 }
  break;
    case L3('h','a', 'k'): // Hakka Chinese
    {
        tr->langopts.stress_flags = S_NO_DIM; // don't
automatically set diminished stress (may be set in the intonation
module)
        tr->langopts.tone_language = 1; // Tone language, use
CalcPitches_Tone() rather than CalcPitches()
        tr->langopts.tone_numbers = 1; // a number after letters
indicates a tone number (eg. pinyin or jyutping)
        tr->langopts.ideographs = 1;
    }
        break;
 case L('g', 'a'): // irish
 case L('g', 'd'): // scots gaelic
 {
  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.stress_flags = S_NO_AUTO_2; // don't use secondary
stress
  tr->langopts.numbers = NUM_OMIT_1_HUNDRED |
NUM_OMIT_1_THOUSAND;
  tr->langopts.accents = 2; // 'capital' after letter name
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 3; // don't count
apostrophe
```

```
  tr->langopts.param[LOPT_IT_LENGTHEN] = 1; // remove [:] phoneme
from non-stressed syllables (Lang=gd)
 }
  break;
 case L('g','n'):   // guarani
  {
   tr->langopts.stress_rule = STRESSPOSN_1R;      // stress on
final syllable
   tr->langopts.length_mods0 = tr->langopts.length_mods;  //
don't lengthen vowels in the last syllable
  }
  break;
 case L('h', 'i'): // Hindi
 case L('n', 'e'): // Nepali
 case L('o', 'r'): // Oriya
 case L('p', 'a'): // Punjabi
 case L('g', 'u'): // Gujarati
 case L('m', 'r'): // Marathi
 {
  static const short stress_lengths_hi[8] = { 190, 190,  210,
210,  0, 0,  230, 250 };
  static const unsigned char stress_amps_hi[8] = { 17, 14, 20,
19, 20, 22, 22, 21 };

  SetupTranslator(tr, stress_lengths_hi, stress_amps_hi);
  tr->encoding = ESPEAKNG_ENCODING_ISCII;
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable

  tr->langopts.stress_rule = 6; // stress on last heaviest
syllable, excluding final syllable
  tr->langopts.stress_flags =  S_MID_DIM | S_FINAL_DIM; // use
'diminished' for unstressed final syllable
  tr->langopts.numbers = NUM_SWAP_TENS;
  tr->langopts.break_numbers = BREAK_LAKH_HI;
  tr->letter_bits_offset = OFFSET_DEVANAGARI;
```

```
  if (name2 == L('p', 'a'))
   tr->letter_bits_offset = OFFSET_GURMUKHI;
  else if (name2 == L('g', 'u')) {
   SetupTranslator(tr, stress_lengths_equal, stress_amps_equal);
   tr->letter_bits_offset = OFFSET_GUJARATI;
   tr->langopts.stress_rule = STRESSPOSN_2R;
  } else if (name2 == L('n', 'e')) {
   SetupTranslator(tr, stress_lengths_equal, stress_amps_equal);
   tr->langopts.break_numbers = BREAK_LAKH;
   tr->langopts.max_digits = 22;
   tr->langopts.numbers2 |= NUM2_ENGLISH_NUMERALS;
  } else if (name2 == L('o', 'r'))
   tr->letter_bits_offset = OFFSET_ORIYA;
  SetIndicLetters(tr);
 }
  break;
 case L('h', 'r'): // Croatian
 case L('b', 's'): // Bosnian
 case L('s', 'r'): // Serbian
 {
  static const unsigned char stress_amps_hr[8] = { 17, 17, 20,
20, 20, 22, 22, 21 };
  static const short stress_lengths_hr[8] = { 180, 160, 200, 200,
0, 0, 220, 230 };
  static const short stress_lengths_sr[8] = { 160, 150, 200, 200,
0, 0, 250, 260 };

  strcpy(tr->dictionary_name, "hbs");

  if (name2 == L('s', 'r'))
   SetupTranslator(tr, stress_lengths_sr, stress_amps_hr);
  else
   SetupTranslator(tr, stress_lengths_hr, stress_amps_hr);
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_2;

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.stress_flags = S_FINAL_NO_2;
```

```
  tr->langopts.param[LOPT_REGRESSIVE_VOICING] = 0x3;
  tr->langopts.max_initial_consonants = 5;
  tr->langopts.spelling_stress = 1;
  tr->langopts.accents = 1;

  tr->langopts.numbers = NUM_SINGLE_STRESS | NUM_HUNDRED_AND |
NUM_OMIT_1_HUNDRED | NUM_DECIMAL_COMMA | NUM_THOUS_SPACE |
NUM_DFRACTION_2 | NUM_ROMAN_CAPITALS;
  tr->langopts.numbers2 = 0xa + NUM2_THOUSANDS_VAR5; // variant
numbers before thousands,milliards
  tr->langopts.our_alphabet = OFFSET_CYRILLIC; // don't say
"cyrillic" before letter names

  SetLetterVowel(tr, 'y');
  SetLetterVowel(tr, 'r');
 }
  break;
 case L('h', 't'): // Haitian Creole
  tr->langopts.stress_rule = STRESSPOSN_1R; // stress on final
syllable
  tr->langopts.stress_flags = S_NO_AUTO_2 | S_FINAL_DIM; // don't
use secondary stress
  tr->langopts.numbers = NUM_SINGLE_STRESS | NUM_OMIT_1_HUNDRED |
NUM_NOPAUSE | NUM_ROMAN | NUM_VIGESIMAL | NUM_DFRACTION_4;
  break;
 case L('h', 'u'): // Hungarian
 {
  static const unsigned char stress_amps_hu[8] = { 17, 17, 19,
19, 20, 22, 22, 21 };
  static const short stress_lengths_hu[8] = { 185, 195, 195, 190,
0, 0, 210, 220 };

  SetupTranslator(tr, stress_lengths_hu, stress_amps_hu);
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_2;

  tr->langopts.vowel_pause = 0x20;
  tr->langopts.stress_rule = STRESSPOSN_1L;
```

```
  tr->langopts.stress_flags = S_FINAL_DIM_ONLY | S_FINAL_NO_2 |
S_NO_AUTO_2 | 0x8000 | S_HYPEN_UNSTRESS;
  tr->langopts.unstressed_wd1 = 2;
  tr->langopts.param[LOPT_IT_DOUBLING] = 1;
  tr->langopts.param[LOPT_ANNOUNCE_PUNCT] = 2; // don't break
clause before announcing . ? !

  tr->langopts.numbers = NUM_DFRACTION_5 | NUM_ALLOW_SPACE |
NUM_ROMAN | NUM_ROMAN_ORDINAL | NUM_ROMAN_CAPITALS |
NUM_ORDINAL_DOT | NUM_OMIT_1_HUNDRED | NUM_OMIT_1_THOUSAND;
  tr->langopts.thousands_sep = ' '; // don't allow dot as
thousands separator
  tr->langopts.decimal_sep = ',';
  tr->langopts.max_roman = 899;
  tr->langopts.min_roman = 1;
  SetLetterVowel(tr, 'y');
  tr->langopts.spelling_stress = 1;
  SetLengthMods(tr, 3); // all equal
 }
  break;
 case L('h', 'y'): // Armenian
 {
  static const short stress_lengths_hy[8] = { 250, 200,  250,
250,  0, 0,  250, 250 };
  static const char hy_vowels[] = { 0x31, 0x35, 0x37, 0x38, 0x3b,
0x48, 0x55, 0 };
  static const char hy_consonants[] = {
   0x32, 0x33, 0x34, 0x36, 0x39, 0x3a, 0x3c, 0x3d, 0x3e, 0x3f,
0x40, 0x41, 0x42, 0x43, 0x44,
   0x46, 0x47, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f, 0x50,
0x51, 0x52, 0x53, 0x54, 0x56, 0
  };
  static const char hy_consonants2[] = { 0x45, 0 };

  SetupTranslator(tr, stress_lengths_hy, NULL);
  tr->langopts.stress_rule = STRESSPOSN_1R; // default stress on
final syllable
```

```
  tr->letter_bits_offset = OFFSET_ARMENIAN;
  memset(tr->letter_bits, 0, sizeof(tr->letter_bits));
  SetLetterBits(tr, LETTERGP_A, hy_vowels);
  SetLetterBits(tr, LETTERGP_VOWEL2, hy_vowels);
  SetLetterBits(tr, LETTERGP_B, hy_consonants); // not including
'j'
  SetLetterBits(tr, LETTERGP_C, hy_consonants);
  SetLetterBits(tr, LETTERGP_C, hy_consonants2); // add 'j'
  tr->langopts.max_initial_consonants = 6;
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_ALLOW_SPACE |
NUM_OMIT_1_HUNDRED;
 }
  break;
 case L('i', 'd'): // Indonesian
 case L('m', 's'): // Malay
 {
  static const short stress_lengths_id[8] = { 160, 200,  180,
180,  0, 0,  220, 240 };
  static const unsigned char stress_amps_id[8] = { 16, 18, 18,
18, 20, 22, 22, 21 };

  SetupTranslator(tr, stress_lengths_id, stress_amps_id);
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_ALLOW_SPACE |
NUM_ROMAN;
  tr->langopts.stress_flags =  S_FINAL_DIM_ONLY | S_FINAL_NO_2;
  tr->langopts.accents = 2; // "capital" after letter name
 }
  break;
 case L('i', 's'): // Icelandic
 {
  static const short stress_lengths_is[8] = { 180, 160, 200, 200,
0, 0, 240, 250 };
  static const wchar_t is_lettergroup_B[] = { 'c', 'f', 'h', 'k',
'p', 't', 'x', 0xfe, 0 }; // voiceless conants, including 'þ'  ??
's'
```

```
  SetupTranslator(tr, stress_lengths_is, NULL);
  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.stress_flags = S_FINAL_NO_2;
  tr->langopts.param[LOPT_IT_LENGTHEN] = 0x11; // remove lengthen
indicator from unstressed vowels
  tr->langopts.param[LOPT_REDUCE] = 2;

  ResetLetterBits(tr, 0x18);
  SetLetterBits(tr, 4, "kpst"); // Letter group F
  SetLetterBits(tr, 3, "jvr"); // Letter group H
  tr->letter_groups[1] = is_lettergroup_B;
  SetLetterVowel(tr, 'y');
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_SINGLE_AND |
NUM_HUNDRED_AND | NUM_AND_UNITS | NUM_1900;
  tr->langopts.numbers2 = 0x2;
 }
  break;
 case L('i', 't'): // Italian
 {
  static const short stress_lengths_it[8] = { 160, 140, 150, 165,
0, 0, 218, 305 };
  static const unsigned char stress_amps_it[8] = { 17, 15, 18,
16, 20, 22, 22, 22 };
  SetupTranslator(tr, stress_lengths_it, stress_amps_it);
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.stress_flags = S_NO_AUTO_2 | S_FINAL_DIM_ONLY |
S_PRIORITY_STRESS;
  tr->langopts.vowel_pause = 1;
  tr->langopts.unstressed_wd1 = 0;
  tr->langopts.unstressed_wd2 = 2;
  tr->langopts.param[LOPT_IT_LENGTHEN] = 2; // remove lengthen
indicator from unstressed or non-penultimate syllables
  tr->langopts.param[LOPT_IT_DOUBLING] = 1; // double the first
consonant if the previous word ends in a stressed vowel (changed
```

```
to =1, 23.01.2014 - only use if prev.word has $double)
  tr->langopts.param[LOPT_SONORANT_MIN] = 130; // limit the
shortening of sonorants before short vowels
  tr->langopts.param[LOPT_REDUCE] = 1; // reduce vowels even if
phonemes are specified in it_list
  tr->langopts.param[LOPT_ALT] = 2; // call
ApplySpecialAttributes2() if a word has $alt or $alt2
  tr->langopts.numbers = NUM_SINGLE_VOWEL | NUM_OMIT_1_HUNDRED
|NUM_DECIMAL_COMMA | NUM_DFRACTION_1 | NUM_ROMAN |
NUM_ROMAN_CAPITALS | NUM_ROMAN_ORDINAL;
  tr->langopts.numbers2 = NUM2_NO_TEEN_ORDINALS;
  tr->langopts.roman_suffix = utf8_ordinal;
  tr->langopts.accents = 2; // Say "Capital" after the letter.
  SetLetterVowel(tr, 'y');
 }
  break;
 case L3('j', 'b', 'o'): // Lojban
 {
  static const short stress_lengths_jbo[8] = { 145, 145, 170,
160, 0, 0, 330, 350 };
  static const wchar_t jbo_punct_within_word[] = { '.', ',',
'\'', 0x2c8, 0 }; // allow period and comma within a word, also
stress marker (from LOPT_CAPS_IN_WORD)

  SetupTranslator(tr, stress_lengths_jbo, NULL);
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.vowel_pause = 0x20c; // pause before a word which
starts with a vowel, or after a word which ends in a consonant
  tr->punct_within_word = jbo_punct_within_word;
  tr->langopts.param[LOPT_CAPS_IN_WORD] = 2; // capitals indicate
stressed syllables
  SetLetterVowel(tr, 'y');
  tr->langopts.max_lengthmod = 368;
 }
  break;
 case L('k', 'a'): // Georgian
 {
```

```
  // character codes offset by 0x1080
  static const char ka_vowels[] = { 0x30, 0x34, 0x38, 0x3d, 0x43,
0x55, 0x57, 0 };
  static const char ka_consonants[] =
  { 0x31, 0x32, 0x33, 0x35, 0x36, 0x37, 0x39, 0x3a, 0x3b, 0x3c,
0x3e, 0x3f, 0x40, 0x41, 0x42, 0x44,
    0x45, 0x46, 0x47, 0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e,
0x4f, 0x50, 0x51, 0x52, 0x53, 0x54, 0x56, 0 };
  SetupTranslator(tr, stress_lengths_ta, stress_amps_ta);
  memset(tr->letter_bits, 0, sizeof(tr->letter_bits));
  SetLetterBits(tr, LETTERGP_A, ka_vowels);
  SetLetterBits(tr, LETTERGP_C, ka_consonants);
  SetLetterBits(tr, LETTERGP_VOWEL2, ka_vowels);

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.stress_flags = S_FINAL_NO_2;
  tr->letter_bits_offset = OFFSET_GEORGIAN;
  tr->langopts.max_initial_consonants = 7;
  tr->langopts.numbers = NUM_VIGESIMAL | NUM_AND_UNITS |
NUM_OMIT_1_HUNDRED |NUM_OMIT_1_THOUSAND | NUM_DFRACTION_5 |
NUM_ROMAN;

  tr->langopts.alt_alphabet = OFFSET_CYRILLIC;
  tr->langopts.alt_alphabet_lang = L('r', 'u');
 }
  break;
 case L('k', 'k'): // Kazakh
 {
  static const unsigned char stress_amps_tr[8] = { 18, 16, 20,
21, 20, 21, 21, 20 };
  static const short stress_lengths_tr[8] = { 190, 180, 230, 230,
0, 0, 250, 250 };

  tr->letter_bits_offset = OFFSET_CYRILLIC;
  memset(tr->letter_bits, 0, sizeof(tr->letter_bits));
  SetLetterBits(tr, LETTERGP_A, (char *)ru_vowels);
  SetLetterBits(tr, LETTERGP_C, (char *)ru_consonants);
```

```
  SetLetterBits(tr, LETTERGP_VOWEL2, (char *)ru_vowels);

  SetupTranslator(tr, stress_lengths_tr, stress_amps_tr);

  tr->langopts.stress_rule = 7; // stress on the last syllable,
before any explicitly unstressed syllable
  tr->langopts.stress_flags = S_NO_AUTO_2 + S_NO_EOC_LENGTHEN; //
no automatic secondary stress, don't lengthen at end-of-clause
  tr->langopts.lengthen_tonic = 0;
  tr->langopts.param[LOPT_SUFFIX] = 1;

  tr->langopts.numbers =  NUM_OMIT_1_HUNDRED | NUM_DFRACTION_6;
  tr->langopts.max_initial_consonants = 2;
  SetLengthMods(tr, 3); // all equal
 }
  break;
 case L('k', 'l'): // Greenlandic
 {
  SetupTranslator(tr, stress_lengths_equal, stress_amps_equal);
  tr->langopts.stress_rule = 12;
  tr->langopts.stress_flags = S_NO_AUTO_2;
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_SWAP_TENS |
NUM_HUNDRED_AND | NUM_OMIT_1_HUNDRED | NUM_ORDINAL_DOT | NUM_1900
| NUM_ROMAN | NUM_ROMAN_CAPITALS | NUM_ROMAN_ORDINAL;
 }
  break;
 case L('k', 'o'): // Korean, TEST
 {
  static const char ko_ivowels[] = { 0x63, 0x64, 0x67, 0x68,
0x6d, 0x72, 0x74, 0x75, 0 }; // y and i vowels
  static const unsigned char ko_voiced[] = { 0x02, 0x05, 0x06,
0xab, 0xaf, 0xb7, 0xbc, 0 }; // voiced consonants, l,m,n,N

  tr->letter_bits_offset = OFFSET_KOREAN;
  tr->langopts.our_alphabet = 0xa700;
  memset(tr->letter_bits, 0, sizeof(tr->letter_bits));
  SetLetterBitsRange(tr, LETTERGP_A, 0x61, 0x75);
```

```
  SetLetterBits(tr, LETTERGP_Y, ko_ivowels);
  SetLetterBits(tr, LETTERGP_G, (const char *)ko_voiced);

  tr->langopts.stress_rule = 8; // ?? 1st syllable if it is
heavy, else 2nd syllable
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1; // disable check
for unpronouncable words
  tr->langopts.numbers = NUM_OMIT_1_HUNDRED;
  tr->langopts.numbers2 = NUM2_MYRIADS;
  tr->langopts.break_numbers = BREAK_MYRIADS;
  tr->langopts.max_digits = 20;
 }
  break;
 case L('k', 'u'): // Kurdish
 {
  static const unsigned char stress_amps_ku[8] = { 18, 18, 20,
20, 20, 22, 22, 21 };
  static const short stress_lengths_ku[8] = { 180, 180, 190, 180,
0, 0, 230, 240 };

  SetupTranslator(tr, stress_lengths_ku, stress_amps_ku);
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_9;

  tr->langopts.stress_rule = 7; // stress on the last syllable,
before any explicitly unstressed syllable

  tr->langopts.numbers = NUM_HUNDRED_AND | NUM_AND_UNITS |
NUM_OMIT_1_HUNDRED | NUM_AND_HUNDRED;
  tr->langopts.max_initial_consonants = 2;
 }
  break;
 case L('k', 'y'): // Kyrgyx
  tr->langopts.numbers = 1;
  break;
 case L('l', 'a'): // Latin
 {
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_4; // includes
```

```
a,e,i,o,u-macron
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.stress_flags = S_NO_AUTO_2;
  tr->langopts.unstressed_wd1 = 0;
  tr->langopts.unstressed_wd2 = 2;
  tr->langopts.param[LOPT_DIERESES] = 1;
  tr->langopts.numbers = NUM_ROMAN;
  tr->langopts.max_roman = 5000;
 }
  break;
 case L('l', 't'): // Lithuanian
 {
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_4;
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.stress_flags = S_NO_AUTO_2;
  tr->langopts.unstressed_wd1 = 0;
  tr->langopts.unstressed_wd2 = 2;
  tr->langopts.param[LOPT_DIERESES] = 1;
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_OMIT_1_HUNDRED |
NUM_DFRACTION_4 | NUM_ORDINAL_DOT;
  tr->langopts.numbers2 = NUM2_THOUSANDS_VAR4;
  tr->langopts.max_roman = 5000;
 }
  break;
 case L('l', 'v'): // latvian
 {
  static const unsigned char stress_amps_lv[8] = { 14, 10, 10, 8,
0, 0, 20, 15 };
  static const short stress_lengths_lv[8] = { 180, 180, 180, 160,
0, 0, 230, 180 };

  SetupTranslator(tr, stress_lengths_lv, stress_amps_lv);

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.spelling_stress = 1;
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_4;
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_OMIT_1_HUNDRED |
```

```
NUM_DFRACTION_4 | NUM_ORDINAL_DOT;
  tr->langopts.stress_flags = S_NO_AUTO_2 | S_FINAL_DIM |
S_FINAL_DIM_ONLY | S_EO_CLAUSE1;
 }
  break;
 case L('m', 'k'): // Macedonian
 {
  static wchar_t vowels_cyrillic[] = {
   // also include ' ' [R]
   0x440, 0x430, 0x435, 0x438, 0x439, 0x43e, 0x443, 0x44b, 0x44d,
   0x44e, 0x44f, 0x450, 0x451, 0x456, 0x457, 0x45d, 0x45e, 0
  };
  static const unsigned char stress_amps_mk[8] = { 17, 17, 20,
20, 20, 22, 22, 21 };
  static const short stress_lengths_mk[8] = { 180, 160, 200, 200,
0, 0, 220, 230 };

  SetupTranslator(tr, stress_lengths_mk, stress_amps_mk);
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_5;
  tr->letter_groups[0] = tr->letter_groups[7] = vowels_cyrillic;
  tr->letter_bits_offset = OFFSET_CYRILLIC;

  tr->langopts.stress_rule = STRESSPOSN_3R; // antipenultimate
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_AND_UNITS |
NUM_OMIT_1_HUNDRED | NUM_OMIT_1_THOUSAND | NUM_DFRACTION_2;
  tr->langopts.numbers2 = 0x8a; // variant numbers before
thousands,milliards
 }
  break;
 case L('m', 't'): // Maltese
 {
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_3;
  tr->langopts.param[LOPT_REGRESSIVE_VOICING] = 0x100; // devoice
at end of word
  tr->langopts.stress_rule = STRESSPOSN_2R; // penultimate
  tr->langopts.numbers = 1;
 }
```

```
  break;
 case L('n', 'l'): // Dutch
 {
  static const short stress_lengths_nl[8] = { 160, 135, 210, 210,
0, 0, 260, 280 };

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.vowel_pause = 0x30; // ??
  tr->langopts.param[LOPT_DIERESES] = 1;
  tr->langopts.param[LOPT_PREFIXES] = 1;
  tr->langopts.param[LOPT_REGRESSIVE_VOICING] = 0x100; // devoice
at end of word
  SetLetterVowel(tr, 'y');

