THE HAPI BOOK

A description of the HTK Application Programming Interface

Julian Odell
Dan Kershaw
Dave Ollason
Valtcho Valtchev
David Whitehouse

Version 1.4

The HAPI Book (for HAPI Version 1.4)

First published: April 1997

Reprinted: January 1999

Revised for HAPI Version 1.4: January 1999

Contents

Part I

Tutorial Overview

Chapter 1

An Introduction to HAPI

The HTK Application Programming Interface (HAPI) is a library of functions providing the programmer with an interface to any speech recognition system supplied by Entropic or developed using the Hidden Markov Model Toolkit (HTK). HTK is a set of UNIX tools which are used to construct all the components of a modern speech recogniser. One of the principal components which can be produced with HTK is a set of Hidden Markov Model (HMM) based acoustic speech models. Other components include the pronunciation dictionary, language models and grammar used for recognition. These can be produced using HTK or Entropic's *grapHvite* Speech Recognition Developer System. HAPI encapsulates these components and provides the programmer with a simple and consistent interface through which they can integrate speech recognition into their applications.

Given a set of acoustic models, a dictionary, and a grammar, each unknown utterance is recognised using a decoder. This is the search engine at the heart of every speech recognition system. The core HAPI package may be shipped with a variety of decoders, for example the standard HTK decoder, the professional **MVX** decoder for medium vocabulary tasks, or the professional **LVX** decoder for large vocabulary tasks. In addition to the decoder, the recogniser also requires an appropriate dictionary and set of HMMs. These HMMs should be trained on speech data which is similar to that which will be recognised. For example, HMMs trained on wide band data collected through a high quality desk top microphone will not perform well if used to recognise speech over the telephone. The dictionary used should provide pronunciations, in terms of models, for all words in the intended recognition vocabulary.

The developer has the choice of either generating their own recognition components or licensing components from the wide range that Entropic has to offer. Entropic supplies pre-trained models for a number of languages and environments together with the corresponding dictionaries. For other requirements, custom models and dictionaries can be built using HTK, which provides a framework in which to produce and evaluate the accuracy and performance of all major types of recognition system.

The particular route taken to produce the recogniser components is irrelevant as far as HAPI is concerned. Having acquired the necessary components, HAPI is all that is required to produce both prototype and commercial applications. HAPI provides the programmer with a simple programming interface to the chosen components allowing them to incorporate speech recognition into their application with the minimum of effort. Although HAPI can be viewed as an extension to HTK (since it uses components produced using HTK and shares the same code libraries) it is better

viewed as a stand alone interface. The underlying recogniser components of a HAPI application can be upgraded with no need to modify any existing application code.

The combination of HAPI, HTK and Entropic's "off the shelf" decoders provides the application developer with the complete set of tools and components required to produce and utilise state-of-the-art technology in applications spanning the entire spectrum of current uses of speech recognition.

This book is divided into four main parts. This first part describes the components of a speech recognition system and provides a high level overview of the capabilities of HAPI in the form of a tutorial on how to build a simple application – a phone dialer. Part 2 provides an in depth description of the facilities available within HAPI and how these facilities relate to HTK for those familiar with the HTK toolkit and who use it to produce their own recognition systems. Part 3 describes a few example programs and touches on areas not directly related to HAPI but which are nonetheless important when producing a recognition system for a real application (such as semantic parsing and dialogue management). Finally the appendices describe the differences between the various flavours of HAPI (due to the different programming languages being used to implement the specification) and provides a complete reference section.

This book does not cover the production of the recognition components (such as the acoustic models or word networks). These processes are described in some detail in the *HTK Book*and the *grapHvite* manual. Although there is a brief description of speech recognition systems in this chapter it is neither detailed nor comprehensive. For more information on the principles and algorithms used in speech recognisers the reader is advised to read the *HTK Book*as well as general literature from the field.

1.1 General Principles of Recognition

A speech recogniser can be viewed as a device which converts an acoustic signal into a sequence of labels (as shown in figure 1.1).

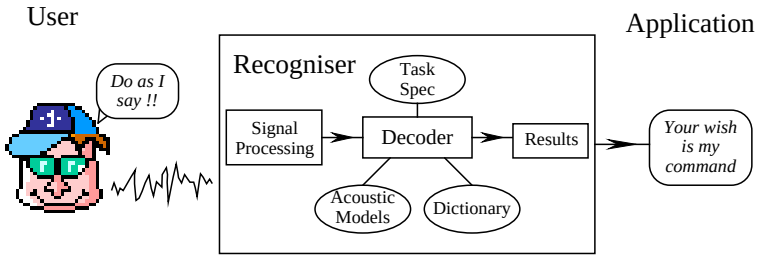

Fig. 1.1 The Speech Recognition Process

The system takes the raw acoustic data, processes it through the recogniser which then matches it against a set of models using a decoder that generates a recognition hypothesis. This hypothesis takes the form of a series of tokens which hopefully represent what the user wanted to say. Each token could represent a single word, a syllable or a complete phrases. The recogniser makes no attempt to derive meaning from the recognised output. Any meaning derived from or actions taken as a consequence of the recogniser output are the responsibility of the application.

The recogniser in HAPI uses Hidden Markov Models (HMMs) to implement a continuous speech recogniser. Markov models are generative statistical models from

which a sequence of observations are produced as the model advances through its states (see figure 1.2). During recognition the likelihood of each model (or sequence of models) generating the observed data is calculated. Comparing the likelihood of each model (or sequences of models) allows the decoder to recognise the unknown speech by selecting the most likely. One of the reasons for the use of HMMs is the existence of a computationally efficient algorithm, the Viterbi algorithm, for finding the most likely sequence of models. Another is the ability to recognise continuous speech as opposed to isolated words. Removing the requirement for a pause between words allows a much more natural speaking style. Both of these features are due to the way in which HMMs can be joined together to generate a network representing all recognisable utterances - the task grammar.