  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_SWAP_TENS |
NUM_OMIT_1_HUNDRED | NUM_OMIT_1_THOUSAND | NUM_ALLOW_SPACE |
NUM_1900 | NUM_ORDINAL_DOT;
  tr->langopts.ordinal_indicator = "e";
  tr->langopts.stress_flags = S_FIRST_PRIMARY;
  memcpy(tr->stress_lengths, stress_lengths_nl,
sizeof(tr->stress_lengths));
 }
  break;
 case L('n', 'o'): // Norwegian
 {
  static const short stress_lengths_no[8] = { 160, 140, 200, 200,
0, 0, 220, 230 };

  SetupTranslator(tr, stress_lengths_no, NULL);
  tr->langopts.stress_rule = STRESSPOSN_1L;
  SetLetterVowel(tr, 'y');
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_HUNDRED_AND |
NUM_ALLOW_SPACE | NUM_1900 | NUM_ORDINAL_DOT;
 }
  break;
 case L('o', 'm'): // Oromo
 {
```

```
  static const unsigned char stress_amps_om[] = { 18, 15, 20, 20,
20, 22, 22, 22 };
  static const short stress_lengths_om[8] = { 200, 200, 200, 200,
0, 0, 200, 200 };

  SetupTranslator(tr, stress_lengths_om, stress_amps_om);
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.stress_flags = S_FINAL_DIM_ONLY | S_FINAL_NO_2 |
0x80000;
  tr->langopts.numbers = NUM_OMIT_1_HUNDRED | NUM_HUNDRED_AND;
  tr->langopts.numbers2 = 0x200; // say "thousands" before its
number
 }
  break;
 case L('p', 'l'): // Polish
 {
  static const short stress_lengths_pl[8] = { 160, 190,  175,
175,  0, 0,  200, 210 };
  static const unsigned char stress_amps_pl[8] = { 17, 13, 19,
19, 20, 22, 22, 21 }; // 'diminished' is used to mark a quieter,
final unstressed syllable

  SetupTranslator(tr, stress_lengths_pl, stress_amps_pl);

  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_2;
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.stress_flags = S_FINAL_DIM_ONLY; // mark
unstressed final syllables as diminished
  tr->langopts.param[LOPT_REGRESSIVE_VOICING] = 0x9;
  tr->langopts.max_initial_consonants = 7; // for example:
wchrzczony :)
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_ALLOW_SPACE |
NUM_DFRACTION_2;
  tr->langopts.numbers2 = NUM2_THOUSANDS_VAR3;
  tr->langopts.param[LOPT_COMBINE_WORDS] = 4 + 0x100; // combine
'nie' (marked with $alt2) with some 1-syllable (and 2-syllable)
words (marked with $alt)
```

```
  SetLetterVowel(tr, 'y');
 }
  break;
 case L('p', 't'): // Portuguese
 {
  static const short stress_lengths_pt[8] = { 170, 115,  210,
240,  0, 0,  260, 280 };
  static const unsigned char stress_amps_pt[8] = { 16, 11, 19,
21, 20, 22, 22, 21 }; // 'diminished' is used to mark a quieter,
final unstressed syllable

  SetupTranslator(tr, stress_lengths_pt, stress_amps_pt);
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable

  tr->langopts.stress_rule = STRESSPOSN_1R; // stress on final
syllable
  tr->langopts.stress_flags =  S_FINAL_DIM_ONLY | S_FINAL_NO_2 |
S_INITIAL_2 | S_PRIORITY_STRESS;
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_DFRACTION_2 |
NUM_HUNDRED_AND | NUM_AND_UNITS | NUM_ROMAN_CAPITALS;
  tr->langopts.numbers2 = NUM2_MULTIPLE_ORDINAL |
NUM2_NO_TEEN_ORDINALS | NUM2_ORDINAL_NO_AND;
  tr->langopts.max_roman = 5000;
  SetLetterVowel(tr, 'y');
  ResetLetterBits(tr, 0x2);
  SetLetterBits(tr, 1, "bcdfgjkmnpqstvxz"); // B  hard
consonants, excluding h,l,r,w,y
  tr->langopts.param[LOPT_ALT] = 2; // call
ApplySpecialAttributes2() if a word has $alt or $alt2
  tr->langopts.accents = 2; // 'capital' after letter name
 }
  break;
 case L('r', 'o'): // Romanian
 {
  static const short stress_lengths_ro[8] = { 170, 170,  180,
180,  0, 0,  240, 260 };
```

```
  static const unsigned char stress_amps_ro[8] = { 15, 13, 18,
18, 20, 22, 22, 21 };

  SetupTranslator(tr, stress_lengths_ro, stress_amps_ro);

  tr->langopts.stress_rule = STRESSPOSN_1R;
  tr->langopts.stress_flags = S_FINAL_VOWEL_UNSTRESSED |
S_FINAL_DIM_ONLY;

  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_2;
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_ALLOW_SPACE |
NUM_DFRACTION_3 | NUM_AND_UNITS | NUM_ROMAN;
  tr->langopts.numbers2 = 0x1e; // variant numbers before all
thousandplex
 }
  break;
 case L('r', 'u'): // Russian
  Translator_Russian(tr);
  break;
 case L('r', 'w'): // Kiryarwanda
 {
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.stress_flags = S_FINAL_DIM_ONLY | S_FINAL_NO_2;
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1; // disable check
for unpronouncable words.  Need to allow "bw'" prefix
  tr->langopts.numbers = NUM_HUNDRED_AND | NUM_AND_UNITS |
NUM_DFRACTION_2 | NUM_AND_HUNDRED;
  tr->langopts.numbers2 = 0x200; // say "thousands" before its
number
 }
  break;
 case L('s', 'k'): // Slovak
 case L('c', 's'): // Czech
 {
  static const char *sk_voiced = "bdgjlmnrvwzaeiouy";
```

```
  SetupTranslator(tr, stress_lengths_sk, stress_amps_sk);
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_2;

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.stress_flags = S_FINAL_DIM_ONLY | S_FINAL_NO_2;
  tr->langopts.param[LOPT_REGRESSIVE_VOICING] = 0x3;
  tr->langopts.max_initial_consonants = 5;
  tr->langopts.spelling_stress = 1;
  tr->langopts.param[LOPT_COMBINE_WORDS] = 4; // combine some
prepositions with the following word

  tr->langopts.numbers = NUM_OMIT_1_HUNDRED | NUM_DFRACTION_2 |
NUM_ROMAN;
  tr->langopts.numbers2 = NUM2_THOUSANDS_VAR2;
  tr->langopts.thousands_sep = 0; // no thousands separator
  tr->langopts.decimal_sep = ',';

  if (name2 == L('c', 's'))
   tr->langopts.numbers2 = 0x108; // variant numbers before
milliards

  SetLetterVowel(tr, 'y');
  SetLetterVowel(tr, 'r');
  ResetLetterBits(tr, 0x20);
  SetLetterBits(tr, 5, sk_voiced);
 }
  break;
 case L('s', 'i'): // Sinhala
 {
  SetupTranslator(tr, stress_lengths_ta, stress_amps_ta);
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.stress_flags =  S_FINAL_DIM_ONLY | S_FINAL_NO_2;
  tr->langopts.spelling_stress = 1;
```

```
  tr->letter_bits_offset = OFFSET_SINHALA;
  memset(tr->letter_bits, 0, sizeof(tr->letter_bits));
  SetLetterBitsRange(tr, LETTERGP_A, 0x05, 0x16); // vowel
letters
  SetLetterBitsRange(tr, LETTERGP_A, 0x4a, 0x73); // + vowel
signs, and virama

  SetLetterBitsRange(tr, LETTERGP_B, 0x4a, 0x73); // vowel signs,
and virama

  SetLetterBitsRange(tr, LETTERGP_C, 0x1a, 0x46); // the main
consonant range

  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1; // disable check
for unpronouncable words
  tr->langopts.suffix_add_e = tr->letter_bits_offset + 0x4a; //
virama
  tr->langopts.numbers =  NUM_OMIT_1_THOUSAND |
NUM_SINGLE_STRESS_L | NUM_DFRACTION_7;
  tr->langopts.numbers2 =  NUM2_PERCENT_BEFORE;
  tr->langopts.break_numbers = BREAK_LAKH_HI;
 }
  break;
 case L('s', 'l'): // Slovenian
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_2;
  tr->langopts.stress_rule = STRESSPOSN_2R; // Temporary
  tr->langopts.stress_flags = S_NO_AUTO_2;
  tr->langopts.param[LOPT_REGRESSIVE_VOICING] = 0x103;
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 0x76; // [v]  don't
count this character at start of word
  tr->langopts.param[LOPT_ALT] = 2; // call
ApplySpecialAttributes2() if a word has $alt or $alt2
  tr->langopts.param[LOPT_IT_LENGTHEN] = 1; // remove lengthen
indicator from unstressed syllables
  tr->letter_bits[(int)'r'] |= 0x80; // add 'r' to letter group
7, vowels for Unpronouncable test
```

```
  tr->langopts.numbers =  NUM_DECIMAL_COMMA | NUM_ALLOW_SPACE |
NUM_SWAP_TENS | NUM_OMIT_1_HUNDRED | NUM_DFRACTION_2 |
NUM_ORDINAL_DOT | NUM_ROMAN;
  tr->langopts.numbers2 = 0x100; // plural forms of millions etc
  tr->langopts.thousands_sep = ' '; // don't allow dot as
thousands separator
  break;
 case L('s', 'q'): // Albanian
 {
  static const short stress_lengths_sq[8] = { 150, 150,  180,
180,  0, 0,  300, 300 };
  static const unsigned char stress_amps_sq[8] = { 16, 12, 16,
16, 20, 20, 21, 19 };

  SetupTranslator(tr, stress_lengths_sq, stress_amps_sq);

  tr->langopts.stress_rule = STRESSPOSN_1R;
  tr->langopts.stress_flags =  S_FINAL_DIM_ONLY | S_FINAL_NO_2 |
S_FINAL_VOWEL_UNSTRESSED;
  SetLetterVowel(tr, 'y');
  tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_HUNDRED_AND |
NUM_AND_UNITS | NUM_DFRACTION_4;
  tr->langopts.accents = 2; // "capital" after letter name
 }
  break;
 case L('s', 'v'): // Swedish
 {
  static const unsigned char stress_amps_sv[] = { 16, 16, 20, 20,
20, 22, 22, 21 };
  static const short stress_lengths_sv[8] = { 160, 135, 220, 220,
0, 0, 250, 280 };
  SetupTranslator(tr, stress_lengths_sv, stress_amps_sv);

  tr->langopts.stress_rule = STRESSPOSN_1L;
  SetLetterVowel(tr, 'y');
  tr->langopts.numbers = NUM_SINGLE_STRESS | NUM_DECIMAL_COMMA |
NUM_ALLOW_SPACE | NUM_1900;
```

```
  tr->langopts.accents = 1;
 }
  break;
 case L('s', 'w'): // Swahili
 case L('t', 'n'): // Setswana
 {
  static const short stress_lengths_sw[8] = { 160, 170,  200,
200,    0,   0,  320, 340 };
  static const unsigned char stress_amps_sw[] = { 16, 12, 19, 19,
20, 22, 22, 21 };

  SetupTranslator(tr, stress_lengths_sw, stress_amps_sw);
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable

  tr->langopts.vowel_pause = 1;
  tr->langopts.stress_rule = STRESSPOSN_2R;
  tr->langopts.stress_flags =  S_FINAL_DIM_ONLY | S_FINAL_NO_2;
  tr->langopts.max_initial_consonants = 4; // for example: mwngi

  tr->langopts.numbers = NUM_AND_UNITS | NUM_HUNDRED_AND |
NUM_SINGLE_AND | NUM_OMIT_1_HUNDRED;
 }
  break;
 case L('t', 'a'): // Tamil
 case L('k', 'n'): // Kannada
 case L('m', 'l'): // Malayalam
 case L('t', 'e'): // Telugu
 {
  SetupTranslator(tr, stress_lengths_ta2, stress_amps_ta);
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.stress_flags =  S_FINAL_DIM_ONLY | S_FINAL_NO_2;
// use 'diminished' for unstressed final syllable
  tr->langopts.spelling_stress = 1;
```

```
  tr->langopts.break_numbers = BREAK_LAKH_DV;

  if (name2 == L('t', 'a')) {
   SetupTranslator(tr, stress_lengths_ta, NULL);
   tr->letter_bits_offset = OFFSET_TAMIL;
   tr->langopts.numbers =  NUM_OMIT_1_THOUSAND;
   tr->langopts.numbers2 = NUM2_ORDINAL_AND_THOUSANDS;
   tr->langopts.param[LOPT_WORD_MERGE] = 1; // don't break vowels
betwen words
  } else if (name2 == L('m', 'l')) {
   static const short stress_lengths_ml[8] = { 180, 160,  240,
240,  0, 0,  260, 260 };
   SetupTranslator(tr, stress_lengths_ml, stress_amps_equal);
   tr->letter_bits_offset = OFFSET_MALAYALAM;
   tr->langopts.numbers = NUM_OMIT_1_THOUSAND |
NUM_OMIT_1_HUNDRED;
   tr->langopts.numbers2 = NUM2_OMIT_1_HUNDRED_ONLY;
   tr->langopts.stress_rule = 13; // 1st syllable, unless 1st
vowel is short and 2nd is long
  } else if (name2 == L('k', 'n')) {
   tr->letter_bits_offset = OFFSET_KANNADA;
   tr->langopts.numbers = 0x1;
  } else if (name2 == L('t', 'e')) {
   tr->letter_bits_offset = OFFSET_TELUGU;
   tr->langopts.numbers = 0x1;
   tr->langopts.numbers2 = NUM2_ORDINAL_DROP_VOWEL;
  }
  SetIndicLetters(tr); // call this after setting OFFSET_
  SetLetterBitsRange(tr, LETTERGP_B, 0x4e, 0x4e); // chillu-
virama (unofficial)
 }
  break;
 case L('t', 'r'): // Turkish
 case L('a', 'z'): // Azerbaijan
 {
  static const unsigned char stress_amps_tr[8] = { 18, 16, 20,
21, 20, 21, 21, 20 };
```

```
  static const short stress_lengths_tr[8] = { 190, 180, 200, 230,
0, 0, 240, 250 };

  SetupTranslator(tr, stress_lengths_tr, stress_amps_tr);
  tr->encoding = ESPEAKNG_ENCODING_ISO_8859_9;

  tr->langopts.stress_rule = 7; // stress on the last syllable,
before any explicitly unstressed syllable
  tr->langopts.stress_flags = S_NO_AUTO_2; // no automatic
secondary stress
  tr->langopts.dotless_i = 1;
  tr->langopts.param[LOPT_SUFFIX] = 1;

  if (name2 == L('a', 'z'))
   tr->langopts.numbers = NUM_SINGLE_STRESS | NUM_DECIMAL_COMMA
| NUM_ALLOW_SPACE | NUM_OMIT_1_HUNDRED | NUM_OMIT_1_THOUSAND |
NUM_DFRACTION_2;
  else
   tr->langopts.numbers = NUM_SINGLE_STRESS | NUM_DECIMAL_COMMA |
NUM_OMIT_1_HUNDRED | NUM_OMIT_1_THOUSAND | NUM_DFRACTION_2;
  tr->langopts.max_initial_consonants = 2;
 }
  break;
 case L('t', 't'): // Tatar
 {
  SetCyrillicLetters(tr);
  SetupTranslator(tr, stress_lengths_fr, stress_amps_fr);
  tr->langopts.stress_rule = STRESSPOSN_1R; // stress on final
syllable
  tr->langopts.stress_flags = S_NO_AUTO_2; // no automatic
secondary stress
  tr->langopts.numbers = NUM_SINGLE_STRESS | NUM_DECIMAL_COMMA |
NUM_OMIT_1_HUNDRED | NUM_OMIT_1_THOUSAND | NUM_DFRACTION_4;
 }
  break;
 case L('u', 'k'): // Ukrainian
 {
```

```
  SetCyrillicLetters(tr);
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 0x432; // [v]  don't
count this character at start of word
 }
  break;
 case L('u', 'r'): // Urdu
 case L('s', 'd'): // Sindhi
 {
  tr->letter_bits_offset = OFFSET_ARABIC;
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1; // disable check
for unpronouncable words
  tr->langopts.numbers = NUM_SWAP_TENS;
  tr->langopts.break_numbers = BREAK_LAKH_UR;
 }
  break;
 case L('v', 'i'): // Vietnamese
 {
  static const short stress_lengths_vi[8] = { 150, 150,  180,
180,  210, 230,  230, 240 };
  static const unsigned char stress_amps_vi[] = { 16, 16, 16, 16,
22, 22, 22, 22 };
  static wchar_t vowels_vi[] = {
    0x61,   0xe0,   0xe1, 0x1ea3,   0xe3, 0x1ea1, // a
   0x103, 0x1eb1, 0x1eaf, 0x1eb3, 0x1eb5, 0x1eb7, // ă
    0xe2, 0x1ea7, 0x1ea5, 0x1ea9, 0x1eab, 0x1ead, // â
    0x65,   0xe8,   0xe9, 0x1ebb, 0x1ebd, 0x1eb9, // e
    0xea, 0x1ec1, 0x1ebf, 0x1ec3, 0x1ec5, 0x1ec7, // i
    0x69,   0xec,   0xed, 0x1ec9,  0x129, 0x1ecb, // i
    0x6f,   0xf2,   0xf3, 0x1ecf,   0xf5, 0x1ecd, // o
    0xf4, 0x1ed3, 0x1ed1, 0x1ed5, 0x1ed7, 0x1ed9, // ô
   0x1a1, 0x1edd, 0x1edb, 0x1edf, 0x1ee1, 0x1ee3, // ơ
    0x75,   0xf9,   0xfa, 0x1ee7,  0x169, 0x1ee5, // u
   0x1b0, 0x1eeb, 0x1ee9, 0x1eed, 0x1eef, 0x1ef1, // ư
    0x79, 0x1ef3,   0xfd, 0x1ef7, 0x1ef9, 0x1ef5, // y
   0
  };
```

```
  SetupTranslator(tr, stress_lengths_vi, stress_amps_vi);
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable

  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.word_gap = 0x21; // length of a final vowel is
less dependent on the next consonant, don't merge consonant with
next word
  tr->letter_groups[0] = tr->letter_groups[7] = vowels_vi;
  tr->langopts.tone_language = 1; // Tone language, use
CalcPitches_Tone() rather than CalcPitches()
  tr->langopts.unstressed_wd1 = 2;
  tr->langopts.numbers = NUM_DECIMAL_COMMA |
NUM_HUNDRED_AND_DIGIT | NUM_DFRACTION_4 | NUM_ZERO_HUNDRED;

 }
  break;
 case L('w', 'o'):
  tr->langopts.stress_rule = STRESSPOSN_1L;
  tr->langopts.numbers = NUM_AND_UNITS | NUM_HUNDRED_AND |
NUM_OMIT_1_HUNDRED | NUM_OMIT_1_THOUSAND | NUM_SINGLE_STRESS;
  break;
 case L3('s', 'h', 'n'):
  tr->langopts.tone_language = 1; // Tone language, use
CalcPitches_Tone() rather than CalcPitches()
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable
  tr->langopts.numbers = 1;
  tr->langopts.break_numbers = BREAK_INDIVIDUAL;
  break;
 case L3('c', 'm', 'n'): // no break, just go to 'zh' case
 case L3('z', 'h', 'y'): // just go to 'zh' case
 case L('z','h'):
 {
  static const short stress_lengths_zh[8] = { 230, 150, 230, 230,
230, 0, 240, 250 }; // 1=tone5. end-of-sentence, 6=tone 1&4,
7=tone 2&3
```

```
  static const unsigned char stress_amps_zh[] = { 22, 16, 22, 22,
22, 22, 22, 22 };

  SetupTranslator(tr, stress_lengths_zh, stress_amps_zh);

  tr->langopts.stress_rule = STRESSPOSN_1R; // stress on final
syllable of a "word"
  tr->langopts.stress_flags = S_NO_DIM; // don't automatically
set diminished stress (may be set in the intonation module)
  tr->langopts.vowel_pause = 0;
  tr->langopts.tone_language = 1; // Tone language, use
CalcPitches_Tone() rather than CalcPitches()
  tr->langopts.length_mods0 = tr->langopts.length_mods; // don't
lengthen vowels in the last syllable
  tr->langopts.tone_numbers = 1; // a number after letters
indicates a tone number (eg. pinyin or jyutping)
  tr->langopts.ideographs = 1;
  tr->langopts.our_alphabet = 0x3100;
  tr->langopts.word_gap = 0x21; // length of a final vowel is
less dependent on the next consonant, don't merge consonant with
next word
  if (name2 == L3('z', 'h', 'y')) {
   tr->langopts.textmode = true;
   tr->langopts.listx = 1; // compile zh_listx after zh_list
   tr->langopts.numbers = 1;
   tr->langopts.numbers2 = NUM2_ZERO_TENS;
   tr->langopts.break_numbers = BREAK_INDIVIDUAL;
  }
  break;
 }
 default:
  tr->langopts.param[LOPT_UNPRONOUNCABLE] = 1; // disable check
for unpronouncable words
  break;
 }

 tr->translator_name = name2;
```

```
 ProcessLanguageOptions(&tr->langopts);
 return tr;
}

void ProcessLanguageOptions(LANGUAGE_OPTIONS *langopts)
{
 if (langopts->numbers & NUM_DECIMAL_COMMA) {
  // use . and ; for thousands and decimal separators
  langopts->thousands_sep = '.';
  langopts->decimal_sep = ',';
 }
 if (langopts->numbers & NUM_THOUS_SPACE)
  langopts->thousands_sep = 0; // don't allow thousands
separator, except space
}

static void Translator_Russian(Translator *tr)
{
 static const unsigned char stress_amps_ru[] = { 16, 16, 18, 18,
20, 24, 24, 22 };
 static const short stress_lengths_ru[8] = { 150, 140, 220, 220,
0, 0, 260, 280 };
 static const char ru_ivowels[] = { 0x15, 0x18, 0x34, 0x37, 0 };
// add "     " to Y lettergroup (iotated vowels & soft-sign)

 SetupTranslator(tr, stress_lengths_ru, stress_amps_ru);
 SetCyrillicLetters(tr);
 SetLetterBits(tr, LETTERGP_Y, ru_ivowels);

 tr->langopts.param[LOPT_UNPRONOUNCABLE] = 0x432; // [v]  don't
count this character at start of word
 tr->langopts.param[LOPT_REGRESSIVE_VOICING] = 1;
 tr->langopts.param[LOPT_REDUCE] = 2;
 tr->langopts.stress_rule = 5;
 tr->langopts.stress_flags = S_NO_AUTO_2;
```