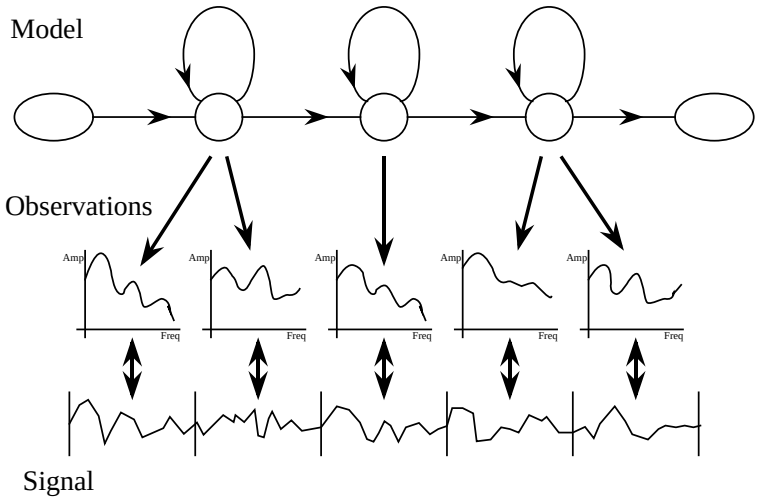

Fig. 1.2 The Generative Markov Model

HMMs do not process the acoustic signal directly but require that the continuous signal be split into a sequence of discrete observations. These observations are normally derived from a spectral representation of the signal, which is assumed to be reasonably constant over the duration of input data represented by each observation. Because of this dependence upon the spectra of the signal, the channel used to capture the data also affects the observations. This dependency means that different acoustic models are needed for different spectral conditions. For example, a set of models trained using wide band speech from a close talking microphone will perform poorly with band limited speech from a telephone.

The sequence of observations is processed in conjunction with the network of HMMs by a Viterbi decoder which finds the most likely path through the network based upon the likelihood of the individual models. By recording sequence information whilst calculating the likelihoods it is possible to reconstruct the most likely sequence of models once the end of the utterance has been reached. Each model typically represents a short sound and models are combined into *words* using a dictionary specifying the pronunciation of each word in terms of the acoustic models. In this case the network and the recognition hypothesis are composed of *words* (figure 1.3 shows and example word level network). Note, however, that each *word* does not have to represent a word in the normal linguistic sense and could instead represent a

phoneme, a syllable or a complete phrase simply by changing entries in the dictionary. In the remainder of this document the term word will be used when referring to these recognition tokens (which will often be words in the normal linguistic sense).

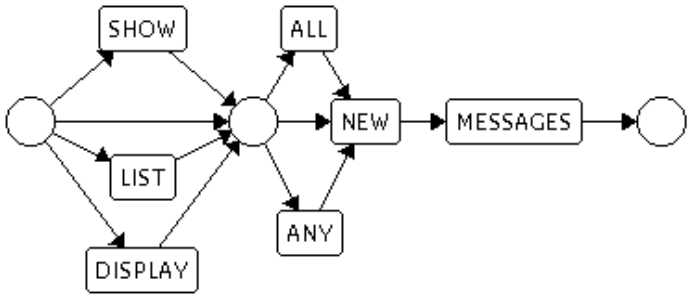

Fig. 1.3 Example Network

This method of operation means that the recogniser will hypothesise the most likely path through the network irrespective of whether this is a close match. There is no explicit test for the speech closely matching the models. All that can be assumed from the result is that the speech matches the hypothesised sequence best. In order to provide rejection of mal-formed utterances it is necessary to introduce another path through the network. This path needs to represent "everything else" that can be said to the recogniser and should be chosen when the input speech is a poor match with the sequence that would otherwise have been recognised. Such a path can be provided by placing a generic model in parallel with the recognition grammar. When the recognition grammar is a poor fit with the input speech the generic model should be more likely and if this occurs the utterance is rejected.

From the foregoing it will be apparent that automatic speech recognition is a fairly complex process involving the interaction of a number of components. Each of these components are listed below along with a brief description of their role:

The purpose of the **recogniser** is to generate a hypothesis for the unknown utterance. This hypothesis is normally the sequence of word tokens that best matches the acoustic signal but could consist of a lattice of hypotheses when alternatives are required.

The core of the recogniser is the **decoder** which matches a sequence of observations against a network of possibilities.

The sequence of observations is produced by a **coder** which takes the raw acoustic signal, performs various signal processing and generates a series of discrete observations that hopefully are a faithful representation of the acoustic signal.

The acoustic signal can come from a variety of **sources**. These control the acquisition of the data either from files prepared off line and stored on disk or direct from a real-time source of sampled data such as the workstation's audio input. HAPI includes an interface that allows the application to define a **driver** which allows the programmer to use HAPI to process data from any source.

The **network** used to process the observations is expressed in terms of individual HMMs which are used to actually calculate the likelihood of each observation. This

network is normally generated from a task grammar which is normally stored as a **lattice** of word level alternatives[1]. However, in large vocabulary systems designed for dictation type applications the network may be generated from a statistical language model.

The **dictionary** describes how to expand each word in the network into a sequence of acoustic models.

These acoustic models form a **HMMSet** which is used to provide the decoder with a method of calculating likelihoods for the individual sounds making up each hypothesis.

1.2 The Position and Role of HAPI

Figure 1.4 shows position of HAPI and HTK in a speech enabled application.

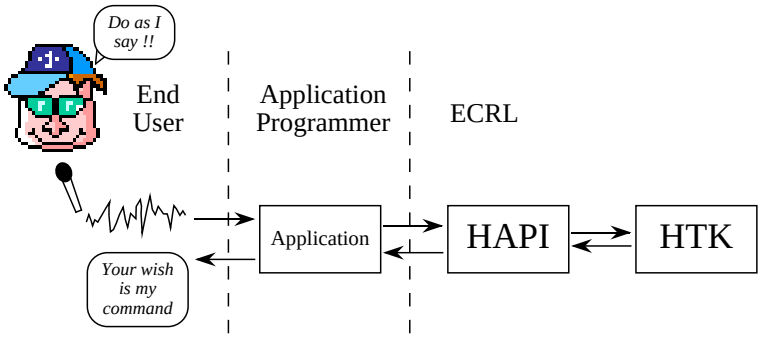

Fig. 1.4 The Application HAPI and HTK

The application manages the interaction with the user (even when HAPI or HTK are directly interfacing to the audio sub-system the application still controls the interaction). The interface to the user may take the form of a single audio channel in the case of a telephone application or may include a combination of audio and visual elements in the case of a desktop application. In either case, HAPI acts as an interface between the application code and the underlying HTK code which implements the core speech recognition system.

The HAPI library is built on top of the HTK libraries which implement the speech recognition system. Like the HTK libraries, HAPI is written in the 'C' programming language. There are however extensions, in the form of wrappers around the core 'C' library, which allow HAPI to be used in other programming environments. For example, there is a version of HAPI called JHAPI that is implemented in JAVA and is well suited to writing platform independent graphical applications.

From the outset HAPI has been designed to be as versatile and flexible as possible and consequently it appears to be a relatively low level interface. This high level of flexibility provides the programmer with a fine degree of control over all aspects of the speech recognition process. At the same time, HAPI has been designed in such

[1] Throughout this document the terms *network* and *lattice* are used somewhat interchangeably. The important distinction is that the recogniser uses the *network* to specify the task syntax. Often this will be specified as a word level *lattice*, however, before recognition, the word level *lattice* must be expanded in terms of the acoustic models in order to generate the *network* needed by the recogniser.

a way as to allow common simple tasks to be accomplished with the minimum of application code. The range of facilities supported by HAPI includes loading the key recogniser components from disk; generating, using and modifying multiple dictionaries and recognition grammars; reading (and writing) speech data from a variety of sources (HAPI includes drivers for "live" audio input and files but can use application defined drivers to recognise data from any source); generating and processing recognition results (including the ability to annotate words with semantic keys to simplify application processing) and producing alternative recognition hypotheses.

As well as providing these basic recognition facilities, the API must also allow the user to produce and evaluate complete interactive applications. Consequently it also contains functions for audio playback; the saving to and loading from disk of audio data and results; the ability to create and modify dictionaries and syntaxes; recognition in different modes suited to different types of applications and even the ability to use different recognisers for different tasks within a single application. For the programmer wishing to really get to grips with the speech part of the application, facilities are provided for direct access to the input waveform, to access detailed timing and scoring information and even the ability to process the same piece of data using multiple recognisers.

Chapter 2

An Overview of HAPI

This chapter discusses the basic design, configuration, memory management and use of HAPI.

2.1 Basics

As mentioned in the previous chapter HAPI and HTK are written in the 'C' programming language. This maximises performance whilst maintaining a good degree of platform independence. To ensure that 'good' programming practices are followed an object oriented programming style has been adopted in the design of both HAPI and HTK. The previous chapter discussed the speech recognition process in general and highlighted the fact that it comprises a number of distinct components and stages. Within HAPI the recogniser is broken into several distinct components, each of which is represented as an **Object** with associated functionality.

In general the use of each object is reasonably uniform and goes through a series of stages; Creation, setup, initialisation (including loading from disk when necessary), use and finally deletion. The creation phase allocates memory for the object and initialises various parameters (such as filenames and other configuration parameters) to default values. These defaults can be overridden during the setup phase by the application prior to the initialisation phase during which the actual data associated with the object is read into memory and the object is prepared for use. In many cases, however, there is no need for extra setting up because the default values, which are read from a configuration file, are those required. And for the simpler objects, the first three stages may be encompassed into a single function call.

During the use phase, the objects are used by the application to perform any required tasks. These can range from speech capture through to the final output of results.

Finally once the application has finished using the object, it should be deleted to free the memory and resources it was using.

2.2 Configuration

A single configuration file is used to set parameters for both HAPI and also the underlying HTK libraries. Each parameter is set by a line in the file of the form

```
[MODULE:] PARAMETER = VALUE
```

The value can be of one of four types; string, boolean, integer or floating point number (although conversion of integers and floating point numbers happens automatically). Booleans should be set to `T`, `TRUE`, `F` or `FALSE`. Integers are normally decimal numbers although they are treated as octal if they start with a `0` and hexadecimal when preceded with `0x`. Floating point numbers are always expressed in base 10 although scientific notation is permitted. All other values are treated as strings although this can be made explicit by quoting the string and quoting is necessary if the string contains white space.

Comments can be present either on lines by themselves or following the `VALUE` field and are preceded by the `#` character.

The `MODULE` field is optional and when it is missing the `PARAMETER` setting applies to all modules. However, since many of the `PARAMETER` names are unique and used in only a single module both forms will frequently have the same effect.

The `TRACE` variable is an example of a non-unique `PARAMETER` which requires the `MODULE` field to be set if module level tracing is desired. For example if the configuration file contains the following lines

```
       TRACE = 1
HAPI:  TRACE = 0777
HParm: TRACE = 020
```

basic level tracing will be enabled in all HTK libraries, whilst in HAPI and `HParm` extra tracing will be generated. Note that the `TRACE` parameter is used in all modules and so the generic setting of `TRACE = 1` is meaningful.

Many of the variables within the configuration file will be used by HTK and will have no direct effect on the HAPI interface but will control how the underlying HTK libraries perform their tasks. In general the settings for these variables are predetermined by the methods used to generate the recogniser components. For example some of the parameters are used to control how the acoustic signal is converted into discrete observations. The settings for these parameters (distinguished by the module name `HParm`) should be the same as those used to train the HMMs. The table below shows the HTK modules which are used by the HAPI library.

Module name	Function
`HAPI`	The HAPI interface itself
`HSHELL`	File IO and configuration handling
`HWAVE`	Waveform file handling
`HPARM`	General coder behaviour
`HAUDIO`	Direct audio input
`HADAPT`	Provides adaptation facility
`HNET`	Recognition network expansion
`HREC`	Recogniser operation

In HTK the configuration file is used to set system dependent default parameters with command line parameters used to specify file locations and what each program should do. For HAPI these tasks are accomplished by a combination of configuration parameters and function calls. Because of the differences in the way in which configuration parameters are used, it is important that all HAPI configuration parameters can be overridden by calls in the application. Since these include filenames, recognition control parameters and switches to enable/disable specific features it is important

to be able to change them during program execution. For instance, if it is necessary to load two separate HMM sets, the only way that the filenames for the second set of models can be specified is by overriding the HMMSet name in the configuration file using calls from the application.

Note however that at present the HTK configuration parameters are scanned when HAPI is initialised but are not updated during the execution of the program and so overriding them has no effect[1]. Normally this is not a problem as the HTK part of the configuration file contains parameters that do not need to be changed during the execution of a single program (compatibility flags, file format selections and enabling/disabling of features).