```
 tr->langopts.numbers = NUM_DECIMAL_COMMA | NUM_OMIT_1_HUNDRED;
 tr->langopts.numbers2 = 0x2 + NUM2_THOUSANDS_VAR1; // variant
numbers before thousands
}
```

Chapter 67

./src/libespeak-ng/numbers.c

```
#include "config.h"

#include <ctype.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wctype.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "dictionary.h"
#include "numbers.h"
#include "readclause.h"
#include "synthdata.h"

#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
```

```
#include "translate.h"

#define M_LIGATURE  0x8000
#define M_NAME      0
#define M_SMALLCAP  1
#define M_TURNED    2
#define M_REVERSED  3
#define M_CURL      4

#define M_ACUTE     5
#define M_BREVE     6
#define M_CARON     7
#define M_CEDILLA   8
#define M_CIRCUMFLEX 9
#define M_DIAERESIS 10
#define M_DOUBLE_ACUTE 11
#define M_DOT_ABOVE 12
#define M_GRAVE     13
#define M_MACRON    14
#define M_OGONEK    15
#define M_RING      16
#define M_STROKE    17
#define M_TILDE     18

#define M_BAR       19
#define M_RETROFLEX 20
#define M_HOOK      21

#define M_MIDDLE_DOT  M_DOT_ABOVE // duplicate of M_DOT_ABOVE
#define M_IMPLOSIVE   M_HOOK

static int n_digit_lookup;
static char *digit_lookup;
static int speak_missing_thousands;
static int number_control;

typedef struct {
```

```
 const char *name;
 int accent_flags;    // bit 0, say before the letter name
} ACCENTS;

// these are tokens to look up in the *_list file.
static ACCENTS accents_tab[] = {
 { "_lig", 1 },
 { "_smc", 0 },  // smallcap
 { "_tur", 0 },  // turned
 { "_rev", 0 },  // reversed
 { "_crl", 0 },  // curl

 { "_acu", 0 },  // acute
 { "_brv", 0 },  // breve
 { "_hac", 0 },  // caron/hacek
 { "_ced", 0 },  // cedilla
 { "_cir", 0 },  // circumflex
 { "_dia", 0 },  // diaeresis
 { "_ac2", 0 },  // double acute
 { "_dot", 0 },  // dot
 { "_grv", 0 },  // grave
 { "_mcn", 0 },  // macron
 { "_ogo", 0 },  // ogonek
 { "_rng", 0 },  // ring
 { "_stk", 0 },  // stroke
 { "_tld", 0 },  // tilde

 { "_bar", 0 },  // bar
 { "_rfx", 0 },  // retroflex
 { "_hok", 0 },  // hook
};

#define CAPITAL  0
#define LETTER(ch, mod1, mod2) (ch-59)+(mod1 << 6)+(mod2 << 11)
#define LIGATURE(ch1, ch2, mod1) (ch1-59)+((ch2-59) << 6)+(mod1
<< 12)+M_LIGATURE
```

```
#define L_ALPHA   60 // U+3B1
#define L_SCHWA   61 // U+259
#define L_OPEN_E  62 // U+25B
#define L_GAMMA   63 // U+3B3
#define L_IOTA    64 // U+3B9

#define L_PHI     67 // U+3C6
#define L_ESH     68 // U+283
#define L_UPSILON 69 // U+3C5
#define L_EZH     70 // U+292
#define L_GLOTTAL 71 // U+294
#define L_RTAP    72 // U+27E
#define L_RLONG   73 // U+27C

static const short non_ascii_tab[] = {
 0,
 0x3b1, 0x259, 0x25b, 0x3b3, 0x3b9, 0x153, 0x3c9,
 0x3c6, 0x283, 0x3c5, 0x292, 0x294, 0x27e, 0x27c
};

// characters U+00e0 to U+017f
static const unsigned short letter_accents_0e0[] = {
 LETTER('a', M_GRAVE, 0), // U+00e0
 LETTER('a', M_ACUTE, 0),
 LETTER('a', M_CIRCUMFLEX, 0),
 LETTER('a', M_TILDE, 0),
 LETTER('a', M_DIAERESIS, 0),
 LETTER('a', M_RING, 0),
 LIGATURE('a', 'e', 0),
 LETTER('c', M_CEDILLA, 0),
 LETTER('e', M_GRAVE, 0),
 LETTER('e', M_ACUTE, 0),
 LETTER('e', M_CIRCUMFLEX, 0),
 LETTER('e', M_DIAERESIS, 0),
 LETTER('i', M_GRAVE, 0),
 LETTER('i', M_ACUTE, 0),
 LETTER('i', M_CIRCUMFLEX, 0),
```

```
LETTER('i', M_DIAERESIS, 0),
LETTER('d', M_NAME, 0), // eth U+00f0
LETTER('n', M_TILDE, 0),
LETTER('o', M_GRAVE, 0),
LETTER('o', M_ACUTE, 0),
LETTER('o', M_CIRCUMFLEX, 0),
LETTER('o', M_TILDE, 0),
LETTER('o', M_DIAERESIS, 0),
0, // division sign
LETTER('o', M_STROKE, 0),
LETTER('u', M_GRAVE, 0),
LETTER('u', M_ACUTE, 0),
LETTER('u', M_CIRCUMFLEX, 0),
LETTER('u', M_DIAERESIS, 0),
LETTER('y', M_ACUTE, 0),
LETTER('t', M_NAME, 0), // thorn
LETTER('y', M_DIAERESIS, 0),
CAPITAL, // U+0100
LETTER('a', M_MACRON, 0),
CAPITAL,
LETTER('a', M_BREVE, 0),
CAPITAL,
LETTER('a', M_OGONEK, 0),
CAPITAL,
LETTER('c', M_ACUTE, 0),
CAPITAL,
LETTER('c', M_CIRCUMFLEX, 0),
CAPITAL,
LETTER('c', M_DOT_ABOVE, 0),
CAPITAL,
LETTER('c', M_CARON, 0),
CAPITAL,
LETTER('d', M_CARON, 0),
CAPITAL, // U+0110
LETTER('d', M_STROKE, 0),
CAPITAL,
LETTER('e', M_MACRON, 0),
```

```
CAPITAL,
LETTER('e', M_BREVE, 0),
CAPITAL,
LETTER('e', M_DOT_ABOVE, 0),
CAPITAL,
LETTER('e', M_OGONEK, 0),
CAPITAL,
LETTER('e', M_CARON, 0),
CAPITAL,
LETTER('g', M_CIRCUMFLEX, 0),
CAPITAL,
LETTER('g', M_BREVE, 0),
CAPITAL, // U+0120
LETTER('g', M_DOT_ABOVE, 0),
CAPITAL,
LETTER('g', M_CEDILLA, 0),
CAPITAL,
LETTER('h', M_CIRCUMFLEX, 0),
CAPITAL,
LETTER('h', M_STROKE, 0),
CAPITAL,
LETTER('i', M_TILDE, 0),
CAPITAL,
LETTER('i', M_MACRON, 0),
CAPITAL,
LETTER('i', M_BREVE, 0),
CAPITAL,
LETTER('i', M_OGONEK, 0),
CAPITAL, // U+0130
LETTER('i', M_NAME, 0), // dotless i
CAPITAL,
LIGATURE('i', 'j', 0),
CAPITAL,
LETTER('j', M_CIRCUMFLEX, 0),
CAPITAL,
LETTER('k', M_CEDILLA, 0),
LETTER('k', M_NAME, 0), // kra
```

```
CAPITAL,
LETTER('l', M_ACUTE, 0),
CAPITAL,
LETTER('l', M_CEDILLA, 0),
CAPITAL,
LETTER('l', M_CARON, 0),
CAPITAL,
LETTER('l', M_MIDDLE_DOT, 0), // U+0140
CAPITAL,
LETTER('l', M_STROKE, 0),
CAPITAL,
LETTER('n', M_ACUTE, 0),
CAPITAL,
LETTER('n', M_CEDILLA, 0),
CAPITAL,
LETTER('n', M_CARON, 0),
LETTER('n', M_NAME, 0), // apostrophe n
CAPITAL,
LETTER('n', M_NAME, 0), // eng
CAPITAL,
LETTER('o', M_MACRON, 0),
CAPITAL,
LETTER('o', M_BREVE, 0),
CAPITAL, // U+0150
LETTER('o', M_DOUBLE_ACUTE, 0),
CAPITAL,
LIGATURE('o', 'e', 0),
CAPITAL,
LETTER('r', M_ACUTE, 0),
CAPITAL,
LETTER('r', M_CEDILLA, 0),
CAPITAL,
LETTER('r', M_CARON, 0),
CAPITAL,
LETTER('s', M_ACUTE, 0),
CAPITAL,
LETTER('s', M_CIRCUMFLEX, 0),
```

```
 CAPITAL,
 LETTER('s', M_CEDILLA, 0),
 CAPITAL, // U+0160
 LETTER('s', M_CARON, 0),
 CAPITAL,
 LETTER('t', M_CEDILLA, 0),
 CAPITAL,
 LETTER('t', M_CARON, 0),
 CAPITAL,
 LETTER('t', M_STROKE, 0),
 CAPITAL,
 LETTER('u', M_TILDE, 0),
 CAPITAL,
 LETTER('u', M_MACRON, 0),
 CAPITAL,
 LETTER('u', M_BREVE, 0),
 CAPITAL,
 LETTER('u', M_RING, 0),
 CAPITAL, // U+0170
 LETTER('u', M_DOUBLE_ACUTE, 0),
 CAPITAL,
 LETTER('u', M_OGONEK, 0),
 CAPITAL,
 LETTER('w', M_CIRCUMFLEX, 0),
 CAPITAL,
 LETTER('y', M_CIRCUMFLEX, 0),
 CAPITAL, // Y-DIAERESIS
 CAPITAL,
 LETTER('z', M_ACUTE, 0),
 CAPITAL,
 LETTER('z', M_DOT_ABOVE, 0),
 CAPITAL,
 LETTER('z', M_CARON, 0),
 LETTER('s', M_NAME, 0), // long-s U+17f
};

// characters U+0250 to U+029F
```

```
static const unsigned short letter_accents_250[] = {
 LETTER('a', M_TURNED, 0), // U+250
 LETTER(L_ALPHA, 0, 0),
 LETTER(L_ALPHA, M_TURNED, 0),
 LETTER('b', M_IMPLOSIVE, 0),
 0, // open-o
 LETTER('c', M_CURL, 0),
 LETTER('d', M_RETROFLEX, 0),
 LETTER('d', M_IMPLOSIVE, 0),
 LETTER('e', M_REVERSED, 0), // U+258
 0, // schwa
 LETTER(L_SCHWA, M_HOOK, 0),
 0, // open-e
 LETTER(L_OPEN_E, M_REVERSED, 0),
 LETTER(L_OPEN_E, M_HOOK, M_REVERSED),
 0,
 LETTER('j', M_BAR, 0),
 LETTER('g', M_IMPLOSIVE, 0), // U+260
 LETTER('g', 0, 0),
 LETTER('g', M_SMALLCAP, 0),
 LETTER(L_GAMMA, 0, 0),
 0, // ramshorn
 LETTER('h', M_TURNED, 0),
 LETTER('h', M_HOOK, 0),
 0,
 LETTER('i', M_BAR, 0), // U+268
 LETTER(L_IOTA, 0, 0),
 LETTER('i', M_SMALLCAP, 0),
 LETTER('l', M_TILDE, 0),
 LETTER('l', M_BAR, 0),
 LETTER('l', M_RETROFLEX, 0),
 LIGATURE('l', 'z', 0),
 LETTER('m', M_TURNED, 0),
 0,
 LETTER('m', M_HOOK, 0),
 0,
 LETTER('n', M_RETROFLEX, 0),
```

```
LETTER('n', M_SMALLCAP, 0),
LETTER('o', M_BAR, 0),
LIGATURE('o', 'e', M_SMALLCAP),
0,
LETTER(L_PHI, 0, 0), // U+278
LETTER('r', M_TURNED, 0),
LETTER(L_RLONG, M_TURNED, 0),
LETTER('r', M_RETROFLEX, M_TURNED),
0,
LETTER('r', M_RETROFLEX, 0),
0, // r-tap
LETTER(L_RTAP, M_REVERSED, 0),
LETTER('r', M_SMALLCAP, 0), // U+280
LETTER('r', M_TURNED, M_SMALLCAP),
LETTER('s', M_RETROFLEX, 0),
0, // esh
LETTER('j', M_HOOK, 0),
LETTER(L_ESH, M_REVERSED, 0),
LETTER(L_ESH, M_CURL, 0),
LETTER('t', M_TURNED, 0),
LETTER('t', M_RETROFLEX, 0), // U+288
LETTER('u', M_BAR, 0),
LETTER(L_UPSILON, 0, 0),
LETTER('v', M_HOOK, 0),
LETTER('v', M_TURNED, 0),
LETTER('w', M_TURNED, 0),
LETTER('y', M_TURNED, 0),
LETTER('y', M_SMALLCAP, 0),
LETTER('z', M_RETROFLEX, 0), // U+290
LETTER('z', M_CURL, 0),
0, // ezh
LETTER(L_EZH, M_CURL, 0),
0, // glottal stop
LETTER(L_GLOTTAL, M_REVERSED, 0),
LETTER(L_GLOTTAL, M_TURNED, 0),
0,
0, // bilabial click U+298
```

```
 LETTER('b', M_SMALLCAP, 0),
 0,
 LETTER('g', M_IMPLOSIVE, M_SMALLCAP),
 LETTER('h', M_SMALLCAP, 0),
 LETTER('j', M_CURL, 0),
 LETTER('k', M_TURNED, 0),
 LETTER('l', M_SMALLCAP, 0),
 LETTER('q', M_HOOK, 0), // U+2a0
 LETTER(L_GLOTTAL, M_STROKE, 0),
 LETTER(L_GLOTTAL, M_STROKE, M_REVERSED),
 LIGATURE('d', 'z', 0),
 0, // dezh
 LIGATURE('d', 'z', M_CURL),
 LIGATURE('t', 's', 0),
 0, // tesh
 LIGATURE('t', 's', M_CURL),
};

static int LookupLetter2(Translator *tr, unsigned int letter,
char *ph_buf)
{
 int len;
 char single_letter[10];

 single_letter[0] = 0;
 single_letter[1] = '_';
 len = utf8_out(letter, &single_letter[2]);
 single_letter[len+2] = ' ';
 single_letter[len+3] = 0;

 if (Lookup(tr, &single_letter[1], ph_buf) == 0) {
  single_letter[1] = ' ';
  if (Lookup(tr, &single_letter[2], ph_buf) == 0)
   TranslateRules(tr, &single_letter[2], ph_buf, 20, NULL, 0,
NULL);
 }
 return ph_buf[0];
```

```
}

void LookupAccentedLetter(Translator *tr, unsigned int letter,
char *ph_buf)
{
 // lookup the character in the accents table
 int accent_data = 0;
 int accent1 = 0;
 int accent2 = 0;
 int flags1, flags2;
 int basic_letter;
 int letter2 = 0;
 char ph_letter1[30];
 char ph_letter2[30];
 char ph_accent1[30];
 char ph_accent2[30];

 ph_accent2[0] = 0;

 if ((letter >= 0xe0) && (letter < 0x17f))
  accent_data = letter_accents_0e0[letter - 0xe0];
 else if ((letter >= 0x250) && (letter <= 0x2a8))
  accent_data = letter_accents_250[letter - 0x250];

 if (accent_data != 0) {
  basic_letter = (accent_data & 0x3f) + 59;
  if (basic_letter < 'a')
   basic_letter = non_ascii_tab[basic_letter-59];

  if (accent_data & M_LIGATURE) {
   letter2 = (accent_data >> 6) & 0x3f;
   letter2 += 59;
   accent2 = (accent_data >> 12) & 0x7;
  } else {
   accent1 = (accent_data >> 6) & 0x1f;
   accent2 = (accent_data >> 11) & 0xf;
  }
```

```
  if ((accent1 == 0) && !(accent_data & M_LIGATURE)) {
   // just a letter name, not an accented character or ligature
   return;
  }

  if ((flags1 = Lookup(tr, accents_tab[accent1].name,
ph_accent1)) != 0) {
   if (LookupLetter2(tr, basic_letter, ph_letter1) != 0) {
    if (accent2 != 0) {
     flags2 = Lookup(tr, accents_tab[accent2].name, ph_accent2);
     if (flags2 & FLAG_ACCENT_BEFORE) {
      strcpy(ph_buf, ph_accent2);
      ph_buf += strlen(ph_buf);
      ph_accent2[0] = 0;
     }
    }
    if (letter2 != 0) {
     // ligature
     LookupLetter2(tr, letter2, ph_letter2);
     sprintf(ph_buf, "%s%c%s%c%s%s", ph_accent1,
phonPAUSE_VSHORT, ph_letter1, phonSTRESS_P, ph_letter2,
ph_accent2);
    } else {
     if (accent1 == 0)
      strcpy(ph_buf, ph_letter1);
     else if ((tr->langopts.accents & 1) || (flags1 &
FLAG_ACCENT_BEFORE) || (accents_tab[accent1].accent_flags & 1))
      sprintf(ph_buf, "%s%c%c%s", ph_accent1, phonPAUSE_VSHORT,
phonSTRESS_P, ph_letter1);
     else
      sprintf(ph_buf, "%c%s%c%s%c", phonSTRESS_2, ph_letter1,
phonPAUSE_VSHORT, ph_accent1, phonPAUSE_VSHORT);
    }
   }
  }
 }
```

```
}

void LookupLetter(Translator *tr, unsigned int letter, int
next_byte, char *ph_buf1, int control)
{
 // control, bit 0:  not the first letter of a word

 int len;
 static char single_letter[10] = { 0, 0 };
 unsigned int dict_flags[2];
 char ph_buf3[40];

 ph_buf1[0] = 0;
 len = utf8_out(letter, &single_letter[2]);
 single_letter[len+2] = ' ';

 if (next_byte == -1) {
  // speaking normal text, not individual characters
  if (Lookup(tr, &single_letter[2], ph_buf1) != 0)
   return;

  single_letter[1] = '_';
  if (Lookup(tr, &single_letter[1], ph_buf3) != 0)
   return; // the character is specified as _* so ignore it when
speaking normal text

  // check whether this character is specified for English
  if (tr->translator_name == L('e', 'n'))
   return; // we are already using English

  SetTranslator2("en");
  if (Lookup(translator2, &single_letter[2], ph_buf3) != 0) {
   // yes, switch to English and re-translate the word
   sprintf(ph_buf1, "%c", phonSWITCH);
  }
  SelectPhonemeTable(voice->phoneme_tab_ix); // revert to
original phoneme table
```

```
  return;
 }

 if ((letter <= 32) || iswspace(letter)) {
  // lookup space as _&32 etc.
  sprintf(&single_letter[1], "_#%d ", letter);
  Lookup(tr, &single_letter[1], ph_buf1);
  return;
 }

 if (next_byte != ' ')
  next_byte = RULE_SPELLING;
 single_letter[3+len] = next_byte; // follow by space-space if
the end of the word, or space-31

 single_letter[1] = '_';

 // if the $accent flag is set for this letter, use the accents
table (below)
 dict_flags[1] = 0;

 if (Lookup(tr, &single_letter[1], ph_buf3) == 0) {
  single_letter[1] = ' ';
  if (Lookup(tr, &single_letter[2], ph_buf3) == 0)
   TranslateRules(tr, &single_letter[2], ph_buf3,
sizeof(ph_buf3), NULL, FLAG_NO_TRACE, NULL);
 }

 if (ph_buf3[0] == 0)
  LookupAccentedLetter(tr, letter, ph_buf3);

 strcpy(ph_buf1, ph_buf3);
 if ((ph_buf1[0] == 0) || (ph_buf1[0] == phonSWITCH))
  return;

 dict_flags[0] = 0;
 dict_flags[1] = 0;
```

```
 SetWordStress(tr, ph_buf1, dict_flags, -1, control & 1);
}

// unicode ranges for non-ascii digits 0-9 (these must be in
ascending order)
static const int number_ranges[] = {
 0x660, 0x6f0, // arabic
 0x966, 0x9e6, 0xa66, 0xae6, 0xb66, 0xbe6, 0xc66, 0xce6, 0xd66,
// indic
 0xe50, 0xed0, 0xf20, 0x1040, 0x1090,
 0
};

static int NonAsciiNumber(int letter)
{
 // Change non-ascii digit into ascii digit '0' to '9', (or -1 if
not)
 const int *p;
 int base;

 for (p = number_ranges; (base = *p) != 0; p++) {
  if (letter < base)
   break; // not found
  if (letter < (base+10))
   return letter-base+'0';
 }
 return -1;
}

#define L_SUB 0x4000 // subscript
#define L_SUP 0x8000 // superscript

static const char *modifiers[] = { NULL, "_sub", "_sup", NULL };

// this list must be in ascending order
static unsigned short derived_letters[] = {
 0x00aa, 'a'+L_SUP,
```

```
0x00b2, '2'+L_SUP,
0x00b3, '3'+L_SUP,
0x00b9, '1'+L_SUP,
0x00ba, 'o'+L_SUP,
0x02b0, 'h'+L_SUP,
0x02b1, 0x266+L_SUP,
0x02b2, 'j'+L_SUP,
0x02b3, 'r'+L_SUP,
0x02b4, 0x279+L_SUP,
0x02b5, 0x27b+L_SUP,
0x02b6, 0x281+L_SUP,
0x02b7, 'w'+L_SUP,
0x02b8, 'y'+L_SUP,
0x02c0, 0x294+L_SUP,
0x02c1, 0x295+L_SUP,
0x02e0, 0x263+L_SUP,
0x02e1, 'l'+L_SUP,
0x02e2, 's'+L_SUP,
0x02e3, 'x'+L_SUP,
0x2070, '0'+L_SUP,
0x2071, 'i'+L_SUP,
0x2074, '4'+L_SUP,
0x2075, '5'+L_SUP,
0x2076, '6'+L_SUP,
0x2077, '7'+L_SUP,
0x2078, '8'+L_SUP,
0x2079, '9'+L_SUP,
0x207a, '+'+L_SUP,
0x207b, '-'+L_SUP,
0x207c, '='+L_SUP,
0x207d, '('+L_SUP,
0x207e, ')'+L_SUP,
0x207f, 'n'+L_SUP,
0x2080, '0'+L_SUB,
0x2081, '1'+L_SUB,
0x2082, '2'+L_SUB,
0x2083, '3'+L_SUB,
```