2.3 Type definitions and memory allocation

In order to work correctly on a number of platforms HAPI specifies particular sizes for most underlying types.

Type	Bytes	Description
`char8`	1	8-bit character
`char8s`	1	signed 8-bit integer
`char8u`	1	unsigned 8-bit integer
`short16`	2	signed 16-bit integer
`short16u`	2	unsigned 16-bit integer
`int32`	4	signed 32-bit integer
`int32u`	4	unsigned 32-bit integer
`float32`	4	single precision IEEE format floating point number
`double64`	8	double precision IEEE format floating point number
`hapiTime`	8	time in milliseconds (as a double64)

The one exception being a pointer (to any of the quantities above or the actual object references themselves) which for practical reasons is always the default size for the machine and operating system being used.

In general memory allocation is handled within HAPI and the return value of most functions is either a simple quantity or a references to pre-existing parts of the memory used by HAPI or HTK. In these cases the returned values are valid until the relevant object is destroyed (or sometimes even for the whole duration of the program). When this is not the case and the returned value is only valid for a limited period its lifetime will be made explicitly clear in the description of each function.

Many HAPI functions return an array of values. In these cases the allocation is handled partly by the application and partly by HAPI. The function returns the size of the array and when the array of values is required, it expects the application to provide a pointer to an array of the necessary size. This can be accomplished by initially calling the function passing `NULL` as the array parameter, then allocating an array of the correct size and finally calling the function again to get the array of return values.

This is best explained by an example. The function **hapiDictAllPhoneNames** returns the number of phones in the phone set of a particular dictionary. It can

[1] This is an aspect of HAPI's behaviour that is planned to change. As HTK is developed and further integration with HAPI takes place the configuration will be rescanned when necessary in order to ensure that its behaviour matches that specified by the current configuration.

optionally fill in a lookup table (as an array of `char` pointers) of their names. The application finds the size of this array by passing a `NULL` pointer in an initial call before allocating the array and then calling the function again.

```
n=hapiDictAllPhoneNames(dict,NULL);
names = (char8**) malloc((n+1)*sizeof(char8*));
hapiDictAllPhoneNames(dict,names);
for (i=1;i<=n;i++) printf("Phone %d == %s\n",i,names[i]);
```

The application is responsible for creating the array but the actual character strings are allocated within HAPI's memory space.

2.4 Error Reporting

The HTK module HSHELL provides HAPI with a standard mechanism for reporting errors and warnings. A typical error message is as follows

```
HRec: ERROR [+8571]
ProcessObservation: incompatible number of streams
```

This indicates that the module HREC is reporting an error number +8571. All errors have positive error numbers and are returned to the calling HAPI object. They are output to the standard error stream. Warnings have negative error numbers and are not returned. They are sent to the standard output stream. The first two digits of an error number indicate the module in which the error occurred (HREC in this case) and the last two digits define the class of error. The second line of the error message names the actual routine in which the error occurred (here `Process Observation`) and the actual error message. All errors and warnings are listed in the reference section at the end of this book indexed by error/warning number. This listing contains more details on each error or warning along with suggested causes.

2.5 Strings and Names

HAPI uses the standard HTK rules for reading strings which are names. A name string consists of a single white space delimited word or a quoted string. Either the single quote ' or the double quote " can be used to quote strings but the start and end quotes must be matched. The backslash \ character can also be used to introduce otherwise reserved characters. The character following a backslash is inserted into the string without special processing unless that character is a digit in the range 0 to 7. In that case, the three characters following the backslash are read and interpreted as an octal character code. When the three characters are not octal digits the result is not well defined.

In summary the special processing is

Notation	Meaning
\\	\
_	represents a space that will not terminate a string
\'	' (and will not end a quoted string)
\"	" (and will not end a quoted string)
\nnn	the character with octal code \nnn

Note that the above allows the same effect to be achieved in a number of different ways. For example,

```
"\"QUOTE"
\"QUOTE
'"QUOTE'
\042QUOTE
```

all produce the string `"QUOTE`.

Note that under some operating systems, HAPI can support the 8-bit character sets used for the representation of various orthographies. In such cases the shell environment variable `$LANG` usually governs which ISO character set is in use.

2.6 Speech Input/Output

HAPI accepts speech input from a variety of sources, and that speech may arrive in various formats. In this section, various conventions related to speech input will be listed and discussed.

There are three principal means by which HAPI will receive speech:

- input from a previously encoded speech parameter file
- input from a waveform file which is encoded as part of the input processing
- input from an audio device which is encoded as part of the input processing.

The distinction between file and direct audio input is controlled directly by the **hapiSourceObject**. For input from a waveform file, a large number of different file formats are supported, including all of the commonly used CD-ROM formats. Input/output for parameter files is limited to the standard HTK file format, and such files would presumably be created using Entropic's HTK Toolkit.

Various aspects of HAPI speech input are controlled by configuration parameters which give details of what processing operations to apply to each input speech file or audio source. These configuration variables control the behavior of the HTK library modules HAudio, HWave, HParm and HSigP. Waveforms are read from files using HWave, or are input directly from an audio device using HAudio. HAPI speech input must be parameterised to match the particular HMM set being used for recognition, and this encoding is performed by HParm using the signal processing operations defined in HSigP.

The direct input of parameterised speech will not be relevant to most HAPI developers; information on this topic can be found in the *HTK Book*. Here, it will be assumed that speech input will come from waveform files or directly from an audio device. Thus the configuration variable `SOURCEKIND` can normally be set to the value `WAVEFORM`. However, note that this variable has by default the value `ANON`, which tells HAPI to determine for itself what type of speech is being input. The configuration variables which control the parameterisation library HParm should reflect the settings required by the given HMM set. For example, the configuration variable `TARGETKIND` should correspond to those used when training the HMMs with HTK. There are a number of HParm configuration variables of this kind; these are mentioned in more detail in section 11.1. For HMMs provided by Entropic, a full configuration file with the correct settings is provided.

When using direct audio input, the input sampling rate may be set explicitly using the configuration parameter SOURCERATE, otherwise HAPI will assume that it has been set by some external means such as an audio control panel. In the latter case, it must be possible for HAUDIO to obtain the sample rate from the audio driver otherwise an error will be generated.

Although the detailed control of audio hardware is typically machine dependent, HAPI provides a number of Boolean configuration variables to request specific input and output sources. These are indicated by the following table

Variable	Source/Sink
LINEIN	line input
MICIN	microphone input
LINEOUT	line output
PHONESOUT	headphones output
SPEAKEROUT	speaker output

The sample rate of waveform files will normally be determined from the input file itself, but it can also be explicitly set using the configuration parameter SOURCERATE. Note that for historical reasons, HTK specifies durations in units of 100 nsecs, thus source rates are also expressed in relation to such units (e.g., a 16kHz sampling rate translates to a source rate of 625).

As waveform files can appear in a variety of formats, the configuration variable SOURCEFORMAT is used to specify the format of the input speech. The following waveform formats are supported:

Name	Description
HTK	The standard HTK file format
TIMIT	As used in the original prototype TIMIT CD-ROM
NIST	The standard SPHERE format used by the US NIST
SCRIBE	Subset of the European SAM standard used in the SCRIBE CD-ROM
SDES1	The Sound Designer 1 format defined by Digidesign Inc.
AIFF	Audio interchange file format
SUNAU8	Subset of 8bit ".au" and ".snd" formats used by Sun and NeXT
OGI	Format used by Oregan Graduate Institute similar to TIMIT
WAV	Microsoft WAVE files used on PCs
ESIG	Entropic Esignal file format
AUDIO	Pseudo format to indicate direct audio input
ALIEN	Pseudo format to indicate unsupported file, the alien header size must be set via the environment variable HDSIZE
NOHEAD	As for the ALIEN format but header size is zero

Table. 2.1 Supported File Formats

2.7 HAPI Objects

Within HAPI, the recogniser is divided into eight components and each component is represented as an object. The underlying structure of the object is completely

hidden from the application (which is passed an address that serves as a reference to the object). The only access that the application has to the object is through the functions associated with the object (which can be viewed as methods). This ensures source code compatibility for applications over different versions of the HAPI library despite any changes in the underlying object structures.

The recogniser consists of the following components:

- HMMSet

  The acoustic models take the form of Hidden Markov Models (HMMs). For convenience these are combined into sets encompassing the range of sounds that need to be recognised. HAPI can use both HMMs provided by Entropic (for instance those shipped with *grapHvite*) and those developed using HTK.

- Transform

  In HAPI, adaptation is accomplished by storing a set of transformations (per speaker), which are applied to the acoustic models (HMMs). After this transformation the acoustic models become more representative of a new speaker and environment.

  Adaptation can also play a part in the recognition process. During online adaptation the recognised speech data together with the recognition hypothesis from the results object can be used to refine the transformations. The new transformations can be applied to the acoustic models, producing an improved representation of the model set for the current user.

- Dictionary

  The output of HAPI is in the form of words. Normally each HAPI word will correspond directly with a real underlying word. However this is not necessarily the case and there is no reason why the output from HAPI needs to correspond directly with linguistic words. Each HAPI word could be a single phone, a syllable or a short phrase. Words are stored in standard HTK dictionary format which is illustrated by the following examples:

```
WORD         [OUTPUT_SYMBOL]  PRONPROB  phone1 phone2
TWENTY_FIVE  [25]                       t w eh n t iy sp f ay v sp
SHORT_PHRASE                            sh ao r t sp f r ey z sp
AY           [phone_ay]                 ay
A                             0.75      ax sp
A                             0.25      ey sp
```

  The dictionary is used to map the word tokens to acoustic models and as the above shows, it can also define an output representation for each token. In the above example, the phrase "Twenty five" is associated with the output symbol `25` as well as the word name `TWENTY_FIVE`. The optional `PRONPROB` indicates the pronunciation probability.

  If the dictionary needs to be modified by HAPI an extra entry is required to specify the phone-set used by the dictionary. This entry (`!PHONE-SET`) has a pronunciation that consists of all the phones in the phone set. For example

```
!PHONE-SET aa ae ah ao aw ax ay b ch d dh eh el en er ey f g hh ih iy
           jh k l m n ng ow oy p r s sh sil sp t th uh uw v w y z zh
```

HAPI can use dictionaries supplied by Entropic (for example those shipped with *grapHvite*) as well as those constructed using HTK or HAPI itself.

- Lattice

  Each recognition task needs to define the set of recognisable words and phrases. This task specification (or grammar) is normally stored as a lattice of allowable word sequences. Lattices can be built using Entropic products such as HTK and *grapHvite*, or can be constructed within HAPI itself.

  This same object is used to store multiple hypotheses output from the recogniser. Lattices generated from recognition results contain additional information about the likelihood and timing of each word. This extra information allows them to be used to generate multiple alternative hypotheses for parts of the utterance.

- Network

  In order to actually perform recognition, the set of HMMs, the lattice of word alternatives and the dictionary need to be combined to provide the recogniser with a structure that it can use to decode each sentence.

  HAPI uses the facilities in HTK for producing and manipulating networks. Normally this means that the network is derived from a lattice, a dictionary and a set of HMMs. However, since the network may be decoder dependent this is not always the case. For instance a dictation decoder may allow any word to follow any other (making the lattice redundant) and use a statistical language model to provide a likelihood for each word sequence.

- Source

  Each task may require a different method of obtaining the acoustic signal to be processed. The signal may be coming from a microphone, an amplifier, another computer or a pre-recorded file. The source object is responsible for managing the acquisition of the acoustic signal.

  From the application programmers point of view the source is viewed as operating in real time. This means that calls to start and stop the source are effective immediately and that functions querying source values (for example the current volume level) provide the value at that instant irrespective of how much of the utterance has been coded and processed.

  The actual interface to the source is relatively easy to expand with only a small number of functions required for each new source type. The standard HAPI library supports direct audio from the workstation and file based processing (from multiple format waveform files as well as HTK format parameterised data). It also includes an interface that allows the application to manage the acquisition of the raw signal. This allows the programmer to write a driver for their particular sampling device or telephony card and still use HAPI in a 'real-time' fashion to process the data (without needing to write the audio data into a file before processing it).

- Coder

  Once the raw acoustic signal is available it must be converted into the form required by the HMMs. The coder processes the digitised signal and converts it into a series of vectors called observations.