```
 0x2084, '4'+L_SUB,
 0x2085, '5'+L_SUB,
 0x2086, '6'+L_SUB,
 0x2087, '7'+L_SUB,
 0x2088, '8'+L_SUB,
 0x2089, '9'+L_SUB,
 0x208a, '+'+L_SUB,
 0x208b, '-'+L_SUB,
 0x208c, '='+L_SUB,
 0x208d, '('+L_SUB,
 0x208e, ')'+L_SUB,
 0x2090, 'a'+L_SUB,
 0x2091, 'e'+L_SUB,
 0x2092, 'o'+L_SUB,
 0x2093, 'x'+L_SUB,
 0x2094, 0x259+L_SUB,
 0x2095, 'h'+L_SUB,
 0x2096, 'k'+L_SUB,
 0x2097, 'l'+L_SUB,
 0x2098, 'm'+L_SUB,
 0x2099, 'n'+L_SUB,
 0x209a, 'p'+L_SUB,
 0x209b, 's'+L_SUB,
 0x209c, 't'+L_SUB,
 0, 0
};

// names, using phonemes available to all languages
static const char *hex_letters[] = {
 "'e:j",
 "b'i:",
 "s'i:",
 "d'i:",
 "'i:",
 "'ef"
};
```

```
int IsSuperscript(int letter)
{
 // is this a subscript or superscript letter ?
 int ix;
 int c;

 for (ix = 0; (c = derived_letters[ix]) != 0; ix += 2) {
  if (c > letter)
   break;
  if (c == letter)
   return derived_letters[ix+1];
 }
 return 0;
}

int TranslateLetter(Translator *tr, char *word, char *phonemes,
int control, ALPHABET *current_alphabet)
{
 // get pronunciation for an isolated letter
 // return number of bytes used by the letter
 // control bit 0:  a non-initial letter in a word
 //         bit 1:  say 'capital'
 //         bit 2:  say character code for unknown letters

 int n_bytes;
 int letter;
 int len;
 int ix;
 int c;
 char *p2;
 char *pbuf;
 const char *modifier;
 ALPHABET *alphabet;
 int al_offset;
 int al_flags;
 int language;
 int number;
```

```
 int phontab_1;
 int speak_letter_number;
 char capital[30];
 char ph_buf[80];
 char ph_buf2[80];
 char ph_alphabet[80];
 char hexbuf[12];
 static char pause_string[] = { phonPAUSE, 0 };

 ph_buf[0] = 0;
 ph_alphabet[0] = 0;
 capital[0] = 0;
 phontab_1 = translator->phoneme_tab_ix;

 n_bytes = utf8_in(&letter, word);

 if ((letter & 0xfff00) == 0x0e000)
  letter &= 0xff; // uncode private usage area

 if (control & 2) {
  // include CAPITAL information
  if (iswupper(letter))
   Lookup(tr, "_cap", capital);
 }
 letter = towlower2(letter, tr);
 LookupLetter(tr, letter, word[n_bytes], ph_buf, control & 1);

 if (ph_buf[0] == 0) {
  // is this a subscript or superscript letter ?
  if ((c = IsSuperscript(letter)) != 0) {
   letter = c & 0x3fff;
   if ((control & 4 ) && ((modifier = modifiers[c >> 14]) !=
NULL)) {
    // don't say "superscript" during normal text reading
    Lookup(tr, modifier, capital);
    if (capital[0] == 0) {
     capital[2] = SetTranslator2("en"); // overwrites previous
```

```
contents of translator2
     Lookup(translator2, modifier, &capital[3]);
     if (capital[3] != 0) {
      capital[0] = phonPAUSE;
      capital[1] = phonSWITCH;
      len = strlen(&capital[3]);
      capital[len+3] = phonSWITCH;
      capital[len+4] = phontab_1;
      capital[len+5] = 0;
     }
    }
   }
  }
  LookupLetter(tr, letter, word[n_bytes], ph_buf, control & 1);
 }

 if (ph_buf[0] == phonSWITCH) {
  strcpy(phonemes, ph_buf);
  return 0;
 }

 if ((ph_buf[0] == 0) && ((number = NonAsciiNumber(letter)) > 0))
{
  // convert a non-ascii number to 0-9
  LookupLetter(tr, number, 0, ph_buf, control & 1);
 }

 al_offset = 0;
 al_flags = 0;
 if ((alphabet = AlphabetFromChar(letter)) != NULL) {
  al_offset = alphabet->offset;
  al_flags = alphabet->flags;
 }

 if (alphabet != current_alphabet) {
  // speak the name of the alphabet
  current_alphabet = alphabet;
```

```
  if ((alphabet != NULL) && !(al_flags & AL_DONT_NAME) &&
(al_offset != translator->letter_bits_offset)) {
   if ((al_flags & AL_DONT_NAME) || (al_offset ==
translator->langopts.alt_alphabet) || (al_offset ==
translator->langopts.our_alphabet)) {
    // don't say the alphabet name
   } else {
    ph_buf2[0] = 0;
    if (Lookup(translator, alphabet->name, ph_alphabet) == 0) {
// the original language for the current voice
     // Can't find the local name for this alphabet, use the
English name
     ph_alphabet[2] = SetTranslator2("en"); // overwrites
previous contents of translator2
     Lookup(translator2, alphabet->name, ph_buf2);
    } else if (translator != tr) {
     phontab_1 = tr->phoneme_tab_ix;
     strcpy(ph_buf2, ph_alphabet);
     ph_alphabet[2] = translator->phoneme_tab_ix;
    }

    if (ph_buf2[0] != 0) {
     // we used a different language for the alphabet name (now
in ph_buf2)
     ph_alphabet[0] = phonPAUSE;
     ph_alphabet[1] = phonSWITCH;
     strcpy(&ph_alphabet[3], ph_buf2);
     len = strlen(ph_buf2) + 3;
     ph_alphabet[len] = phonSWITCH;
     ph_alphabet[len+1] = phontab_1;
     ph_alphabet[len+2] = 0;
    }
   }
  }
 }

 // caution: SetWordStress() etc don't expect phonSWITCH +
```

```
phoneme table number

 if (ph_buf[0] == 0) {
  if ((al_offset != 0) && (al_offset ==
translator->langopts.alt_alphabet))
   language = translator->langopts.alt_alphabet_lang;
  else if ((alphabet != NULL) && (alphabet->language != 0) &&
!(al_flags & AL_NOT_LETTERS))
   language = alphabet->language;
  else
   language = L('e', 'n');

  if ((language != tr->translator_name) || (language == L('k',
'o'))) {
   char *p3;
   int initial, code;
   char hangul_buf[12];

   // speak in the language for this alphabet (or English)
   ph_buf[2] = SetTranslator2(WordToString2(language));

   if (translator2 != NULL) {
    if (((code = letter - 0xac00) >= 0) && (letter <= 0xd7af)) {
     // Special case for Korean letters.
     // break a syllable hangul into 2 or 3 individual jamo

     hangul_buf[0] = ' ';
     p3 = &hangul_buf[1];
     if ((initial = (code/28)/21) != 11) {
      p3 += utf8_out(initial + 0x1100, p3);
     }
     utf8_out(((code/28) % 21) + 0x1161, p3); // medial
     utf8_out((code % 28) + 0x11a7, &p3[3]); // final
     p3[6] = ' ';
     p3[7] = 0;
     ph_buf[3] = 0;
     TranslateRules(translator2, &hangul_buf[1], &ph_buf[3],
```

```
sizeof(ph_buf)-3, NULL, 0, NULL);
     SetWordStress(translator2, &ph_buf[3], NULL, -1, 0);
    } else
     LookupLetter(translator2, letter, word[n_bytes], &ph_buf[3],
control & 1);

    if (ph_buf[3] == phonSWITCH) {
     // another level of language change
     ph_buf[2] = SetTranslator2(&ph_buf[4]);
     LookupLetter(translator2, letter, word[n_bytes], &ph_buf[3],
control & 1);
    }

    SelectPhonemeTable(voice->phoneme_tab_ix); // revert to
original phoneme table

    if (ph_buf[3] != 0) {
     ph_buf[0] = phonPAUSE;
     ph_buf[1] = phonSWITCH;
     len = strlen(&ph_buf[3]) + 3;
     ph_buf[len] = phonSWITCH; // switch back
     ph_buf[len+1] = tr->phoneme_tab_ix;
     ph_buf[len+2] = 0;
    }
   }
  }
 }

 if (ph_buf[0] == 0) {
  // character name not found

  if (ph_buf[0] == 0) {
   speak_letter_number = 1;
   if (!(al_flags & AL_NO_SYMBOL)) {
    if (iswalpha(letter))
     Lookup(translator, "_?A", ph_buf);
```

```
    if ((ph_buf[0] == 0) && !iswspace(letter))
     Lookup(translator, "_??", ph_buf);

    if (ph_buf[0] == 0)
     EncodePhonemes("l'et@", ph_buf, NULL);
   }

   if (!(control & 4) && (al_flags & AL_NOT_CODE)) {
    // don't speak the character code number, unless we want full
details of this character
    speak_letter_number = 0;
   }

   if (speak_letter_number) {
    if (al_offset == 0x2800) {
     // braille dots symbol, list the numbered dots
     p2 = hexbuf;
     for (ix = 0; ix < 8; ix++) {
      if (letter & (1 << ix))
       *p2++ = '1'+ix;
     }
     *p2 = 0;
    } else {
     // speak the hexadecimal number of the character code
     sprintf(hexbuf, "%x", letter);
    }

    pbuf = ph_buf;
    for (p2 = hexbuf; *p2 != 0; p2++) {
     pbuf += strlen(pbuf);
     *pbuf++ = phonPAUSE_VSHORT;
     LookupLetter(translator, *p2, 0, pbuf, 1);
     if (((pbuf[0] == 0) || (pbuf[0] == phonSWITCH)) && (*p2 >=
'a')) {
      // This language has no translation for 'a' to 'f', speak
English names using base phonemes
      EncodePhonemes(hex_letters[*p2 - 'a'], pbuf, NULL);
```

```
    }
   }
   strcat(pbuf, pause_string);
  }
 }
}

len = strlen(phonemes);

if (tr->langopts.accents & 2)  // 'capital' before or after the
word ?
 sprintf(ph_buf2, "%c%s%s%s", 0xff, ph_alphabet, ph_buf,
capital);
else
 sprintf(ph_buf2, "%c%s%s%s", 0xff, ph_alphabet, capital,
ph_buf); // the 0xff marker will be removed or replaced in
SetSpellingStress()
if ((len + strlen(ph_buf2)) < N_WORD_PHONEMES)
 strcpy(&phonemes[len], ph_buf2);
return n_bytes;
}

void SetSpellingStress(Translator *tr, char *phonemes, int
control, int n_chars)
{
 // Individual letter names, reduce the stress of some.
 int ix;
 unsigned int c;
 int n_stress = 0;
 int prev = 0;
 int count;
 unsigned char buf[N_WORD_PHONEMES];

 for (ix = 0; (c = phonemes[ix]) != 0; ix++) {
  if ((c == phonSTRESS_P) && (prev != phonSWITCH))
   n_stress++;
  buf[ix] = prev = c;
```

```
 }
 buf[ix] = 0;

 count = 0;
 prev = 0;
 for (ix = 0; (c = buf[ix]) != 0; ix++) {
  if ((c == phonSTRESS_P) && (n_chars > 1) && (prev !=
phonSWITCH)) {
   count++;

   if (tr->langopts.spelling_stress == 1) {
    // stress on initial letter when spelling
    if (count > 1)
     c = phonSTRESS_3;
   } else {
    if (count != n_stress) {
     if (((count % 3) != 0) || (count == n_stress-1))
      c = phonSTRESS_3; // reduce to secondary stress
    }
   }
  } else if (c == 0xff) {
   if ((control < 2) || (ix == 0))
    continue; // don't insert pauses

   if (((count % 3) == 0) || (control > 2))
    c = phonPAUSE_NOLINK; // pause following a primary stress
   else
    c = phonPAUSE_VSHORT;
  }
  *phonemes++ = prev = c;
 }
 if (control >= 2)
  *phonemes++ = phonPAUSE_NOLINK;

}

// Numbers
```

```
static char ph_ordinal2[12];
static char ph_ordinal2x[12];

static int CheckDotOrdinal(Translator *tr, char *word, char
*word_end, WORD_TAB *wtab, int roman)
{
 int ordinal = 0;
 int c2;
 int nextflags;

 if ((tr->langopts.numbers & NUM_ORDINAL_DOT) && ((word_end[0] ==
'.') || (wtab[0].flags & FLAG_HAS_DOT)) && !(wtab[1].flags &
FLAG_NOSPACE)) {
  if (roman || !(wtab[1].flags & FLAG_FIRST_UPPER)) {
   if (word_end[0] == '.')
    utf8_in(&c2, &word_end[2]);
   else
    utf8_in(&c2, &word_end[0]);

   if ((word_end[0] != 0) && (word_end[1] != 0) && ((c2 == 0) ||
(wtab[0].flags & FLAG_COMMA_AFTER) || IsAlpha(c2))) {
    // ordinal number is indicated by dot after the number
    // but not if the next word starts with an upper-case letter
    // (c2 == 0) is for cases such as, "2.,"
    ordinal = 2;
    if (word_end[0] == '.')
     word_end[0] = ' ';

    if ((roman == 0) && (tr->translator_name == L('h', 'u'))) {
     // lang=hu don't treat dot as ordinal indicator if the next
word is a month name ($alt). It may have a suffix.
     nextflags = 0;
     if (IsAlpha(c2))
      nextflags = TranslateWord(tr, &word_end[2], NULL, NULL);

     if ((tr->prev_dict_flags[0] & FLAG_ALT_TRANS) && ((c2 == 0)
```

```
|| (wtab[0].flags & FLAG_COMMA_AFTER) || iswdigit(c2)))
      ordinal = 0; // TEST  09.02.10

     if (nextflags & FLAG_ALT_TRANS)
      ordinal = 0;

     if (nextflags & FLAG_ALT3_TRANS) {
      if (word[-2] == '-')
       ordinal = 0; // e.g. december 2-5. között

      if (tr->prev_dict_flags[0] & (FLAG_ALT_TRANS |
FLAG_ALT3_TRANS))
       ordinal = 0x22;
     }
    }
   }
  }
 }
 return ordinal;
}

static int hu_number_e(const char *word, int thousandplex, int
value)
{
 // lang-hu: variant form of numbers when followed by hyphen and
a suffix starting with 'a' or 'e' (but not a, e, az, ez, azt,
ezt, att. ett

 if ((word[0] == 'a') || (word[0] == 'e')) {
  if ((word[1] == ' ') || (word[1] == 'z') || ((word[1] == 't')
&& (word[2] == 't')))
   return 0;

  if (((thousandplex == 1) || ((value % 1000) == 0)) && (word[1]
== 'l'))
   return 0; // 1000-el
```

```
  return 1;
 }
 return 0;
}

int TranslateRoman(Translator *tr, char *word, char *ph_out,
WORD_TAB *wtab)
{
 int c;
 char *p;
 const char *p2;
 int acc;
 int prev;
 int value;
 int subtract;
 int repeat = 0;
 int n_digits = 0;
 char *word_start;
 int num_control = 0;
 unsigned int flags[2];
 char ph_roman[30];
 char number_chars[N_WORD_BYTES];

 static const char *roman_numbers = "ixcmvld";
 static int roman_values[] = { 1, 10, 100, 1000, 5, 50, 500 };

 acc = 0;
 prev = 0;
 subtract = 0x7fff;
 ph_out[0] = 0;
 flags[0] = 0;
 flags[1] = 0;

 if (((tr->langopts.numbers & NUM_ROMAN_CAPITALS) &&
!(wtab[0].flags & FLAG_ALL_UPPER)) || IsDigit09(word[-2]))
  return 0; // not '2xx'
```

```
 if (word[1] == ' ') {
  if ((tr->langopts.numbers & (NUM_ROMAN_CAPITALS |
NUM_ROMAN_ORDINAL | NUM_ORDINAL_DOT)) && (wtab[0].flags &
FLAG_HAS_DOT)) {
   // allow single letter Roman ordinal followed by dot.
  } else
   return 0; // only one letter, don't speak as a Roman Number
 }

 word_start = word;
 while ((c = *word++) != ' ') {
  if ((p2 = strchr(roman_numbers, c)) == NULL)
   return 0;

  value = roman_values[p2 - roman_numbers];
  if (value == prev) {
   repeat++;
   if (repeat >= 3)
    return 0;
  } else
   repeat = 0;

  if ((prev > 1) && (prev != 10) && (prev != 100)) {
   if (value >= prev)
    return 0;
  }
  if ((prev != 0) && (prev < value)) {
   if (((acc % 10) != 0) || ((prev*10) < value))
    return 0;
   subtract = prev;
   value -= subtract;
  } else if (value >= subtract)
   return 0;
  else
   acc += prev;
  prev = value;
  n_digits++;
```

```
 }

 if (IsDigit09(word[0]))
  return 0; // e.g. 'xx2'

 acc += prev;
 if (acc < tr->langopts.min_roman)
  return 0;

 if (acc > tr->langopts.max_roman)
  return 0;

 Lookup(tr, "_roman", ph_roman); // precede by "roman" if _rom is
defined in *_list
 p = &ph_out[0];

 if ((tr->langopts.numbers & NUM_ROMAN_AFTER) == 0) {
  strcpy(ph_out, ph_roman);
  p = &ph_out[strlen(ph_roman)];
 }

 sprintf(number_chars, "  %d %s    ", acc,
tr->langopts.roman_suffix);

 if (word[0] == '.') {
  // dot has not been removed.  This implies that there was no
space after it
  return 0;
 }

 if (CheckDotOrdinal(tr, word_start, word, wtab, 1))
  wtab[0].flags |= FLAG_ORDINAL;

 if (tr->langopts.numbers & NUM_ROMAN_ORDINAL) {
  if (tr->translator_name == L('h', 'u')) {
   if (!(wtab[0].flags & FLAG_ORDINAL)) {
    if ((wtab[0].flags & FLAG_HYPHEN_AFTER) && hu_number_e(word,
```

```
0, acc)) {
     // should use the 'e' form of the number
     num_control |= 1;
    } else
     return 0;
   }
  } else
   wtab[0].flags |= FLAG_ORDINAL;
 }

 tr->prev_dict_flags[0] = 0;
 tr->prev_dict_flags[1] = 0;
 TranslateNumber(tr, &number_chars[2], p, flags, wtab,
num_control);

 if (tr->langopts.numbers & NUM_ROMAN_AFTER)
  strcat(ph_out, ph_roman);

 return 1;
}

static const char *M_Variant(int value)
{
 // returns M, or perhaps MA or MB for some cases

 bool teens = false;

 if (((value % 100) > 10) && ((value % 100) < 20))
  teens = true;

 switch ((translator->langopts.numbers2 >> 6) & 0x7)
 {
 case 1: // lang=ru  use singular for xx1 except for x11
  if ((teens == false) && ((value % 10) == 1))
   return "1M";
  break;
 case 2: // lang=cs,sk
```

```
  if ((value >= 2) && (value <= 4))
   return "0MA";
  break;
 case 3: // lang=pl
  if ((teens == false) && (((value % 10) >= 2) && ((value % 10)
<= 4)))
   return "0MA";
  break;
 case 4: // lang=lt
  if ((teens == true) || ((value % 10) == 0))
   return "0MB";
  if ((value % 10) == 1)
   return "0MA";
  break;
 case 5: // lang=bs,hr,sr
  if (teens == false) {
   if ((value % 10) == 1)
    return "1M";
   if (((value % 10) >= 2) && ((value % 10) <= 4))
    return "0MA";
  }
  break;
 }
 return "0M";
}

static int LookupThousands(Translator *tr, int value, int
thousandplex, int thousands_exact, char *ph_out)
{
 // thousands_exact:  bit 0  no hundreds,tens,or units,  bit 1
ordinal numberr
 int found;
 int found_value = 0;
 char string[12];
 char ph_of[12];
 char ph_thousands[40];
 char ph_buf[40];
```

```
ph_of[0] = 0;

// first look for a match with the exact value of thousands
if (value > 0) {
 if (thousands_exact & 1) {
  if (thousands_exact & 2) {
   // ordinal number
   sprintf(string, "_%dM%do", value, thousandplex);
   found_value = Lookup(tr, string, ph_thousands);
  }
  if (!found_value && (number_control & 1)) {
   // look for the 'e' variant
   sprintf(string, "_%dM%de", value, thousandplex);
   found_value = Lookup(tr, string, ph_thousands);
  }
  if (!found_value) {
   // is there a different pronunciation if there are no
hundreds,tens,or units ? (LANG=ta)
   sprintf(string, "_%dM%dx", value, thousandplex);
   found_value = Lookup(tr, string, ph_thousands);
  }
 }
 if (found_value == 0) {
  sprintf(string, "_%dM%d", value, thousandplex);
  found_value = Lookup(tr, string, ph_thousands);
 }
}

if (found_value == 0) {
 if ((value % 100) >= 20)
  Lookup(tr, "_0of", ph_of);

 found = 0;
 if (thousands_exact & 1) {
  if (thousands_exact & 2) {
   // ordinal number
```

```
    sprintf(string, "_%s%do", M_Variant(value), thousandplex);
    found = Lookup(tr, string, ph_thousands);
   }
   if (!found && (number_control & 1)) {
    // look for the 'e' variant
    sprintf(string, "_%s%de", M_Variant(value), thousandplex);
    found = Lookup(tr, string, ph_thousands);
   }
   if (!found) {
    // is there a different pronunciation if there are no
hundreds,tens,or units ?
    sprintf(string, "_%s%dx", M_Variant(value), thousandplex);
    found = Lookup(tr, string, ph_thousands);
   }
  }
  if (found == 0) {
   sprintf(string, "_%s%d", M_Variant(value), thousandplex);

   if (Lookup(tr, string, ph_thousands) == 0) {
    if (thousandplex > 3) {
     sprintf(string, "_0M%d", thousandplex-1);
     if (Lookup(tr, string, ph_buf) == 0) {
      // say "millions" if this name is not available and neither
is the next lower
      Lookup(tr, "_0M2", ph_thousands);
      speak_missing_thousands = 3;
     }
    }
    if (ph_thousands[0] == 0) {
     // repeat "thousand" if higher order names are not available
     sprintf(string, "_%dM1", value);
     if ((found_value = Lookup(tr, string, ph_thousands)) == 0)
      Lookup(tr, "_0M1", ph_thousands);
     speak_missing_thousands = 2;
    }
   }
  }
```

```
 }
 sprintf(ph_out, "%s%s", ph_of, ph_thousands);

 if ((value == 1) && (thousandplex == 1) && (tr->langopts.numbers
& NUM_OMIT_1_THOUSAND))
  return 1;

 return found_value;
}

static int LookupNum2(Translator *tr, int value, int
thousandplex, const int control, char *ph_out)
{
 // Lookup a 2 digit number
 // control bit 0: ordinal number
 // control bit 1: final tens and units (not number of thousands)
(use special form of '1', LANG=de "eins")
 // control bit 2: tens and units only, no higher digits
 // control bit 3: use feminine form of '2' (for thousands
 // control bit 4: speak zero tens
 // control bit 5: variant of ordinal number (lang=hu)
 //         bit 8   followed by decimal fraction
 //         bit 9: use #f form for both tens and units (lang=ml)

 int found;
 int ix;
 int units;
 int tens;
 int is_ordinal;
 int used_and = 0;
 int found_ordinal = 0;
 int next_phtype;
 int ord_type = 'o';
 char string[12]; // for looking up entries in *_list
 char ph_ordinal[20];
 char ph_tens[50];
 char ph_digits[50];
```

```
char ph_and[12];

units = value % 10;
tens = value / 10;

found = 0;
ph_ordinal[0] = 0;
ph_tens[0] = 0;
ph_digits[0] = 0;
ph_and[0] = 0;

if (control & 0x20)
 ord_type = 'q';

is_ordinal = control & 1;

if ((control & 2) && (n_digit_lookup == 2)) {
 // pronunciation of the final 2 digits has already been found
 strcpy(ph_out, digit_lookup);
} else {
 if (digit_lookup[0] == 0) {
  // is there a special pronunciation for this 2-digit number
  if (control & 8) {
   // is there a feminine or thousands-variant form?
   sprintf(string, "_%dfx", value);
   if ((found = Lookup(tr, string, ph_digits)) == 0) {
    sprintf(string, "_%df", value);
    found = Lookup(tr, string, ph_digits);
   }
  } else if (is_ordinal) {
   strcpy(ph_ordinal, ph_ordinal2);

   if (control & 4) {
    sprintf(string, "_%d%cx", value, ord_type); // LANG=hu,
special word for 1. 2. when there are no higher digits
    if ((found = Lookup(tr, string, ph_digits)) != 0) {
     if (ph_ordinal2x[0] != 0)
```

```
       strcpy(ph_ordinal, ph_ordinal2x); // alternate
pronunciation (lang=an)
     }
    }
    if (found == 0) {
     sprintf(string, "_%d%c", value, ord_type);
     found = Lookup(tr, string, ph_digits);
    }
    found_ordinal = found;
   }

   if (found == 0) {
    if (control & 2) {
     // the final tens and units of a number
     if (number_control & 1) {
      // look for 'e' variant
      sprintf(string, "_%de", value);
      found = Lookup(tr, string, ph_digits);
     }
    } else {
     // followed by hundreds or thousands etc
     if ((tr->langopts.numbers2 & NUM2_ORDINAL_AND_THOUSANDS) &&
(thousandplex <= 1))
      sprintf(string, "_%do", value); // LANG=TA
     else
      sprintf(string, "_%da", value);
     found = Lookup(tr, string, ph_digits);
    }

    if (!found) {
     if ((is_ordinal) && (tr->langopts.numbers2 &
NUM2_NO_TEEN_ORDINALS)) {
      // don't use numbers 10-99 to make ordinals, always use
_1Xo etc (lang=pt)
     } else {
      sprintf(string, "_%d", value);
      found = Lookup(tr, string, ph_digits);
```

```
      }
     }
    }
   }

   // no, speak as tens+units

   if ((value < 10) && (control & 0x10)) {
    // speak leading zero
    Lookup(tr, "_0", ph_tens);
   } else {
    if (found)
     ph_tens[0] = 0;
    else {
     if (is_ordinal) {
      sprintf(string, "_%dX%c", tens, ord_type);
      if (Lookup(tr, string, ph_tens) != 0) {
       found_ordinal = 1;

       if ((units != 0) && (tr->langopts.numbers2 &
NUM2_MULTIPLE_ORDINAL)) {
        // Use the ordinal form of tens as well as units. Add the
ordinal ending
        strcat(ph_tens, ph_ordinal2);
       }
      }
     }
     if (found_ordinal == 0) {
      if (control & 0x200)
       sprintf(string, "_%dXf", tens);
      else
       sprintf(string, "_%dX", tens);
      Lookup(tr, string, ph_tens);
     }

     if ((ph_tens[0] == 0) && (tr->langopts.numbers &
NUM_VIGESIMAL)) {
```

```
     // tens not found,  (for example) 73 is 60+13
     units = (value % 20);
     sprintf(string, "_%dX", tens & 0xfe);
     Lookup(tr, string, ph_tens);
    }

    ph_digits[0] = 0;
    if (units > 0) {
     found = 0;

     if ((control & 2) && (digit_lookup[0] != 0)) {
      // we have an entry for this digit (possibly together with
the next word)
      strcpy(ph_digits, digit_lookup);
      found_ordinal = 1;
      ph_ordinal[0] = 0;
     } else {
      if (control & 8) {
       // is there a variant form of this number?
       sprintf(string, "_%df", units);
       found = Lookup(tr, string, ph_digits);
      }
      if ((is_ordinal) && ((tr->langopts.numbers & NUM_SWAP_TENS)
== 0)) {
       // ordinal
       sprintf(string, "_%d%c", units, ord_type);
       if ((found = Lookup(tr, string, ph_digits)) != 0)
        found_ordinal = 1;
      }
      if (found == 0) {
       if ((number_control & 1) && (control & 2)) {
        // look for 'e' variant
        sprintf(string, "_%de", units);
        found = Lookup(tr, string, ph_digits);
       } else if (((control & 2) == 0) || ((tr->langopts.numbers
& NUM_SWAP_TENS) != 0)) {
        // followed by hundreds or thousands (or tens)
```

```
        if ((tr->langopts.numbers2 & NUM2_ORDINAL_AND_THOUSANDS)
&& (thousandplex <= 1))
         sprintf(string, "_%do", units);  // LANG=TA,  only for
100s, 1000s
        else
         sprintf(string, "_%da", units);
        found = Lookup(tr, string, ph_digits);
       }
      }
      if (found == 0) {
       sprintf(string, "_%d", units);
       Lookup(tr, string, ph_digits);
      }
     }
    }
   }
  }

  if ((is_ordinal) && (found_ordinal == 0) && (ph_ordinal[0] ==
0)) {
   if ((value >= 20) && (((value % 10) == 0) ||
(tr->langopts.numbers & NUM_SWAP_TENS)))
    Lookup(tr, "_ord20", ph_ordinal);
   if (ph_ordinal[0] == 0)
    Lookup(tr, "_ord", ph_ordinal);
  }

  if ((tr->langopts.numbers & (NUM_SWAP_TENS | NUM_AND_UNITS)) &&
(ph_tens[0] != 0) && (ph_digits[0] != 0)) {
   Lookup(tr, "_0and", ph_and);

   if ((is_ordinal) && (tr->langopts.numbers2 &
NUM2_ORDINAL_NO_AND))
    ph_and[0] = 0;

   if (tr->langopts.numbers & NUM_SWAP_TENS)
    sprintf(ph_out, "%s%s%s%s", ph_digits, ph_and, ph_tens,
```

```
ph_ordinal);
   else
    sprintf(ph_out, "%s%s%s%s", ph_tens, ph_and, ph_digits,
ph_ordinal);
   used_and = 1;
  } else {
   if (tr->langopts.numbers & NUM_SINGLE_VOWEL) {
    // remove vowel from the end of tens if units starts with a
vowel (LANG=Italian)
    if (((ix = strlen(ph_tens)-1) >= 0) && (ph_digits[0] != 0)) {
     if ((next_phtype = phoneme_tab[(unsigned
int)(ph_digits[0])]->type) == phSTRESS)
      next_phtype = phoneme_tab[(unsigned
int)(ph_digits[1])]->type;

     if ((phoneme_tab[(unsigned int)(ph_tens[ix])]->type ==
phVOWEL) && (next_phtype == phVOWEL))
      ph_tens[ix] = 0;
    }
   }

   if ((tr->langopts.numbers2 & NUM2_ORDINAL_DROP_VOWEL) &&
(ph_ordinal[0] != 0)) {
    ix = sprintf(ph_out, "%s%s", ph_tens, ph_digits);
    if ((ix > 0) && (phoneme_tab[(unsigned
char)(ph_out[ix-1])]->type == phVOWEL))
     ix--;
    sprintf(&ph_out[ix], "%s", ph_ordinal);
   } else
    sprintf(ph_out, "%s%s%s", ph_tens, ph_digits, ph_ordinal);
  }
 }

 if (tr->langopts.numbers & NUM_SINGLE_STRESS_L) {
  // only one primary stress, on the first part (tens)
  found = 0;
  for (ix = 0; ix < (signed)strlen(ph_out); ix++) {
```

```
   if (ph_out[ix] == phonSTRESS_P) {
    if (found)
     ph_out[ix] = phonSTRESS_3;
    else
     found = 1;
   }
  }
 } else if (tr->langopts.numbers & NUM_SINGLE_STRESS) {
  // only one primary stress
  found = 0;
  for (ix = strlen(ph_out)-1; ix >= 0; ix--) {
   if (ph_out[ix] == phonSTRESS_P) {
    if (found)
     ph_out[ix] = phonSTRESS_3;
    else
     found = 1;
   }
  }
 }
 return used_and;
}

static int LookupNum3(Translator *tr, int value, char *ph_out,
bool suppress_null, int thousandplex, int control)
{
 // Translate a 3 digit number
 //  control  bit 0,  previous thousands
 //           bit 1,  ordinal number
 //           bit 5   variant form of ordinal number
 //           bit 8   followed by decimal fraction

 int found;
 int hundreds;
 int tensunits;
 int x;
 int ix;
 int exact;
```

```
int ordinal;
int tplex;
bool say_zero_hundred = false;
bool say_one_hundred;
char string[12]; // for looking up entries in **_list
char buf1[100];
char buf2[100];
char ph_100[20];
char ph_10T[20];
char ph_digits[50];
char ph_thousands[50];
char ph_hundred_and[12];
char ph_thousand_and[12];

ordinal = control & 0x22;
hundreds = value / 100;
tensunits = value % 100;
buf1[0] = 0;

ph_thousands[0] = 0;
ph_thousand_and[0] = 0;

if ((tr->langopts.numbers & NUM_ZERO_HUNDRED) && ((control & 1)
|| (hundreds >= 10)))
 say_zero_hundred = true; // lang=vi

if ((hundreds > 0) || say_zero_hundred) {
 found = 0;
 if (ordinal && (tensunits == 0)) {
  // ordinal number, with no tens or units
  found = Lookup(tr, "_0Co", ph_100);
 }
 if (found == 0) {
  if (tensunits == 0) {
   // special form for exact hundreds?
   found = Lookup(tr, "_0C0", ph_100);
  }
```

```
   if (!found)
    Lookup(tr, "_0C", ph_100);
  }

  if (((tr->langopts.numbers & NUM_1900) != 0) && (hundreds ==
19)) {
   // speak numbers such as 1984 as years: nineteen-eighty-four
  } else if (hundreds >= 10) {
   ph_digits[0] = 0;

   exact = 0;
   if ((value % 1000) == 0)
    exact = 1;

   tplex = thousandplex+1;
   if (tr->langopts.numbers2 & NUM2_MYRIADS)
    tplex = 0;

   if (LookupThousands(tr, hundreds / 10, tplex, exact | ordinal,
ph_10T) == 0) {
    x = 0;
    if (tr->langopts.numbers2 & (1 << tplex))
     x = 8; // use variant (feminine) for before thousands and
millions
    if (tr->translator_name == L('m', 'l'))
     x = 0x208;
    LookupNum2(tr, hundreds/10, thousandplex, x, ph_digits);
   }

   if (tr->langopts.numbers2 & 0x200)
    sprintf(ph_thousands, "%s%c%s%c", ph_10T, phonEND_WORD,
ph_digits, phonEND_WORD); // say "thousands" before its number,
not after
   else
    sprintf(ph_thousands, "%s%c%s%c", ph_digits, phonEND_WORD,
ph_10T, phonEND_WORD);
```

```
   hundreds %= 10;
   if ((hundreds == 0) && (say_zero_hundred == false))
    ph_100[0] = 0;
   suppress_null = true;
   control |= 1;
  }

  ph_digits[0] = 0;

  if ((hundreds > 0) || say_zero_hundred) {
   if ((tr->langopts.numbers & NUM_AND_HUNDRED) && ((control & 1)
|| (ph_thousands[0] != 0)))
    Lookup(tr, "_0and", ph_thousand_and);

   suppress_null = true;

   found = 0;
   if ((ordinal)
       && ((tensunits == 0) || (tr->langopts.numbers2 &
NUM2_MULTIPLE_ORDINAL))) {
    // ordinal number
    sprintf(string, "_%dCo", hundreds);
    found = Lookup(tr, string, ph_digits);

    if ((tr->langopts.numbers2 & NUM2_MULTIPLE_ORDINAL) &&
(tensunits > 0)) {
     // Use ordinal form of hundreds, as well as for tens and
units
     // Add ordinal suffix to the hundreds
     strcat(ph_digits, ph_ordinal2);
    }
   }

   if ((hundreds == 0) && say_zero_hundred)
    Lookup(tr, "_0", ph_digits);
   else {
    if ((hundreds == 1) && (tr->langopts.numbers2 &
```

```
NUM2_OMIT_1_HUNDRED_ONLY) && ((control & 1) == 0)) {
     // only look for special 100 if there are previous thousands
    } else {
     if ((!found) && (tensunits == 0)) {
      // is there a special pronunciation for exactly n00 ?
      sprintf(string, "_%dC0", hundreds);
      found = Lookup(tr, string, ph_digits);
     }

     if (!found) {
      sprintf(string, "_%dC", hundreds);
      found = Lookup(tr, string, ph_digits);  // is there a
specific pronunciation for n-hundred ?
     }
    }

    if (found)
     ph_100[0] = 0;
    else {
     say_one_hundred = true;
     if (hundreds == 1) {
      if ((tr->langopts.numbers & NUM_OMIT_1_HUNDRED) != 0)
       say_one_hundred = false;
     }

     if (say_one_hundred == true)
      LookupNum2(tr, hundreds, thousandplex, 0, ph_digits);
    }
   }
  }

  sprintf(buf1, "%s%s%s%s", ph_thousands, ph_thousand_and,
ph_digits, ph_100);
 }

 ph_hundred_and[0] = 0;
 if (tensunits > 0) {
```

```
  if ((control & 2) && (tr->langopts.numbers2 &
NUM2_MULTIPLE_ORDINAL)) {
   // Don't use "and" if we apply ordinal to both hundreds and
units
  } else {
   if ((value > 100) || ((control & 1) && (thousandplex == 0))) {
    if ((tr->langopts.numbers & NUM_HUNDRED_AND) ||
((tr->langopts.numbers & NUM_HUNDRED_AND_DIGIT) && (tensunits <
10)))
     Lookup(tr, "_0and", ph_hundred_and);
   }
   if ((tr->langopts.numbers & NUM_THOUSAND_AND) && (hundreds ==
0) && ((control & 1) || (ph_thousands[0] != 0)))
    Lookup(tr, "_0and", ph_hundred_and);
  }
 }

 buf2[0] = 0;

 if ((tensunits != 0) || (suppress_null == false)) {
  x = 0;
  if (thousandplex == 0) {
   x = 2; // allow "eins" for 1 rather than "ein"
   if (ordinal)
    x = 3; // ordinal number
   if ((value < 100) && !(control & 1))
    x |= 4; // tens and units only, no higher digits
   if (ordinal & 0x20)
    x |= 0x20; // variant form of ordinal number
  } else if (tr->langopts.numbers2 & (1 << thousandplex))
   x = 8; // use variant (feminine) for before thousands and
millions

  if ((tr->translator_name == L('m', 'l')) && (thousandplex ==
1))
   x |= 0x208; // use #f form for both tens and units
```

```
  if ((tr->langopts.numbers2 & NUM2_ZERO_TENS) && ((control & 1)
|| (hundreds > 0))) {
   // LANG=zh,
   x |= 0x10;
  }

  if (LookupNum2(tr, tensunits, thousandplex, x | (control &
0x100), buf2) != 0) {
   if (tr->langopts.numbers & NUM_SINGLE_AND)
    ph_hundred_and[0] = 0; // don't put 'and' after 'hundred' if
there's 'and' between tens and units
  }
 } else {
  if (ph_ordinal2[0] != 0) {
   ix = strlen(buf1);
   if ((ix > 0) && (buf1[ix-1] == phonPAUSE_SHORT))
    buf1[ix-1] = 0; // remove pause before addding ordinal suffix
   strcpy(buf2, ph_ordinal2);
  }
 }

 sprintf(ph_out, "%s%s%c%s", buf1, ph_hundred_and, phonEND_WORD,
buf2);

 return 0;
}

static bool CheckThousandsGroup(char *word, int group_len)
{
 // Is this a group of 3 digits which looks like a thousands
group?
 int ix;

 if (IsDigit09(word[group_len]) || IsDigit09(-1))
  return false;

 for (ix = 0; ix < group_len; ix++) {
```

```
  if (!IsDigit09(word[ix]))
   return false;
 }
 return true;
}

static int TranslateNumber_1(Translator *tr, char *word, char
*ph_out, unsigned int *flags, WORD_TAB *wtab, int control)
{
 //  Number translation with various options
 // the "word" may be up to 4 digits
 // "words" of 3 digits may be preceded by another number "word"
for thousands or millions

 int n_digits;
 int value;
 int ix;
 int digix;
 unsigned char c;
 bool suppress_null = false;
 int decimal_point = 0;
 int thousandplex = 0;
 int thousands_exact = 1;
 int thousands_inc = 0;
 int prev_thousands = 0;
 int ordinal = 0;
 int this_value;
 int decimal_count;
 int max_decimal_count;
 int decimal_mode;
 int suffix_ix;
 int skipwords = 0;
 int group_len;
 int len;
 char *p;
 char string[32]; // for looking up entries in **_list
 char buf1[100];
```

```
 char ph_append[50];
 char ph_buf[200];
 char ph_buf2[50];
 char ph_zeros[50];
 char suffix[30]; // string[] must be long enough for
sizeof(suffix)+2
 char buf_digit_lookup[50];

 static const char str_pause[2] = { phonPAUSE_NOLINK, 0 };

 n_digit_lookup = 0;
 buf_digit_lookup[0] = 0;
 digit_lookup = buf_digit_lookup;
 number_control = control;

 for (ix = 0; IsDigit09(word[ix]); ix++) ;
 n_digits = ix;
 value = this_value = atoi(word);

 group_len = 3;
 if (tr->langopts.numbers2 & NUM2_MYRIADS)
  group_len = 4;

 // is there a previous thousands part (as a previous "word") ?
 if ((n_digits == group_len) && (word[-2] ==
tr->langopts.thousands_sep) && IsDigit09(word[-3]))
  prev_thousands = 1;
 else if ((tr->langopts.thousands_sep == ' ') ||
(tr->langopts.numbers & NUM_ALLOW_SPACE)) {
  // thousands groups can be separated by spaces
  if ((n_digits == 3) && !(wtab->flags & FLAG_MULTIPLE_SPACES) &&
IsDigit09(word[-2]))
   prev_thousands = 1;
 }
 if (prev_thousands == 0)
  speak_missing_thousands = 0;
```

```
 ph_ordinal2[0] = 0;
 ph_zeros[0] = 0;

 if (prev_thousands || (word[0] != '0')) {
  // don't check for ordinal if the number has a leading zero
  ordinal = CheckDotOrdinal(tr, word, &word[ix], wtab, 0);
 }

 if ((word[ix] == '.') && !IsDigit09(word[ix+1]) &&
!IsDigit09(word[ix+2]) && !(wtab[1].flags & FLAG_NOSPACE)) {
  // remove dot unless followed by another number
  word[ix] = 0;
 }

 if ((ordinal == 0) || (tr->translator_name == L('h', 'u'))) {
  // NOTE lang=hu, allow both dot and ordinal suffix, eg.
"december 21.-én"
  // look for an ordinal number suffix after the number
  ix++;
  p = suffix;
  if (wtab[0].flags & FLAG_HYPHEN_AFTER) {
   *p++ = '-';
   ix++;
  }
  while ((word[ix] != 0) && (word[ix] != ' ') && (ix <
(int)(sizeof(suffix)-1)))
   *p++ = word[ix++];
  *p = 0;

  if (suffix[0] != 0) {
   if ((tr->langopts.ordinal_indicator != NULL) &&
(strcmp(suffix, tr->langopts.ordinal_indicator) == 0))
    ordinal = 2;
   else if (!IsDigit09(suffix[0])) { // not _#9 (tab)
    sprintf(string, "_#%s", suffix);
    if (Lookup(tr, string, ph_ordinal2)) {
     // this is an ordinal suffix
```

```
     ordinal = 2;
     flags[0] |= FLAG_SKIPWORDS;
     skipwords = 1;
     sprintf(string, "_x#%s", suffix);
     Lookup(tr, string, ph_ordinal2x); // is there an alternate
pronunciation?
    }
   }
  }
 }

 if (wtab[0].flags & FLAG_ORDINAL)
  ordinal = 2;

 ph_append[0] = 0;
 ph_buf2[0] = 0;

 if ((word[0] == '0') && (prev_thousands == 0) && (word[1] != '
') && (word[1] != tr->langopts.decimal_sep)) {
  if ((n_digits == 2) && (word[3] == ':') && IsDigit09(word[5])
&& isspace(word[7])) {
   // looks like a time 02:30, omit the leading zero
  } else {
   if (n_digits > 3) {
    flags[0] &= ~FLAG_SKIPWORDS;
    return 0; // long number string with leading zero, speak as
individual digits
   }