The coder provides buffering between the source (which runs as close to real time as the underlying operating system will allow) and the recogniser (that will sometimes be compute bound and running slower) whilst converting the input signal into the observations required by the set of HMMs in use.

Once the whole utterance has been read the coder provides access to the observations (and when appropriate the waveform) for further recognition, saving to disk (for keeping records, later replay and even off-line experimentation or adaptation) and further application processing.

- Recogniser

  This object manages the decoder that matches the sequence of observations from the coder against the network of HMMs and produces a hypothesis for the unknown speech. This hypothesis may consist of a single sequence of words but, if the decoder is set up appropriately, it may include a lattice of possible alternatives answers.

- Results

  Once the decoder has matched the signal against the network it will hypothesise a sequence of words as being the best fit. These results can be accessed to determine what the sequence is, where the word boundaries are and if there are any alternatives hypotheses found.

Figure 2.1 shows the object hierarchy and dependences. The HMMSet, a dictionary and a word lattice (which may be loaded as an explicit HAPI lattice object or read directly from disk) are used to construct the recognition network object. The source object manages the acquisition of speech which is then converted into observations by the coder object. The recogniser object makes use of the network object in decoding the output from the coder and produces output in the form of a results object (which can be used to access the most likely hypothesis or converted into a lattice object to access multiple alternative hypotheses.

The lattice object is used in two places. To simplify the interface and remove redundancy in the code both input word lattices and results hypotheses share a single HAPI object. However, this sharing is not appropriate for all functions and care must be taken to ensure that the underlying lattice will support the required operation. For example, the results lattices contain extra information (in the form of likelihoods for each word) that allows N-Best alternatives to be generated for portions of the utterance. A syntax network will not contain this information and so the N-Best functions will not work.

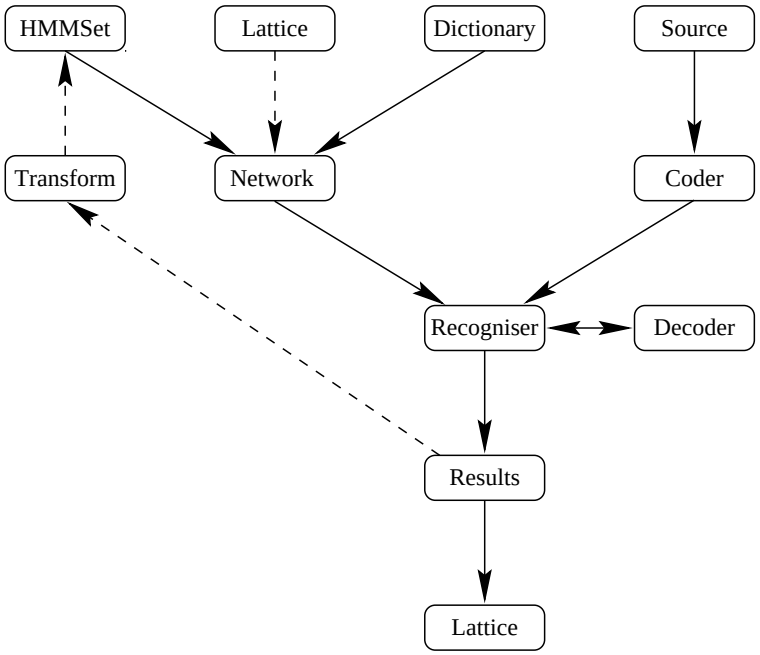

Fig. 2.1 HAPI Object Interaction

Most HAPI objects rely on Entropic's HTK libraries (available in the HTK Toolkit package), upon which HAPI is built, to provide the underlying structures and core functionality. For example the HMMSet object corresponds directly to an `HMMSet` as defined in HMODEL, and functions within HMODEL are used by HAPI to access the `HMMSet` and the individual HMMs. Often the core HTK functionality will have been supplemented with extra facilities that are useful for applications using speech recognition but which are not needed for the HTK tools. Taking the dictionary as an example; each **hapiDictObject** contains an HTK `Dict` structure (defined in `HDict.h`) and makes use of functions within the HDICT library module to load, save and modify the dictionary. However, in addition to these facilities provided within HTK, HAPI has introduced the concept of a phone set for each dictionary and added the ability to add and delete words/pronunciations to a pre-existing dictionary.

2.8 Operation Modes

A typical HAPI program will initially allocate and initialise the set of objects it needs to use. Often these will just be a single HMMSet, dictionary, network, source, coder and controlling recogniser but for some applications multiple networks, user dictionaries and multiple sources will be needed. Once the objects have been initialised (a process which includes loading the required information from disk and specifying the way in which the objects will be used and their interdependencies) the recogniser is ready for use. The current recognisers within HAPI operate in discrete utterance mode with results generated an utterance at a time. The boundaries of each utterance are defined by the coder with its decisions being controlled by either the application explicitly marking the start and end of each utterance or by a speech detector listening to the input and deciding when the user starts and stops talking.

To implement these methods HAPI implements two methods for source control.

- Directly

  The application explicitly calls **hapiSourceStart** and **hapiSourceStop** to begin and end each utterance. These calls will usually be made as a result of interaction with the user. For example the user may press the mouse button or a key to start the audio, begin speaking and when finished press a button or key to mark the end of the utterance.

- With speech detector.

  The application calls **hapiSourceStart** to indicate that the user may start speaking at any time and HAPI should start monitoring the audio input. Processing of the audio signal commences when the speech detector triggers and then continues until the speech detector decides that the user has stopped speaking.

Once the input mode has been chosen, recognition of each utterance can proceed using one of three modes of recogniser operation.

- Frame-by-frame

  The application can explicitly request that the decoder process a certain number of frames and then return control back to the application. This allows the application to maintain control of the processor and perform its own operations between calls to the recogniser.

- Synchronous callback

  The application surrenders control to the decoder until the utterance has been completed. The application can provide a callback function to enable results to be processed during recognition. This function can also be used to perform general application tasks but this is not recommended.

- *Asynchronous callback*

  In a multi-threaded environment the decoder will operate in its own thread (which also calls back the application with results) allowing the application to maintain control of the processor with the recogniser effectively running in the background. Currently only the callback function may use HAPI functions. The application will block until recognition has completed if it calls another HAPI function. In other words HAPI is single threaded but is thread safe.