   // speak leading zeros
   for (ix = 0; (word[ix] == '0') && (ix < (n_digits-1)); ix++)
    Lookup(tr, "_0", &ph_zeros[strlen(ph_zeros)]);
  }
 }

 if ((tr->langopts.numbers & NUM_ALLOW_SPACE) && (word[n_digits]
== ' '))
```

```
  thousands_inc = 1;
 else if (word[n_digits] == tr->langopts.thousands_sep)
  thousands_inc = 2;

 suffix_ix = n_digits+2;
 if (thousands_inc > 0) {
  // if the following "words" are three-digit groups, count them
and add
  // a "thousand"/"million" suffix to this one
  digix = n_digits + thousands_inc;

  while (((wtab[thousandplex+1].flags & FLAG_MULTIPLE_SPACES) ==
0) && CheckThousandsGroup(&word[digix], group_len)) {
   for (ix = 0; ix < group_len; ix++) {
    if (word[digix+ix] != '0') {
     thousands_exact = 0;
     break;
    }
   }

   thousandplex++;
   digix += group_len;
   if ((word[digix] == tr->langopts.thousands_sep) ||
((tr->langopts.numbers & NUM_ALLOW_SPACE) && (word[digix] == '
'))) {
    suffix_ix = digix+2;
    digix += thousands_inc;
   } else
    break;
  }
 }

 if ((value == 0) && prev_thousands)
  suppress_null = true;

 if (tr->translator_name == L('h', 'u')) {
  // variant form of numbers when followed by hyphen and a suffix
```

```
starting with 'a' or 'e' (but not a, e, az, ez, azt, ezt
  if ((wtab[thousandplex].flags & FLAG_HYPHEN_AFTER) &&
(thousands_exact == 1) && hu_number_e(&word[suffix_ix],
thousandplex, value))
   number_control |= 1; // use _1e variant of number
 }

 if ((word[n_digits] == tr->langopts.decimal_sep) &&
IsDigit09(word[n_digits+1])) {
  // this "word" ends with a decimal point
  Lookup(tr, "_dpt", ph_append);
  decimal_point = 0x100;
 } else if (suppress_null == false) {
  if (thousands_inc > 0) {
   if (thousandplex > 0) {
    if ((suppress_null == false) && (LookupThousands(tr, value,
thousandplex, thousands_exact, ph_append))) {
     // found an exact match for N thousand
     value = 0;
     suppress_null = true;
    }
   }
  }
 } else if (speak_missing_thousands == 1) {
  // speak this thousandplex if there was no word for the
previous thousandplex
  sprintf(string, "_0M%d", thousandplex+1);
  if (Lookup(tr, string, buf1) == 0) {
   sprintf(string, "_0M%d", thousandplex);
   Lookup(tr, string, ph_append);
  }
 }

 if ((ph_append[0] == 0) && (word[n_digits] == '.') &&
(thousandplex == 0))
  Lookup(tr, "_.", ph_append);
```

```
 if (thousandplex == 0) {
  char *p2;
  // look for combinations of the number with the next word
  p = word;
  while (IsDigit09(p[1])) p++; // just use the last digit
  if (IsDigit09(p[-1])) {
   p2 = p - 1;
   if (LookupDictList(tr, &p2, buf_digit_lookup, flags,
FLAG_SUFX, wtab)) // lookup 2 digits
    n_digit_lookup = 2;
  }

  if ((buf_digit_lookup[0] == 0) && (*p != '0')) {
   // LANG=hu ?
   // not found, lookup only the last digit (?? but not if dot-
ordinal has been found)
   if (LookupDictList(tr, &p, buf_digit_lookup, flags, FLAG_SUFX,
wtab)) // don't match '0', or entries with $only
    n_digit_lookup = 1;
  }

  if (prev_thousands == 0) {
   if ((decimal_point == 0) && (ordinal == 0)) {
    // Look for special pronunciation for this number in
isolation (LANG=kl)
    sprintf(string, "_%dn", value);
    if (Lookup(tr, string, ph_out))
     return 1;
   }

   if (tr->langopts.numbers2 & NUM2_PERCENT_BEFORE) {
    // LANG=si, say "percent" before the number
    p2 = word;
    while ((*p2 != ' ') && (*p2 != 0))
     p2++;
    if (p2[1] == '%') {
     Lookup(tr, "%", ph_out);
```

```
    ph_out += strlen(ph_out);
    p2[1] = ' ';
   }
  }
 }

}

LookupNum3(tr, value, ph_buf, suppress_null, thousandplex,
prev_thousands | ordinal | decimal_point);
if ((thousandplex > 0) && (tr->langopts.numbers2 & 0x200))
 sprintf(ph_out, "%s%s%c%s%s", ph_zeros, ph_append,
phonEND_WORD, ph_buf2, ph_buf); // say "thousands" before its
number
else
 sprintf(ph_out, "%s%s%s%c%s", ph_zeros, ph_buf2, ph_buf,
phonEND_WORD, ph_append);

while (decimal_point) {
 n_digits++;

 decimal_count = 0;
 while (IsDigit09(word[n_digits+decimal_count]))
  decimal_count++;

 max_decimal_count = 2;
 switch (decimal_mode = (tr->langopts.numbers & 0xe000))
 {
 case NUM_DFRACTION_4:
  max_decimal_count = 5;
  // fallthrough:
 case NUM_DFRACTION_2:
  // French/Polish decimal fraction
  while (word[n_digits] == '0') {
   Lookup(tr, "_0", buf1);
   strcat(ph_out, buf1);
   decimal_count--;
```

```
    n_digits++;
   }
   if ((decimal_count <= max_decimal_count) &&
IsDigit09(word[n_digits])) {
    LookupNum3(tr, atoi(&word[n_digits]), buf1, false, 0, 0);
    strcat(ph_out, buf1);
    n_digits += decimal_count;
   }
   break;
  case NUM_DFRACTION_1: // italian, say "hundredths" if leading
zero
  case NUM_DFRACTION_5: // hungarian, always say "tenths" etc.
  case NUM_DFRACTION_6: // kazakh, always say "tenths" etc,
before the decimal fraction
   LookupNum3(tr, atoi(&word[n_digits]), ph_buf, false, 0, 0);
   if ((word[n_digits] == '0') || (decimal_mode !=
NUM_DFRACTION_1)) {
    // decimal part has leading zeros, so add a "hundredths" or
"thousandths" suffix
    sprintf(string, "_0Z%d", decimal_count);
    if (Lookup(tr, string, buf1) == 0)
     break; // revert to speaking single digits

    if (decimal_mode == NUM_DFRACTION_6)
     strcat(ph_out, buf1);
    else
     strcat(ph_buf, buf1);
   }
   strcat(ph_out, ph_buf);
   n_digits += decimal_count;
   break;
  case NUM_DFRACTION_3:
   // Romanian decimal fractions
   if ((decimal_count <= 4) && (word[n_digits] != '0')) {
    LookupNum3(tr, atoi(&word[n_digits]), buf1, false, 0, 0);
    strcat(ph_out, buf1);
    n_digits += decimal_count;
```

```
   }
   break;
  case NUM_DFRACTION_7:
   // alternative form of decimal fraction digits, except the
final digit
   while (decimal_count-- > 1) {
    sprintf(string, "_%cd", word[n_digits]);
    if (Lookup(tr, string, buf1) == 0)
     break;
    n_digits++;
    strcat(ph_out, buf1);
   }
  }

  while (IsDigit09(c = word[n_digits]) && (strlen(ph_out) <
(N_WORD_PHONEMES - 10))) {
   // speak any remaining decimal fraction digits individually
   value = word[n_digits++] - '0';
   LookupNum2(tr, value, 0, 2, buf1);
   len = strlen(ph_out);
   sprintf(&ph_out[len], "%c%s", phonEND_WORD, buf1);
  }

  // something after the decimal part ?
  if (Lookup(tr, "_dpt2", buf1))
   strcat(ph_out, buf1);

  if ((c == tr->langopts.decimal_sep) &&
IsDigit09(word[n_digits+1])) {
   Lookup(tr, "_dpt", buf1);
   strcat(ph_out, buf1);
  } else
   decimal_point = 0;
 }
 if ((ph_out[0] != 0) && (ph_out[0] != phonSWITCH)) {
  int next_char;
  char *p;
```

```
  p = &word[n_digits+1];

  p += utf8_in(&next_char, p);
  if ((tr->langopts.numbers & NUM_NOPAUSE) && (next_char == ' '))
   utf8_in(&next_char, p);

  if (!iswalpha(next_char) && (thousands_exact == 0))
   strcat(ph_out, str_pause); // don't add pause for 100s,  6th,
etc.
 }

 speak_missing_thousands--;

 if (skipwords)
  dictionary_skipwords = skipwords;
 return 1;
}

int TranslateNumber(Translator *tr, char *word1, char *ph_out,
unsigned int *flags, WORD_TAB *wtab, int control)
{
 if ((option_sayas == SAYAS_DIGITS1) || (wtab[0].flags &
FLAG_INDIVIDUAL_DIGITS))
  return 0; // speak digits individually

 if (tr->langopts.numbers != 0)
  return TranslateNumber_1(tr, word1, ph_out, flags, wtab,
control);
 return 0;
}
```

Chapter 68

./tests/api.c

```
#include "config.h"

#include <assert.h>
#include <stdlib.h>
#include <string.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/speak_lib.h>
#include <espeak-ng/encoding.h>

#include "speech.h"
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

// region espeak_Initialize

static void
test_espeak_terminate_without_initialize()
{
 printf("testing espeak_Terminate without espeak_Initialize\n");
```

```
 assert(event_list == NULL);
 assert(translator == NULL);
 assert(p_decoder == NULL);

 assert(espeak_Terminate() == EE_OK);
 assert(event_list == NULL);
 assert(translator == NULL);
 assert(p_decoder == NULL);
}

// similar lines removed from listning...
// endregion

int
main(int argc, char **argv)
{
 (void)argc; // unused parameter

 char *progdir = strdup(argv[0]);
 char *dir = strrchr(progdir, '/');
 if (dir != NULL) *dir = 0;

 test_espeak_terminate_without_initialize();
 test_espeak_initialize();

 test_espeak_synth();
 test_espeak_synth(); // Check that this does not crash when run
a second time.
 test_espeak_synth_no_voices(progdir);
 test_espeak_synth();

 test_espeak_ng_synthesize();
 test_espeak_ng_synthesize(); // Check that this does not crash
when run a second time.
 test_espeak_ng_synthesize_no_voices(progdir);
 test_espeak_ng_synthesize();
```

```
 test_espeak_set_voice_by_name_null_voice();
 test_espeak_set_voice_by_name_blank_voice();
 test_espeak_set_voice_by_name_valid_voice();
 test_espeak_set_voice_by_name_invalid_voice();
 test_espeak_set_voice_by_name_language_variant_intonation_parame
ter();

 test_espeak_set_voice_by_properties_empty();
 test_espeak_set_voice_by_properties_blank_language();
 test_espeak_set_voice_by_properties_with_valid_language();
 test_espeak_set_voice_by_properties_with_invalid_language();

 free(progdir);

 return EXIT_SUCCESS;
}
```

Chapter 69

./tests/fuzzrunner.c

```
#include "config.h"
#include "speech.h"

#include <errno.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#include <espeak-ng/espeak_ng.h>

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size);

int main(int argc, char **argv) {
 int i;

 for (i = 1; i < argc; i++) {
  size_t filesize = GetFileLength(argv[i]);
  FILE *stream = fopen(argv[i], "r");
  unsigned char *text = NULL;
  if (stream == NULL) {
   perror(argv[i]);
   exit(EXIT_FAILURE);
```

```
        }

        text = (unsigned char *) malloc(filesize + 1);
        if (text == NULL) {
         espeak_ng_PrintStatusCodeMessage(ENOMEM, stderr, NULL);
         exit(EXIT_FAILURE);
        }

        fread(text, 1, filesize, stream);
        text[filesize] = 0;
        fclose(stream);

        LLVMFuzzerTestOneInput(text, filesize);
        free(text);
       }

      return EXIT_SUCCESS;
     }
```

Chapter 70

./tests/readclause.c

```
#include "config.h"

#include <assert.h>
#include <errno.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/stat.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/encoding.h>

#include "readclause.h"
#include "speech.h"
#include "phoneme.h"
#include "voice.h"
#include "synthesize.h"
#include "translate.h"

// Arguments to ReadClause. Declared here to avoid duplicating
them across the
```

```
// different test functions.
static char source[N_TR_SOURCE+40]; // extra space for embedded
command & voice change info at end
static short charix[N_TR_SOURCE+4];
static int charix_top = 0;
static int tone2;
static char voice_change_name[40];

static espeak_ng_STATUS
set_text(const char *text, const char *voicename)
{
 espeak_ng_STATUS status = espeak_ng_SetVoiceByName(voicename);
 if (status != ENS_OK)
  return status;

 if (p_decoder == NULL)
  p_decoder = create_text_decoder();

 count_characters = 0;
 return text_decoder_decode_string(p_decoder, text, -1,
ESPEAKNG_ENCODING_UTF_8);
}

static void
test_latin()
{
 printf("testing Latin (Latn)\n");

 assert(clause_type_from_codepoint('?') == CLAUSE_QUESTION);
 assert(clause_type_from_codepoint('!') == CLAUSE_EXCLAMATION);
 assert(clause_type_from_codepoint(',') == CLAUSE_COMMA);
 assert(clause_type_from_codepoint(':') == CLAUSE_COLON);
 assert(clause_type_from_codepoint(';') == CLAUSE_SEMICOLON);

 assert(clause_type_from_codepoint(0x00A1) == (CLAUSE_SEMICOLON |
CLAUSE_OPTIONAL_SPACE_AFTER));
 assert(clause_type_from_codepoint(0x00Bf) == (CLAUSE_SEMICOLON |
```

```
CLAUSE_OPTIONAL_SPACE_AFTER));

 assert(clause_type_from_codepoint(0x2013) == CLAUSE_SEMICOLON);
 assert(clause_type_from_codepoint(0x2014) == CLAUSE_SEMICOLON);
 assert(clause_type_from_codepoint(0x2026) == (CLAUSE_SEMICOLON |
CLAUSE_SPEAK_PUNCTUATION_NAME | CLAUSE_OPTIONAL_SPACE_AFTER));
}

static void
test_latin_sentence()
{
 printf("testing Latin (Latn) ... sentence\n");

 assert(clause_type_from_codepoint('a') == CLAUSE_NONE);
 assert(clause_type_from_codepoint('.') == CLAUSE_PERIOD);

 short retix[] = {
  0, 2, 3, 4, 5, 6, // Jane
  0, 8, 9, 10, 11, 12, 13, 14, 15, // finished
  0, 17, 18, // #1
  0, 20, 21, // in
  0, 23, 24, 25, // the
  0, 27, 28, 29, 30 }; // race

 assert(set_text("Janet finished #1 in the race.", "en") ==
ENS_OK);

 charix_top = 0;
 assert(ReadClause(translator, source, charix, &charix_top,
N_TR_SOURCE, &tone2, voice_change_name) == (CLAUSE_PERIOD |
CLAUSE_DOT_AFTER_LAST_WORD));
 assert(!strcmp(source, "Janet finished #1 in the race "));
 assert(charix_top == (sizeof(retix)/sizeof(retix[0])) - 1);
 assert(!memcmp(charix, retix, sizeof(retix)));
 assert(tone2 == 0);
 assert(voice_change_name[0] == 0);
```

```
 charix_top = 0;
 assert(ReadClause(translator, source, charix, &charix_top,
N_TR_SOURCE, &tone2, voice_change_name) == CLAUSE_EOF);
 assert(!strcmp(source, " "));
 assert(charix_top == 0);
}

static void
test_greek()
{
 printf("testing Greek (Grek)\n");

 assert(clause_type_from_codepoint(0x037E) == CLAUSE_QUESTION);
 assert(clause_type_from_codepoint(0x0387) == CLAUSE_SEMICOLON);
}

static void
test_armenian()
{
 printf("testing Armenian (Armn)\n");

 assert(clause_type_from_codepoint(0x055B) == (CLAUSE_EXCLAMATION
| CLAUSE_PUNCTUATION_IN_WORD));
 assert(clause_type_from_codepoint(0x055C) == (CLAUSE_EXCLAMATION
| CLAUSE_PUNCTUATION_IN_WORD));
 assert(clause_type_from_codepoint(0x055D) == CLAUSE_COMMA);
 assert(clause_type_from_codepoint(0x055E) == (CLAUSE_QUESTION |
CLAUSE_PUNCTUATION_IN_WORD));
 assert(clause_type_from_codepoint(0x0589) == (CLAUSE_PERIOD |
CLAUSE_OPTIONAL_SPACE_AFTER));
}

static void
test_arabic()
{
 printf("testing Arabic (Arab)\n");
```

```
 assert(clause_type_from_codepoint(0x060C) == CLAUSE_COMMA);
 assert(clause_type_from_codepoint(0x061B) == CLAUSE_SEMICOLON);
 assert(clause_type_from_codepoint(0x061F) == CLAUSE_QUESTION);
 assert(clause_type_from_codepoint(0x06D4) == CLAUSE_PERIOD);
}

static void
test_devanagari()
{
 printf("testing Devanagari (Deva)\n");

 assert(clause_type_from_codepoint(0x0964) == (CLAUSE_PERIOD |
CLAUSE_OPTIONAL_SPACE_AFTER));
}

static void
test_tibetan()
{
 printf("testing Tibetan (Tibt)\n");

 assert(clause_type_from_codepoint(0x0F0D) == (CLAUSE_PERIOD |
CLAUSE_OPTIONAL_SPACE_AFTER));
 assert(clause_type_from_codepoint(0x0F0E) == CLAUSE_PARAGRAPH);
}

static void
test_sinhala()
{
 printf("testing Sinhala (Sinh)\n");

 assert(clause_type_from_codepoint(0x0DF4) == (CLAUSE_PERIOD |
CLAUSE_OPTIONAL_SPACE_AFTER));
}

static void
test_georgian()
{
```

```
 printf("testing Georgian (Geor)\n");

 assert(clause_type_from_codepoint(0x10FB) == CLAUSE_PARAGRAPH);
}

static void
test_ethiopic()
{
 printf("testing Ethiopic (Ethi)\n");

 assert(clause_type_from_codepoint(0x1362) == CLAUSE_PERIOD);
 assert(clause_type_from_codepoint(0x1363) == CLAUSE_COMMA);
 assert(clause_type_from_codepoint(0x1364) == CLAUSE_SEMICOLON);
 assert(clause_type_from_codepoint(0x1365) == CLAUSE_COLON);
 assert(clause_type_from_codepoint(0x1366) == CLAUSE_COLON);
 assert(clause_type_from_codepoint(0x1367) == CLAUSE_QUESTION);
 assert(clause_type_from_codepoint(0x1368) == CLAUSE_PARAGRAPH);
}

static void
test_ideographic()
{
 printf("testing Ideographic (Hani)\n");

 assert(clause_type_from_codepoint(0x3001) == (CLAUSE_COMMA |
CLAUSE_OPTIONAL_SPACE_AFTER));
 assert(clause_type_from_codepoint(0x3002) == (CLAUSE_PERIOD |
CLAUSE_OPTIONAL_SPACE_AFTER));
}

static void
test_fullwidth()
{
 printf("testing Full Width\n");

 assert(clause_type_from_codepoint(0xFF01) == (CLAUSE_EXCLAMATION
| CLAUSE_OPTIONAL_SPACE_AFTER));
```

```
 assert(clause_type_from_codepoint(0xFF0C) == (CLAUSE_COMMA |
CLAUSE_OPTIONAL_SPACE_AFTER));
 assert(clause_type_from_codepoint(0xFF0E) == (CLAUSE_PERIOD |
CLAUSE_OPTIONAL_SPACE_AFTER));
 assert(clause_type_from_codepoint(0xFF1A) == (CLAUSE_COLON |
CLAUSE_OPTIONAL_SPACE_AFTER));
 assert(clause_type_from_codepoint(0xFF1B) == (CLAUSE_SEMICOLON |
CLAUSE_OPTIONAL_SPACE_AFTER));
 assert(clause_type_from_codepoint(0xFF1F) == (CLAUSE_QUESTION |
CLAUSE_OPTIONAL_SPACE_AFTER));
}

static void
test_uts51_emoji_character()
{
 printf("testing Emoji ... UTS-51 ED-3. emoji character\n");

 short retix[] = {
  0, -1, -1,
  2, -1, -1,
  3, -1, -1,
  4, -1, -1, -1,
  5, -1, -1, -1,
  6 };

 assert(set_text(
  "\xE2\x86\x94"      // [2194]  left right arrow
  "\xE2\x86\x95"      // [2195]  up down arrow
  "\xE2\x9B\x94"      // [26D5]  no entry
  "\xF0\x9F\x90\x8B"  // [1F40B] whale
  "\xF0\x9F\x90\xAC", // [1F42C] dolphin
  "en") == ENS_OK);

 charix_top = 0;
 assert(ReadClause(translator, source, charix, &charix_top,
N_TR_SOURCE, &tone2, voice_change_name) == CLAUSE_EOF);
 assert(!strcmp(source,
```

```
  "\xE2\x86\x94"     // [2194]  left right arrow
  "\xE2\x86\x95"     // [2195]  up down arrow
  "\xE2\x9B\x94"     // [26D5]  no entry
  "\xF0\x9F\x90\x8B" // [1F40B] whale
  "\xF0\x9F\x90\xAC" // [1F42C] dolphin
  " "));
 assert(charix_top == (sizeof(retix)/sizeof(retix[0])) - 1);
 assert(!memcmp(charix, retix, sizeof(retix)));
 assert(tone2 == 0);
 assert(voice_change_name[0] == 0);
}

static void
test_uts51_text_presentation_sequence()
{
 printf("testing Emoji ... UTS-51 ED-8a. text presentation
sequence\n");

 short retix[] = {
  0, 2, -1, -1,
  3, 4, -1, -1,
  5, -1, -1, 6, -1, -1,
  7, -1, -1, -1, 8, -1, -1,
  9 };

 assert(set_text(
  "#\xEF\xB8\x8E"                 // [0023 FE0E]  number sign
(text style)
  "4\xEF\xB8\x8E"                 // [0034 FE0E]  digit four
(text style)
  "\xE2\x80\xBC\xEF\xB8\x8E"      // [203C FE0E]  double
exclamation mark (text style)
  "\xF0\x9F\x97\x92\xEF\xB8\x8E", // [1F5D2 FE0E] spiral note pad
(text style)
  "en") == ENS_OK);

 charix_top = 0;
```

```
 assert(ReadClause(translator, source, charix, &charix_top,
N_TR_SOURCE, &tone2, voice_change_name) == CLAUSE_EOF);
 assert(!strcmp(source,
  "#\xEF\xB8\x8E"                // [0023 FE0E]  number sign
(text style)
  "4\xEF\xB8\x8E"                // [0034 FE0E]  digit four (text
style)
  "\xE2\x80\xBC\xEF\xB8\x8E"     // [203C FE0E]  double
exclamation mark (text style)
  "\xF0\x9F\x97\x92\xEF\xB8\x8E" // [1F5D2 FE0E] spiral note pad
(text style)
  " "));
 assert(charix_top == (sizeof(retix)/sizeof(retix[0])) - 1);
 assert(!memcmp(charix, retix, sizeof(retix)));
 assert(tone2 == 0);
 assert(voice_change_name[0] == 0);
}

static void
test_uts51_emoji_presentation_sequence()
{
 printf("testing Emoji ... UTS-51 ED-9a. emoji presentation
sequence\n");

 short retix[] = {
  0, 2, -1, -1,
  3, 4, -1, -1,
  5, -1, -1, 6, -1, -1,
  7, -1, -1, -1, 8, -1, -1,
  9 };

 assert(set_text(
  "#\xEF\xB8\x8F"                 // [0023 FE0F]  number sign
(emoji style)
  "4\xEF\xB8\x8F"                 // [0034 FE0F]  digit four
(emoji style)
  "\xE2\x80\xBC\xEF\xB8\x8F"      // [203C FE0F]  double
```

```
exclamation mark (emoji style)
  "\xF0\x9F\x97\x92\xEF\xB8\x8F", // [1F5D2 FE0F] spiral note pad
(emoji style)
  "en") == ENS_OK);

 charix_top = 0;
 assert(ReadClause(translator, source, charix, &charix_top,
N_TR_SOURCE, &tone2, voice_change_name) == CLAUSE_EOF);
 assert(!strcmp(source,
  "#\xEF\xB8\x8F"                // [0023 FE0F]  number sign
(emoji style)
  "4\xEF\xB8\x8F"                // [0034 FE0F]  digit four
(emoji style)
  "\xE2\x80\xBC\xEF\xB8\x8F"     // [203C FE0F]  double
exclamation mark (emoji style)
  "\xF0\x9F\x97\x92\xEF\xB8\x8F" // [1F5D2 FE0F] spiral note pad
(emoji style)
  " "));
 assert(charix_top == (sizeof(retix)/sizeof(retix[0])) - 1);
 assert(!memcmp(charix, retix, sizeof(retix)));
 assert(tone2 == 0);
 assert(voice_change_name[0] == 0);
}

static void
test_uts51_emoji_modifier_sequence()
{
 printf("testing Emoji ... UTS-51 ED-13. emoji modifier
sequence\n");

 short retix[] = {
  0, -1, -1, 2, -1, -1, -1,
  3, -1, -1, -1, 4, -1, -1, -1,
  5, -1, -1, -1, 6, -1, -1, -1,
  7 };

 assert(set_text(
```

```
  "\xE2\x98\x9D\xF0\x9F\x8F\xBB"      // [261D 1F3FB]  index
pointing up; light skin tone
  "\xF0\x9F\x91\xB0\xF0\x9F\x8F\xBD"  // [1F5D2 1F3FD] bride with
veil; medium skin tone
  "\xF0\x9F\x92\xAA\xF0\x9F\x8F\xBF", // [1F4AA 1F3FF] flexed
biceps; dark skin tone
  "en") == ENS_OK);

 charix_top = 0;
 assert(ReadClause(translator, source, charix, &charix_top,
N_TR_SOURCE, &tone2, voice_change_name) == CLAUSE_EOF);
 assert(!strcmp(source,
  "\xE2\x98\x9D\xF0\x9F\x8F\xBB"     // [261D 1F3FB]  index
pointing up; light skin tone
  "\xF0\x9F\x91\xB0\xF0\x9F\x8F\xBD" // [1F5D2 1F3FD] bride with
veil; medium skin tone
  "\xF0\x9F\x92\xAA\xF0\x9F\x8F\xBF" // [1F4AA 1F3FF] flexed
biceps; dark skin tone
  " "));
 assert(charix_top == (sizeof(retix)/sizeof(retix[0])) - 1);
 assert(!memcmp(charix, retix, sizeof(retix)));
 assert(tone2 == 0);
 assert(voice_change_name[0] == 0);
}

static void
test_uts51_emoji_flag_sequence()
{
 printf("testing Emoji ... UTS-51 ED-14. emoji flag sequence\n");

 short retix[] = {
  0, -1, -1, -1, 2, -1, -1, -1,
  3, -1, -1, -1, 4, -1, -1, -1,
  5, -1, -1, -1, 6, -1, -1, -1,
  7, -1, -1, -1, 8, -1, -1, -1,
  9 };
```

```
 assert(set_text(
  "\xF0\x9F\x87\xA6\xF0\x9F\x87\xB7"  // [1F1E6 1F1F7] AR
(argentina)
  "\xF0\x9F\x87\xA7\xF0\x9F\x87\xAC"  // [1F1E7 1F1EC] BG
(bulgaria)
  "\xF0\x9F\x87\xAC\xF0\x9F\x87\xA8"  // [1F1EC 1F1E8] GC --
unknown country flag
  "\xF0\x9F\x87\xAC\xF0\x9F\x87\xB1", // [1F1EC 1F1F1] GL
(greenland)
  "en") == ENS_OK);

 charix_top = 0;
 assert(ReadClause(translator, source, charix, &charix_top,
N_TR_SOURCE, &tone2, voice_change_name) == CLAUSE_EOF);
 assert(!strcmp(source,
  "\xF0\x9F\x87\xA6\xF0\x9F\x87\xB7" // [1F1E6 1F1F7] AR
(argentina)
  "\xF0\x9F\x87\xA7\xF0\x9F\x87\xAC" // [1F1E7 1F1EC] BG
(bulgaria)
  "\xF0\x9F\x87\xAC\xF0\x9F\x87\xA8" // [1F1EC 1F1E8] GC --
unknown country flag
  "\xF0\x9F\x87\xAC\xF0\x9F\x87\xB1" // [1F1EC 1F1F1] GL
(greenland)
  " "));
 assert(charix_top == (sizeof(retix)/sizeof(retix[0])) - 1);
 assert(!memcmp(charix, retix, sizeof(retix)));
 assert(tone2 == 0);
 assert(voice_change_name[0] == 0);
}

static void
test_uts51_emoji_tag_sequence_emoji_character()
{
 printf("testing Emoji ... UTS-51 ED-14a. emoji tag sequence
(emoji character)\n");

 short retix[] = {
```

```
  0, -1, -1, -1, // emoji character
  2, -1, -1, -1, 3, -1, -1, -1, 4, -1, -1, -1, 5, -1, -1, -1, 6,
-1, -1, -1, // tag spec
  7, -1, -1, -1, // tag term
  8, -1, -1, -1, // emoji character
  9, -1, -1, -1, 10, -1, -1, -1, 11, -1, -1, -1, 12, -1, -1, -1,
13, -1, -1, -1, // tag spec
  14, -1, -1, -1, // tag term
  15, -1, -1, -1, // emoji character
  16, -1, -1, -1, 17, -1, -1, -1, 18, -1, -1, -1, 19, -1, -1, -1,
// tag spec
  20, -1, -1, -1, // tag term
  21 };

 assert(set_text(
  // tag_base = emoji_character (RGI sequence)
  "\xF0\x9F\x8F\xB4"  // [1F3F4] flag
  "\xF3\xA0\x81\xA7"  // [E0067] tag : g
  "\xF3\xA0\x81\xA2"  // [E0062] tag : b
  "\xF3\xA0\x81\xA5"  // [E0065] tag : e
  "\xF3\xA0\x81\xAE"  // [E006E] tag : n
  "\xF3\xA0\x81\xA7"  // [E006E] tag : g
  "\xF3\xA0\x81\xBF"  // [E007F] tag : (cancel)
  // tag_base = emoji_character (RGI sequence)
  "\xF0\x9F\x8F\xB4"  // [1F3F4] flag
  "\xF3\xA0\x81\xA7"  // [E0067] tag : g
  "\xF3\xA0\x81\xA2"  // [E0062] tag : b
  "\xF3\xA0\x81\xB3"  // [E0065] tag : s
  "\xF3\xA0\x81\xA3"  // [E006E] tag : c
  "\xF3\xA0\x81\xB4"  // [E006E] tag : t
  "\xF3\xA0\x81\xBF"  // [E007F] tag : (cancel)
  // tag_base = emoji_character (non-RGI sequence)
  "\xF0\x9F\x8F\xB4"  // [1F3F4] flag
  "\xF3\xA0\x81\xB5"  // [E0067] tag : u
  "\xF3\xA0\x81\xB3"  // [E0062] tag : s
  "\xF3\xA0\x81\xA3"  // [E0065] tag : c
  "\xF3\xA0\x81\xA1"  // [E006E] tag : a
```

```
  "\xF3\xA0\x81\xBF", // [E007F] tag : (cancel)
  "en") == ENS_OK);

 charix_top = 0;
 assert(ReadClause(translator, source, charix, &charix_top,
N_TR_SOURCE, &tone2, voice_change_name) == CLAUSE_EOF);
 assert(!strcmp(source,
  // tag_base = emoji_character (RGI sequence)
  "\xF0\x9F\x8F\xB4" // [1F3F4] flag
  "\xF3\xA0\x81\xA7" // [E0067] tag : g
  "\xF3\xA0\x81\xA2" // [E0062] tag : b
  "\xF3\xA0\x81\xA5" // [E0065] tag : e
  "\xF3\xA0\x81\xAE" // [E006E] tag : n
  "\xF3\xA0\x81\xA7" // [E006E] tag : g
  "\xF3\xA0\x81\xBF" // [E007F] tag : (cancel)
  // tag_base = emoji_character (RGI sequence)
  "\xF0\x9F\x8F\xB4" // [1F3F4] flag
  "\xF3\xA0\x81\xA7" // [E0067] tag : g
  "\xF3\xA0\x81\xA2" // [E0062] tag : b
  "\xF3\xA0\x81\xB3" // [E0065] tag : s
  "\xF3\xA0\x81\xA3" // [E006E] tag : c
  "\xF3\xA0\x81\xB4" // [E006E] tag : t
  "\xF3\xA0\x81\xBF" // [E007F] tag : (cancel)
  // tag_base = emoji_character (non-RGI sequence)
  "\xF0\x9F\x8F\xB4" // [1F3F4] flag
  "\xF3\xA0\x81\xB5" // [E0067] tag : u
  "\xF3\xA0\x81\xB3" // [E0062] tag : s
  "\xF3\xA0\x81\xA3" // [E0065] tag : c
  "\xF3\xA0\x81\xA1" // [E006E] tag : a
  "\xF3\xA0\x81\xBF" // [E007F] tag : (cancel)
  " "));
 assert(charix_top == (sizeof(retix)/sizeof(retix[0])) - 1);
 assert(!memcmp(charix, retix, sizeof(retix)));
 assert(tone2 == 0);
 assert(voice_change_name[0] == 0);
}
```

```
static void
test_uts51_emoji_combining_sequence()
{
 printf("testing Emoji ... UTS-51 ED-14b. emoji combining
sequence\n");

 short retix[] = {
  0, -1, -1, 2, -1, -1,              // emoji character
  3, -1, -1, 4, -1, -1, 5, -1, -1, // text presentation sequence
  6, -1, -1, 7, -1, -1, 8, -1, -1, // emoji presentation sequence
  9 };

 assert(set_text(
  "\xE2\x86\x95\xE2\x83\x9E"              // [2195 20DE]      up
down arrow; Me (enclosing square)
  "\xE2\x86\x95\xEF\xB8\x8E\xE2\x83\x9E"  // [2195 FE0E 20DE] up
down arrow; Me (enclosing square)
  "\xE2\x86\x95\xEF\xB8\x8F\xE2\x83\x9E", // [2195 FE0F 20DE] up
down arrow; Me (enclosing square)
  "en") == ENS_OK);

 charix_top = 0;
 assert(ReadClause(translator, source, charix, &charix_top,
N_TR_SOURCE, &tone2, voice_change_name) == CLAUSE_EOF);
 assert(!strcmp(source,
  "\xE2\x86\x95\xE2\x83\x9E"             // [2195 20DE]      up
down arrow; Me (enclosing square)
  "\xE2\x86\x95\xEF\xB8\x8E\xE2\x83\x9E" // [2195 FE0E 20DE] up
down arrow; Me (enclosing square)
  "\xE2\x86\x95\xEF\xB8\x8F\xE2\x83\x9E" // [2195 FE0F 20DE] up
down arrow; Me (enclosing square)
  " "));
 assert(charix_top == (sizeof(retix)/sizeof(retix[0])) - 1);
 assert(!memcmp(charix, retix, sizeof(retix)));
 assert(tone2 == 0);
 assert(voice_change_name[0] == 0);
}
```

```
static void
test_uts51_emoji_keycap_sequence()
{
 printf("testing Emoji ... UTS-51 ED-14c. emoji keycap
sequence\n");

 short retix[] = {
  0, 2, -1, -1, 3, -1, -1,
  4, 5, -1, -1, 6, -1, -1,
  7, 8, -1, -1, 9, -1, -1,
  10 };

 assert(set_text(
  "5\xEF\xB8\x8E\xE2\x83\xA3"  // [0035 FE0E 20E3] keycap 5
  "#\xEF\xB8\x8E\xE2\x83\xA3"  // [0023 FE0E 20E3] keycap #
  "*\xEF\xB8\x8E\xE2\x83\xA3", // [002A FE0E 20E3] keycap *
  "en") == ENS_OK);

 charix_top = 0;
 assert(ReadClause(translator, source, charix, &charix_top,
N_TR_SOURCE, &tone2, voice_change_name) == CLAUSE_EOF);
 assert(!strcmp(source,
  "5\xEF\xB8\x8E\xE2\x83\xA3" // [0035 FE0E 20E3] keycap 5
  "#\xEF\xB8\x8E\xE2\x83\xA3" // [0023 FE0E 20E3] keycap #
  "*\xEF\xB8\x8E\xE2\x83\xA3" // [002A FE0E 20E3] keycap *
  " "));
 assert(charix_top == (sizeof(retix)/sizeof(retix[0])) - 1);
 assert(!memcmp(charix, retix, sizeof(retix)));
 assert(tone2 == 0);
 assert(voice_change_name[0] == 0);
}

int
main(int argc, char **argv)
{
 (void)argc; // unused parameter
```

```
 (void)argv; // unused parameter

 assert(espeak_Initialize(AUDIO_OUTPUT_SYNCHRONOUS, 0, NULL,
espeakINITIALIZE_DONT_EXIT) == 22050);

 test_latin();
 test_latin_sentence();

 test_greek();
 test_armenian();
 test_arabic();
 test_devanagari();
 test_tibetan();
 test_sinhala();
 test_georgian();
 test_ethiopic();
 test_ideographic();
 test_fullwidth();

 test_uts51_emoji_character();
 test_uts51_text_presentation_sequence();
 test_uts51_emoji_presentation_sequence();
 test_uts51_emoji_modifier_sequence();
 test_uts51_emoji_flag_sequence();
 test_uts51_emoji_tag_sequence_emoji_character();
 test_uts51_emoji_combining_sequence();
 test_uts51_emoji_keycap_sequence();

 assert(espeak_Terminate() == EE_OK);

 return EXIT_SUCCESS;
}

// References:
//    [UTS-51]     Unicode Emoji
(http://www.unicode.org/reports/tr51/tr51-12.html) 5.0-12.
2017-05-18
```

Chapter 71

./tests/encoding.c

```
#include "config.h"

#include <assert.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

#include <espeak-ng/espeak_ng.h>
#include <espeak-ng/encoding.h>

static void
test_unbound_text_decoder()
{
 printf("testing unbound text decoder\n");

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(decoder != NULL);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}
```

```
static void
test_unknown_encoding()
{
 printf("testing unknown encodings\n");

 assert(espeak_ng_EncodingFromName(NULL) ==
ESPEAKNG_ENCODING_UNKNOWN);
 assert(espeak_ng_EncodingFromName("") ==
ESPEAKNG_ENCODING_UNKNOWN);
 assert(espeak_ng_EncodingFromName("abcxyz") ==
ESPEAKNG_ENCODING_UNKNOWN);
 assert(espeak_ng_EncodingFromName("US") ==
ESPEAKNG_ENCODING_UNKNOWN); // wrong case

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_UNKNOWN) == ENS_UNKNOWN_TEXT_ENCODING);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_us_ascii_encoding()
{
 printf("testing US-ASCII encoding\n");

 assert(espeak_ng_EncodingFromName("US-ASCII") ==
ESPEAKNG_ENCODING_US_ASCII);
 assert(espeak_ng_EncodingFromName("iso-ir-6") ==
ESPEAKNG_ENCODING_US_ASCII);
 assert(espeak_ng_EncodingFromName("ANSI_X3.4-1968") ==
ESPEAKNG_ENCODING_US_ASCII);
 assert(espeak_ng_EncodingFromName("ANSI_X3.4-1986") ==
ESPEAKNG_ENCODING_US_ASCII);
```

```
 assert(espeak_ng_EncodingFromName("ISO_646.irv:1991") ==
ESPEAKNG_ENCODING_US_ASCII);
 assert(espeak_ng_EncodingFromName("ISO646-US") ==
ESPEAKNG_ENCODING_US_ASCII);
 assert(espeak_ng_EncodingFromName("us") ==
ESPEAKNG_ENCODING_US_ASCII);
 assert(espeak_ng_EncodingFromName("IBM367") ==
ESPEAKNG_ENCODING_US_ASCII);
 assert(espeak_ng_EncodingFromName("cp367") ==
ESPEAKNG_ENCODING_US_ASCII);
 assert(espeak_ng_EncodingFromName("csASCII") ==
ESPEAKNG_ENCODING_US_ASCII);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_US_ASCII) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_koi8_r_encoding()
{
 printf("testing KOI8-R encoding\n");
```

```
 assert(espeak_ng_EncodingFromName("KOI8-R") ==
ESPEAKNG_ENCODING_KOI8_R);
 assert(espeak_ng_EncodingFromName("csKOI8R") ==
ESPEAKNG_ENCODING_KOI8_R);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_KOI8_R) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x021a);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iscii_encoding()
{
 printf("testing ISCII encoding\n");

 assert(espeak_ng_EncodingFromName("ISCII") ==
ESPEAKNG_ENCODING_ISCII);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xE6", 5,
ESPEAKNG_ENCODING_ISCII) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
```

```
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xfffd);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xfffd);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x094c);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_8859_1_encoding()
{
 printf("testing ISO-8859-1 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-1") ==
ESPEAKNG_ENCODING_ISO_8859_1);
 assert(espeak_ng_EncodingFromName("ISO_8859-1") ==
ESPEAKNG_ENCODING_ISO_8859_1);
 assert(espeak_ng_EncodingFromName("ISO_8859-1:1987") ==
ESPEAKNG_ENCODING_ISO_8859_1);
 assert(espeak_ng_EncodingFromName("iso-ir-100") ==
ESPEAKNG_ENCODING_ISO_8859_1);
 assert(espeak_ng_EncodingFromName("latin1") ==
ESPEAKNG_ENCODING_ISO_8859_1);
 assert(espeak_ng_EncodingFromName("l1") ==
ESPEAKNG_ENCODING_ISO_8859_1);
 assert(espeak_ng_EncodingFromName("IBM819") ==
ESPEAKNG_ENCODING_ISO_8859_1);
 assert(espeak_ng_EncodingFromName("cp819") ==
ESPEAKNG_ENCODING_ISO_8859_1);
 assert(espeak_ng_EncodingFromName("csISOLatin1") ==
ESPEAKNG_ENCODING_ISO_8859_1);
```

```
 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_ISO_8859_1) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xDE);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_8859_2_encoding()
{
 printf("testing ISO-8859-2 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-2") ==
ESPEAKNG_ENCODING_ISO_8859_2);
 assert(espeak_ng_EncodingFromName("ISO_8859-2") ==
ESPEAKNG_ENCODING_ISO_8859_2);
 assert(espeak_ng_EncodingFromName("ISO_8859-2:1987") ==
ESPEAKNG_ENCODING_ISO_8859_2);
 assert(espeak_ng_EncodingFromName("iso-ir-101") ==
ESPEAKNG_ENCODING_ISO_8859_2);
 assert(espeak_ng_EncodingFromName("latin2") ==
ESPEAKNG_ENCODING_ISO_8859_2);
 assert(espeak_ng_EncodingFromName("l2") ==
ESPEAKNG_ENCODING_ISO_8859_2);
```

```
 assert(espeak_ng_EncodingFromName("csISOLatin2") ==
ESPEAKNG_ENCODING_ISO_8859_2);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_ISO_8859_2) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0162);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_8859_3_encoding()
{
 printf("testing ISO-8859-3 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-3") ==
ESPEAKNG_ENCODING_ISO_8859_3);
 assert(espeak_ng_EncodingFromName("ISO_8859-3") ==
ESPEAKNG_ENCODING_ISO_8859_3);
 assert(espeak_ng_EncodingFromName("ISO_8859-3:1988") ==
ESPEAKNG_ENCODING_ISO_8859_3);
 assert(espeak_ng_EncodingFromName("iso-ir-109") ==
ESPEAKNG_ENCODING_ISO_8859_3);
 assert(espeak_ng_EncodingFromName("latin3") ==
ESPEAKNG_ENCODING_ISO_8859_3);
```

```
 assert(espeak_ng_EncodingFromName("l3") ==
ESPEAKNG_ENCODING_ISO_8859_3);
 assert(espeak_ng_EncodingFromName("csISOLatin3") ==
ESPEAKNG_ENCODING_ISO_8859_3);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_ISO_8859_3) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x015C);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_8859_4_encoding()
{
 printf("testing ISO-8859-4 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-4") ==
ESPEAKNG_ENCODING_ISO_8859_4);
 assert(espeak_ng_EncodingFromName("ISO_8859-4") ==
ESPEAKNG_ENCODING_ISO_8859_4);
 assert(espeak_ng_EncodingFromName("ISO_8859-4:1988") ==
ESPEAKNG_ENCODING_ISO_8859_4);
 assert(espeak_ng_EncodingFromName("iso-ir-110") ==
ESPEAKNG_ENCODING_ISO_8859_4);
```