2.9 HAPI Version history

Some of the features currently available in HAPI were not available in previous versions.

The following table lists the new features added for each version.

1.0 First HAPI library.

Asynchronous callback and confidence scores not included in released library.

1.1 Source driver added.

1.2 Transform object introduced.

More robust handling of errors.

Confidence scores implemented.

1.3 Automatic silence calibration facility.

Adaptation functions added to transform object.

Speaker adaptation tutorial included.

A new part describing both the standard HTK decoder[2] for development tasks and the professional LVX decoder for large vocabulary dictation type tasks.

1.4 Initial semantic tagging implementation.

Additional Microsoft WAV file support.

[2]An optimised version of the HTK decoder is available and is called the (professional) MVX decoder.

Chapter 3

Using HAPI: An Example Application

Possibly the easiest way to understand HAPI is to use it to produce an application. This chapter describes how HAPI can be used to develop a simple recognition-enabled application - a voice operated telephone dialing system. The first section of the tutorial will describe the basic recognition system and how to get this running with the initial program and the next section will describe the 'C' program itself. Chapter 4 describes some ways in which this application can be extended. Chapter 5 introduces speaker adaptation, and tests new speaker adapted models using the extensions of Chapter 4.

The initial system accepts a string of spoken digits and produces the recognised telephone number. The grammar includes a small number of filler words to allow for more natural spoken input but only accepts numbers said as a sequence of digits. For example if the user says

```
DIAL OH ONE TWO TWO THREE THREE TWO FOUR FIVE SIX ZERO
```

the recogniser should produce the required number (`01223324560`) as output. As this is meant as a demonstration of how to use HAPI rather than how to build user interfaces, the initial application will use the console to control input and to display the recogniser's output. The speech is read from the workstation's built in audio input.

The improvements described in the next chapter include extension of the recognition grammar to allow for more natural ways of expressing telephone numbers and the addition of a phone book of numbers that can be dialed by name. This extended system accepts the following input as being equivalent to the digit string above.

```
DIAL OH ONE DOUBLE TWO THREE THIRTY TWO FORTY FIVE SIXTY
CALL ENTROPIC FAX MACHINE
```

(assuming that the phone book had an appropriate entry for `ENTROPIC FAX MACHINE`).

Finally ways in which the user interface can be improved will be outlined, with sections covering the addition of the ability to alter and extend the phone book as well as to correct recognition errors by selecting alternatives.

Since this is a tutorial for HAPI rather than a real application it uses the console for user output rather than actually dialing the numbers. A real application would

be completely audio based with feedback to the user in the form of spoken prompts. Part III of this book covers some of the issues of producing a real application. This starts by outlining ways in which the tutorial dialer program can be modified to work in a hands free manner, controlled only by the spoken commands once the dialogue has been started.

3.1 Getting started

To get our first HAPI application running we need the following components

- A set of acoustic models (HMMs) for the required language and task.
- A dictionary mapping the required words into these models.
- A network specifying the recognition grammar.
- A configuration file containing the location of the above components.
- The application program itself (produced from the 'C' source code and HAPI).

The tutorial is available in both British (UK) English and American (US) English language versions. These can be found in the `Tutorial/UK` and `Tutorial/US` directories respectively. Within these directories various files can be found, such as the acoustic models, dictionaries, networks and configurations pertaining to each version. Unless otherwise stated, examples will be taken from the `US` directory.

Acoustic Models

This tutorial comes complete with a simple set of acoustic models (users of *grapHvite* may wish to use the more complex models that comes with that package). These are supplied in the form of a master model file (MMF) together with a `list` file, which lists the HMMs in the corresponding MMF. For the US version, the model file (`ecrl_us.mmf` together with the list file `ecrl_us.list`) contains a set of monophone models that can be used for both the extensible phone book, the grammar words and adaptation. The models were trained from high quality speech (16kHz sampling rate from a close talking head mounted microphone) and will work best from data recorded in similar circumstances. They will not function well with 8kHz telephone speech.

Dictionary

The dictionary for the basic US digit dialer (`basic_us.dct` shown below) is very simple with each word having a single phonetic spelling. The `PHONE_SET` line defines the phonetic set being used for the US version, and differs slightly from the UK version.

```
!PHONE-SET aa ae ah ao aw ax ay b ch d dh eh en er ey f g hh ih iy
jh k l m n ng ow oy p r s sh sil sp t th uh uw v w y z zh

!SIL            []    sil

CALL            []    k aa l sp
```

```
CONTACT         []    k aa n t ae k t sp
DIAL            []    d ay l sp
PHONE           []    f ow n sp
PLEASE          []    p l iy z sp
RING            []    r ih ng sp

ZERO            [0]   z iy r ow sp
OH              [0]   ow sp
ONE             [1]   w ah n sp
TWO             [2]   t uw sp
THREE           [3]   th r iy sp
FOUR            [4]   f ao r sp
FIVE            [5]   f ay v sp
SIX             [6]   s ih k s sp
SEVEN           [7]   s eh v ih n sp
EIGHT           [8]   ey t sp
NINE            [9]   n ay n sp
```

The pronunciation of each word is comprised of a sequence of phone models followed by a short pause model (`sp`). The `sp` model is effectively optional since it includes a path that passes through the model without processing any observations. This optional short pause greatly improves accuracy as people rarely speak without ever pausing between words.

As described later the output symbols are used to generate the recognised telephone number. These are blank for filler words (which provide no additional meaning to the output) and contain the appropriate digit for each number.

Network

Similarly the initial word network is relatively simple, with a loop of the digits following a few filler words with non-optional silence at the beginning and end of sentence. This is shown graphically (as it would appear in the *grapHvite* netbuilder) in figure 3.1.