```
 assert(espeak_ng_EncodingFromName("latin4") ==
ESPEAKNG_ENCODING_ISO_8859_4);
 assert(espeak_ng_EncodingFromName("l4") ==
ESPEAKNG_ENCODING_ISO_8859_4);
 assert(espeak_ng_EncodingFromName("csISOLatin4") ==
ESPEAKNG_ENCODING_ISO_8859_4);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_ISO_8859_4) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x016A);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_8859_5_encoding()
{
 printf("testing ISO-8859-5 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-5") ==
ESPEAKNG_ENCODING_ISO_8859_5);
 assert(espeak_ng_EncodingFromName("ISO_8859-5") ==
ESPEAKNG_ENCODING_ISO_8859_5);
 assert(espeak_ng_EncodingFromName("ISO_8859-5:1988") ==
ESPEAKNG_ENCODING_ISO_8859_5);
```

```
 assert(espeak_ng_EncodingFromName("iso-ir-144") ==
ESPEAKNG_ENCODING_ISO_8859_5);
 assert(espeak_ng_EncodingFromName("cyrillic") ==
ESPEAKNG_ENCODING_ISO_8859_5);
 assert(espeak_ng_EncodingFromName("csISOLatinCyrillic") ==
ESPEAKNG_ENCODING_ISO_8859_5);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_ISO_8859_5) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x043E);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_8859_6_encoding()
{
 printf("testing ISO-8859-6 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-6") ==
ESPEAKNG_ENCODING_ISO_8859_6);
 assert(espeak_ng_EncodingFromName("ISO_8859-6") ==
ESPEAKNG_ENCODING_ISO_8859_6);
 assert(espeak_ng_EncodingFromName("ISO_8859-6:1987") ==
ESPEAKNG_ENCODING_ISO_8859_6);
```

```
 assert(espeak_ng_EncodingFromName("iso-ir-127") ==
ESPEAKNG_ENCODING_ISO_8859_6);
 assert(espeak_ng_EncodingFromName("ECMA-114") ==
ESPEAKNG_ENCODING_ISO_8859_6);
 assert(espeak_ng_EncodingFromName("ASMO-708") ==
ESPEAKNG_ENCODING_ISO_8859_6);
 assert(espeak_ng_EncodingFromName("arabic") ==
ESPEAKNG_ENCODING_ISO_8859_6);
 assert(espeak_ng_EncodingFromName("csISOLatinArabic") ==
ESPEAKNG_ENCODING_ISO_8859_6);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDA", 5,
ESPEAKNG_ENCODING_ISO_8859_6) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x063A);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_8859_7_encoding()
{
 printf("testing ISO-8859-7 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-7") ==
ESPEAKNG_ENCODING_ISO_8859_7);
```

```
 assert(espeak_ng_EncodingFromName("ISO_8859-7") ==
ESPEAKNG_ENCODING_ISO_8859_7);
 assert(espeak_ng_EncodingFromName("ISO_8859-7:1987") ==
ESPEAKNG_ENCODING_ISO_8859_7);
 assert(espeak_ng_EncodingFromName("iso-ir-126") ==
ESPEAKNG_ENCODING_ISO_8859_7);
 assert(espeak_ng_EncodingFromName("ECMA-118") ==
ESPEAKNG_ENCODING_ISO_8859_7);
 assert(espeak_ng_EncodingFromName("ELOT_928") ==
ESPEAKNG_ENCODING_ISO_8859_7);
 assert(espeak_ng_EncodingFromName("greek") ==
ESPEAKNG_ENCODING_ISO_8859_7);
 assert(espeak_ng_EncodingFromName("greek8") ==
ESPEAKNG_ENCODING_ISO_8859_7);
 assert(espeak_ng_EncodingFromName("csISOLatinGreek") ==
ESPEAKNG_ENCODING_ISO_8859_7);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_ISO_8859_7) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x03AE);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
```

```
test_iso_8859_8_encoding()
{
 printf("testing ISO-8859-8 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-8") ==
ESPEAKNG_ENCODING_ISO_8859_8);
 assert(espeak_ng_EncodingFromName("ISO_8859-8") ==
ESPEAKNG_ENCODING_ISO_8859_8);
 assert(espeak_ng_EncodingFromName("ISO_8859-8:1988") ==
ESPEAKNG_ENCODING_ISO_8859_8);
 assert(espeak_ng_EncodingFromName("iso-ir-138") ==
ESPEAKNG_ENCODING_ISO_8859_8);
 assert(espeak_ng_EncodingFromName("hebrew") ==
ESPEAKNG_ENCODING_ISO_8859_8);
 assert(espeak_ng_EncodingFromName("csISOLatinHebrew") ==
ESPEAKNG_ENCODING_ISO_8859_8);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xEE", 5,
ESPEAKNG_ENCODING_ISO_8859_8) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x05de);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
```

```
test_iso_8859_9_encoding()
{
 printf("testing ISO-8859-9 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-9") ==
ESPEAKNG_ENCODING_ISO_8859_9);
 assert(espeak_ng_EncodingFromName("ISO_8859-9") ==
ESPEAKNG_ENCODING_ISO_8859_9);
 assert(espeak_ng_EncodingFromName("ISO_8859-9:1989") ==
ESPEAKNG_ENCODING_ISO_8859_9);
 assert(espeak_ng_EncodingFromName("iso-ir-148") ==
ESPEAKNG_ENCODING_ISO_8859_9);
 assert(espeak_ng_EncodingFromName("latin5") ==
ESPEAKNG_ENCODING_ISO_8859_9);
 assert(espeak_ng_EncodingFromName("l5") ==
ESPEAKNG_ENCODING_ISO_8859_9);
 assert(espeak_ng_EncodingFromName("csISOLatin5") ==
ESPEAKNG_ENCODING_ISO_8859_9);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_ISO_8859_9) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x015e);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}
```

```
static void
test_iso_8859_10_encoding()
{
 printf("testing ISO-8859-10 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-10") ==
ESPEAKNG_ENCODING_ISO_8859_10);
 assert(espeak_ng_EncodingFromName("ISO_8859-10") ==
ESPEAKNG_ENCODING_ISO_8859_10);
 assert(espeak_ng_EncodingFromName("ISO_8859-10:1992") ==
ESPEAKNG_ENCODING_ISO_8859_10);
 assert(espeak_ng_EncodingFromName("iso-ir-157") ==
ESPEAKNG_ENCODING_ISO_8859_10);
 assert(espeak_ng_EncodingFromName("latin6") ==
ESPEAKNG_ENCODING_ISO_8859_10);
 assert(espeak_ng_EncodingFromName("l6") ==
ESPEAKNG_ENCODING_ISO_8859_10);
 assert(espeak_ng_EncodingFromName("csISOLatin6") ==
ESPEAKNG_ENCODING_ISO_8859_10);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_ISO_8859_10) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x00de);
 assert(text_decoder_eof(decoder) == 1);
```

```
 destroy_text_decoder(decoder);
}

static void
test_iso_8859_11_encoding()
{
 printf("testing ISO-8859-11 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-11") ==
ESPEAKNG_ENCODING_ISO_8859_11);
 assert(espeak_ng_EncodingFromName("TIS-620") ==
ESPEAKNG_ENCODING_ISO_8859_11);
 assert(espeak_ng_EncodingFromName("csTIS620") ==
ESPEAKNG_ENCODING_ISO_8859_11);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xEE", 5,
ESPEAKNG_ENCODING_ISO_8859_11) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0e4e);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_8859_13_encoding()
{
```

```
 printf("testing ISO-8859-13 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-13") ==
ESPEAKNG_ENCODING_ISO_8859_13);
 assert(espeak_ng_EncodingFromName("csISO885913") ==
ESPEAKNG_ENCODING_ISO_8859_13);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xEE", 5,
ESPEAKNG_ENCODING_ISO_8859_13) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x012b);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_8859_14_encoding()
{
 printf("testing ISO-8859-14 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-14") ==
ESPEAKNG_ENCODING_ISO_8859_14);
 assert(espeak_ng_EncodingFromName("ISO_8859-14") ==
ESPEAKNG_ENCODING_ISO_8859_14);
 assert(espeak_ng_EncodingFromName("ISO_8859-14:1998") ==
ESPEAKNG_ENCODING_ISO_8859_14);
```

```
 assert(espeak_ng_EncodingFromName("iso-ir-199") ==
ESPEAKNG_ENCODING_ISO_8859_14);
 assert(espeak_ng_EncodingFromName("iso-celtic") ==
ESPEAKNG_ENCODING_ISO_8859_14);
 assert(espeak_ng_EncodingFromName("latin8") ==
ESPEAKNG_ENCODING_ISO_8859_14);
 assert(espeak_ng_EncodingFromName("l8") ==
ESPEAKNG_ENCODING_ISO_8859_14);
 assert(espeak_ng_EncodingFromName("csISO885914") ==
ESPEAKNG_ENCODING_ISO_8859_14);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_ISO_8859_14) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0176);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_8859_15_encoding()
{
 printf("testing ISO-8859-15 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-15") ==
ESPEAKNG_ENCODING_ISO_8859_15);
```

```
 assert(espeak_ng_EncodingFromName("ISO_8859-15") ==
ESPEAKNG_ENCODING_ISO_8859_15);
 assert(espeak_ng_EncodingFromName("Latin-9") ==
ESPEAKNG_ENCODING_ISO_8859_15);
 assert(espeak_ng_EncodingFromName("csISO885915") ==
ESPEAKNG_ENCODING_ISO_8859_15);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xBE", 5,
ESPEAKNG_ENCODING_ISO_8859_15) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0178);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_8859_16_encoding()
{
 printf("testing ISO-8859-16 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-8859-16") ==
ESPEAKNG_ENCODING_ISO_8859_16);
 assert(espeak_ng_EncodingFromName("ISO_8859-16") ==
ESPEAKNG_ENCODING_ISO_8859_16);
 assert(espeak_ng_EncodingFromName("ISO_8859-16:2001") ==
ESPEAKNG_ENCODING_ISO_8859_16);
```

```
 assert(espeak_ng_EncodingFromName("iso-ir-226") ==
ESPEAKNG_ENCODING_ISO_8859_16);
 assert(espeak_ng_EncodingFromName("latin10") ==
ESPEAKNG_ENCODING_ISO_8859_16);
 assert(espeak_ng_EncodingFromName("l10") ==
ESPEAKNG_ENCODING_ISO_8859_16);
 assert(espeak_ng_EncodingFromName("csISO885916") ==
ESPEAKNG_ENCODING_ISO_8859_16);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aG\x92\xA0\xDE", 5,
ESPEAKNG_ENCODING_ISO_8859_16) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x92);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x021a);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_utf_8_encoding()
{
 printf("testing UTF-8 encoding\n");

 assert(espeak_ng_EncodingFromName("UTF-8") ==
ESPEAKNG_ENCODING_UTF_8);
 assert(espeak_ng_EncodingFromName("csUTF8") ==
ESPEAKNG_ENCODING_UTF_8);
```

```
 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 // 1-byte UTF-8 sequences
 assert(text_decoder_decode_string(decoder,
"\x0D\x1E\x20\x35\x42\x57\x65\x77", 8, ESPEAKNG_ENCODING_UTF_8)
== ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x000D);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x001E);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0020);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0035);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0042);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0057);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0065);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0077);
 assert(text_decoder_eof(decoder) == 1);

 // UTF-8 tail bytes without an initial length indicator
character
 assert(text_decoder_decode_string(decoder, "\x84\x92\xA8\xB5",
4, ESPEAKNG_ENCODING_UTF_8) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD);
```

```
 assert(text_decoder_eof(decoder) == 1);

 // 2-byte UTF-8 sequences
 assert(text_decoder_decode_string(decoder,
"\xC2\xA0\xD0\xB0\xC5\x65\xC2\xA0", 7, ESPEAKNG_ENCODING_UTF_8)
== ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x00A0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0430);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD); // \x65 is not a
continuation byte, so \xC5 is invalid
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0065);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD); // incomplete:
\xA0 is past the end of the string
 assert(text_decoder_eof(decoder) == 1);

 // 3-byte UTF-8 sequences
 assert(text_decoder_decode_string(decoder,
"\xE4\xBA\x8C\xE8\x42\xE2\x93\x44\xE4\xA0\x80", 9,
ESPEAKNG_ENCODING_UTF_8) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x4E8C);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD); // \x42 is not a
continuation byte, so \xE8 is invalid
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0042);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD); // \x44 is not a
continuation byte, so \xE2\x93 is invalid
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0044);
 assert(text_decoder_eof(decoder) == 0);
```

```
 assert(text_decoder_getc(decoder) == 0xFFFD); // incomplete:
\xA0 is past the end of the string
 assert(text_decoder_eof(decoder) == 1);

 // 4-byte UTF-8 sequences
 assert(text_decoder_decode_string(decoder, "\xF0\x90\x8C\x82\xF4
\x8F\xBF\xBF\xF3\x61\xF3\xA5\x32\xF3\x87\xB2\x36\xF1\xA0\x80\x80"
, 18, ESPEAKNG_ENCODING_UTF_8) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x10302);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x10FFFF);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD); // \x61 is not a
continuation byte, so \xF3 is invalid
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0061);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD); // \x32 is not a
continuation byte, so \xF3\xA5 is invalid
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0032);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD); // \x36 is not a
continuation byte, so \xF3\x87\xB2 is invalid
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x0036);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD); // incomplete:
\xA0 is past the end of the string
 assert(text_decoder_eof(decoder) == 1);

 // out of range (> 0x10FFFF)
 assert(text_decoder_decode_string(decoder, "\xF4\x90\x80\x80",
4, ESPEAKNG_ENCODING_UTF_8) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD);
```

```
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_iso_10646_ucs_2_encoding()
{
 printf("testing ISO-10646-UCS-2 encoding\n");

 assert(espeak_ng_EncodingFromName("ISO-10646-UCS-2") ==
ESPEAKNG_ENCODING_ISO_10646_UCS_2);
 assert(espeak_ng_EncodingFromName("csUnicode") ==
ESPEAKNG_ENCODING_ISO_10646_UCS_2);

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder,
"a\00G\00\xA0\00\x22\x21\x23\x21", 9,
ESPEAKNG_ENCODING_ISO_10646_UCS_2) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x2122);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xFFFD);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_char_decoder()
```

```
{
 printf("testing char decoder\n");

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 // null string
 assert(text_decoder_decode_string(decoder, NULL, -1,
ESPEAKNG_ENCODING_ISO_8859_1) == ENS_OK);
 assert(text_decoder_eof(decoder) == 1);
 assert(text_decoder_getc(decoder) == 0);
 assert(text_decoder_eof(decoder) == 1);

 // string length
 assert(text_decoder_decode_string(decoder, "aG", -1,
ESPEAKNG_ENCODING_ISO_8859_1) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_wchar_decoder()
{
 printf("testing wchar_t decoder\n");

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 // null string
 assert(text_decoder_decode_wstring(decoder, NULL, -1) ==
ENS_OK);
 assert(text_decoder_eof(decoder) == 1);
```

```
 assert(text_decoder_getc(decoder) == 0);
 assert(text_decoder_eof(decoder) == 1);

 // wide-character string
 assert(text_decoder_decode_wstring(decoder, L"aG\xA0\x2045", 4)
== ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x2045);
 assert(text_decoder_eof(decoder) == 1);

 // string length
 assert(text_decoder_decode_wstring(decoder, L"aG\xA0\x2045", -1)
== ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0x2045);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_auto_decoder()
{
```

```
 printf("testing auto decoder (UTF-8 + codepage-based
fallback)\n");

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 // null string
 assert(text_decoder_decode_string_auto(decoder, NULL, -1,
ESPEAKNG_ENCODING_ISO_8859_1) == ENS_OK);
 assert(text_decoder_eof(decoder) == 1);
 assert(text_decoder_getc(decoder) == 0);
 assert(text_decoder_eof(decoder) == 1);

 // UTF-8
 assert(text_decoder_decode_string_auto(decoder, "aG\xC2\xA0 ",
5, ESPEAKNG_ENCODING_ISO_8859_1) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == ' ');
 assert(text_decoder_eof(decoder) == 1);

 // ISO-8859-1
 assert(text_decoder_decode_string_auto(decoder, "aG\240f", 4,
ESPEAKNG_ENCODING_ISO_8859_1) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0xA0);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'f');
 assert(text_decoder_eof(decoder) == 1);
```

```
 // string length
 assert(text_decoder_decode_string_auto(decoder, "aG", -1,
ESPEAKNG_ENCODING_ISO_8859_1) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 0);
 assert(text_decoder_eof(decoder) == 1);

 destroy_text_decoder(decoder);
}

static void
test_peekc()
{
 printf("testing peekc\n");

 espeak_ng_TEXT_DECODER *decoder = create_text_decoder();

 assert(text_decoder_decode_string(decoder, "aGd", 3,
ESPEAKNG_ENCODING_US_ASCII) == ENS_OK);
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'a');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_peekc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'G');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_peekc(decoder) == 'd');
 assert(text_decoder_eof(decoder) == 0);
 assert(text_decoder_getc(decoder) == 'd');
 assert(text_decoder_eof(decoder) == 1);

 // Calling peekc past the end of the buffer.
```

```c
  assert(text_decoder_peekc(decoder) == '\0');
  assert(text_decoder_eof(decoder) == 1);

  destroy_text_decoder(decoder);
}

int
main(int argc, char **argv)
{
  (void)argc; // unused parameter
  (void)argv; // unused parameter

  test_unbound_text_decoder();
  test_unknown_encoding();

  test_us_ascii_encoding();
  test_koi8_r_encoding();
  test_iscii_encoding();

  test_iso_8859_1_encoding();
  test_iso_8859_2_encoding();
  test_iso_8859_3_encoding();
  test_iso_8859_4_encoding();
  test_iso_8859_5_encoding();
  test_iso_8859_6_encoding();
  test_iso_8859_7_encoding();
  test_iso_8859_8_encoding();
  test_iso_8859_9_encoding();
  test_iso_8859_10_encoding();
  test_iso_8859_11_encoding();
  // ISO-8859-12 is not a valid encoding.
  test_iso_8859_13_encoding();
  test_iso_8859_14_encoding();
  test_iso_8859_15_encoding();
  test_iso_8859_16_encoding();

  test_utf_8_encoding();
```

```
  test_iso_10646_ucs_2_encoding();

  test_char_decoder();
  test_wchar_decoder();
  test_auto_decoder();

  test_peekc();

  return EXIT_SUCCESS;
}
```

Chapter 72

./tests/ssml-fuzzer.c

```
#include "config.h"

#include <stdint.h>
#include <stdlib.h>

#include <espeak-ng/espeak_ng.h>

static int initialized = 0;

static int SynthCallback(short *wav, int numsamples, espeak_EVENT
*events) {
 /* prevent warning for unused arguments */
 (void) wav;
 (void) numsamples;
 (void) events;

 return 0;
}

extern int LLVMFuzzerTestOneInput(const uint8_t *data, size_t
size);
extern int LLVMFuzzerTestOneInput(const uint8_t *data, size_t
```

```
size) {
 if (!initialized) {
  espeak_Initialize(AUDIO_OUTPUT_SYNCHRONOUS, 0, NULL, 0);
  espeak_SetSynthCallback(SynthCallback);
  initialized = 1;
 }

 int synth_flags = espeakCHARS_UTF8 | espeakPHONEMES |
espeakSSML;
 espeak_Synth((char*) data, size + 1, 0, POS_CHARACTER, 0,
              synth_flags, NULL, NULL);

 return 0;
}
```

Chapter 73

./emscripten/espeakng_glue.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <emscripten.h>
#include "speak_lib.h"

static int gSamplerate = 0;

class eSpeakNGWorker {
public:
  eSpeakNGWorker() : rate(espeakRATE_NORMAL), pitch(50),
current_voice(NULL) {
    if (!gSamplerate) {
      gSamplerate = espeak_Initialize(
        AUDIO_OUTPUT_SYNCHRONOUS, 100, NULL,
espeakINITIALIZE_DONT_EXIT);
    }
    samplerate = gSamplerate;
    voices = espeak_ListVoices(NULL);
  }

  void synth_(const char* aText, void* aCallback) {
    t_espeak_callback* cb =
```

```
reinterpret_cast<t_espeak_callback*>(aCallback);
    espeak_SetSynthCallback(cb);
    espeak_SetParameter(espeakPITCH, pitch, 0);
    espeak_SetParameter(espeakRATE, rate, 0);

    if (current_voice)
      espeak_SetVoiceByProperties(current_voice);
    else
      espeak_SetVoiceByName("default");

    espeak_Synth(aText, 0, 0, POS_CHARACTER, 0, 0, NULL, NULL);

    // Reset callback so other instances will work too.
    espeak_SetSynthCallback(NULL);
  }

  int synth_ipa_(const char* aText, const char* virtualFileName)
{

  /* phoneme_mode
    bit 1:   0=eSpeak's ascii phoneme names, 1= International
Phonetic Alphabet (as UTF-8 characters).
    bit 7:   use (bits 8-23) as a tie within multi-letter
phonemes names
    bits 8-23:  separator character, between phoneme names
  */

    espeak_SetSynthCallback(NULL);

    int phoneme_options                = (1 << 1); // Use IPA
    int use_custom_phoneme_separator  = (0 << 7);
    int phonemes_separator             = ' '; // Use a default
value

    int phoneme_conf                   = phoneme_options |
(phonemes_separator << 8);
```

```
    FILE* f_phonemes_out = fopen(virtualFileName,"wb");
    if(!f_phonemes_out)
      return -1;

    //espeak_ng_InitializeOutput(ENOUTPUT_MODE_SYNCHRONOUS, 0,
NULL);
    espeak_SetPhonemeTrace(phoneme_conf, f_phonemes_out);
    espeak_Synth(aText, 0, 0, POS_CHARACTER, 0, 0, NULL, NULL);
    espeak_SetPhonemeTrace(0, NULL);
    fclose(f_phonemes_out);

    return 0;
  }

  long set_voice(
        const char* aName,
        const char* aLang=NULL,
        unsigned char aGender=0,
        unsigned char aAge=0,
        unsigned char aVariant = 0
    ) {
    long result = 0;
    if (aLang || aGender || aAge || aVariant) {
      espeak_VOICE props = { 0 };
      props.name = aName;
      props.languages = aLang;
      props.gender = aGender;
      props.age = aAge;
      props.variant = aVariant;
      result = espeak_SetVoiceByProperties(&props);
    } else {
      result = espeak_SetVoiceByName(aName);
    }

    // This way we don't need to allocate the name/lang strings
to the heap.
    // Instead, we store the actual global voice.
```

```
    current_voice = espeak_GetCurrentVoice();

    return result;
  }

  int getSizeOfEventStruct_() {
    return sizeof(espeak_EVENT);
  }

  const espeak_VOICE** voices;
  int samplerate;
  int rate;
  int pitch;

private:
  espeak_VOICE* current_voice;
};

#include <glue.cpp>
```