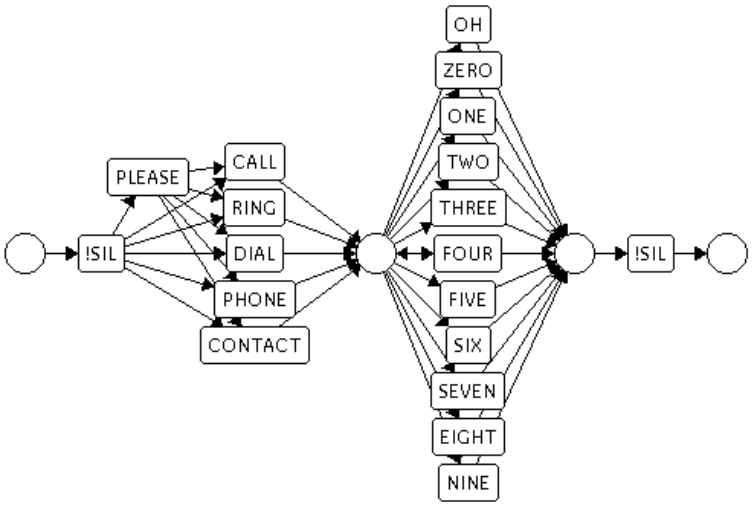

Fig. 3.1 Initial Network

The actual network is stored in a text file in HTK lattice format. The HTK book describes how to generate a network file from a set of grammar rules. Alternatively, the *grapHvite* netbuilder can be used to generate networks interactively.

Configuration

Every application using HAPI needs a configuration file. This specifies to HAPI where to find the resources required by the application and also configures the HTK libraries for the particular models and operation mode of the recogniser.

The configuration for the basic US digit dialer is as follows, and can also be found in the file `basic_us.cfg`.

```
# File locations
HAPI:   DICTFILE      = ./basic_us.dct
HAPI:   NETFILE       = ./basic.lat
HAPI:   HMMLIST       = ./ecrl_us.list
HAPI:   MMF           = ./ecrl_us.mmf

# Set up input
HParm:  MICIN         = F
HParm:  LINEIN        = T

# Set up HAPI pruning levels etc
HAPI:   LMSCALE       = 0.0
HAPI:   WORDPEN       = -20.0
HAPI:   GENBEAM       = 150.0

# Now set up for coding
HParm:  TARGETKIND    = MFCC_0_D_A_Z
HParm:  SOURCEKIND    = WAVEFORM
HParm:  SOURCERATE    = 625
HParm:  TARGETRATE    = 100000
HParm:  WINDOWSIZE    = 250000
HParm:  ZMEANSOURCE   = TRUE
HParm:  USEHAMMING    = TRUE
HParm:  PREEMCOEF     = 0.97
HParm:  NUMCHANS      = 24
HParm:  CEPLIFTER     = 22
HParm:  NUMCEPS       = 12
HParm:  CEPMEANTC     = 0.995
HParm:  CEPMEANWAIT   = 160
```

The first part of the configuration file just indicates to the library where it can find the HMMs, dictionary and word level network. The next part changes some of the default recognition parameters to values that are better suited to this particular application.

Finally the remainder of the configuration file contains the HTK parameters that control the way in which the audio waveform is encoded into observations. This is dependent upon the way in which the HMMs were trained and this part of the configuration file should be supplied by the provider of the models and used unaltered.

Compilation

The tutorial comes as part of the HAPI package. This package will normally be installed into a single directory hierarchy with the actual 'C' HAPI library found in the `HAPI/lib` directory. The library consists of two parts,

- HAPI.h

  A 'C' source code file defining the types and function prototypes that are used by HAPI.

- A platform dependent library of object code which contains the HAPI functions themselves. Under `UNIX` this is an archive file, while under Windows NT/95 this is a Microsoft Visual C++^{TM} (MSVC) lib/dll pair. Note that under Windows NT/95 the location of the runtime library (`HAPI_htk.dll`) should be in the execution path. The library also comes in three different versions, dependent on the Entropic product that HAPI was shipped with.

Product	Unix Library Name	PC MSVC Library Name
HTK	HAPI_htk.$CPU.a	HAPI_htk.lib
grapHvite	HAPI_mvx.$CPU.a	HAPI_mvx.lib
TRANSCRIBER	HAPI_lvx.$CPU.a	HAPI_lvx.lib

The tutorial (found in the `HAPI/Tutorial` directory) works on all supported workstations. However, as machines and operating systems differ there is a set of instructions for compiling HAPI programs for each supported architecture supplied as part of the HAPI installation. Machine dependent files are identified by the presence of a `$CPU` tag on the filename. The top level `README.txt` file includes a list of supported architectures for each installation and the table below lists some of the possible settings for `$CPU`. Note that the `README.txt` file should be referred to for the most up to date supported architecture information.

Machine	OS Version	CPU setting
Sun Sparc	Solaris 2.5-2.6	sun4_solaris
Hewlett-Packard PA-RISC	HPUX 9.05-10.20	hp700
Silicon Graphics MIPS	IRIX 5.3-6.5	sgi
Intel Pentium	Linux 2.0.x	linux (or linux_glibc)
Intel Pentium	Windows NT/95	win32
DEC Alpha	Digital UNIX 4.0	decosf_alpha

For machines using a version of the `UNIX` operating system the compilation instructions are basically the same for each platform and are summarised below. The only significant differences between the `UNIX` platforms are due to differences in the 'C' compiler flags and system libraries needed.

The first stage of compilation is to set up the environment. For example on a `linux` machine a `csh` or `tcsh` user should type the following,

```
> setenv HAPIDIR /usr/local/HAPI
> setenv HAPIBIN ~/bin
> cd $HAPIDIR
> source $HAPIDIR/env/env.linux
```

or for the `sh`, `ksh` or `bash` user,

```
> set HAPIDIR=/usr/local/HAPI; export HAPIDIR
> set HAPIBIN=~/bin; export HAPIBIN
> cd $HAPIDIR
> . $HAPIDIR/env/exp.linux
```

The environment variables explicitly set are used in the `Makefile` supplied with the tutorial and specify where HAPI is installed (`HAPIDIR`) and where the executable programs should be kept (`HAPIBIN`). Some values (such as the location of HAPI) are dependent upon each installation and must be set individually. However, the settings for the 'C' compiler are only dependent upon the machine architecture and do not change between installations so they can be set in one go from a supplied environment file. The table below lists the most important of these environment variables (where `script` refers to either `env` or `exp`). Note that the supplied `Makefile` can serve as a template for new applications although the only vital parts are the flags which ensure that the compiler can find the HAPI header file and library. These are the only changes that are required to include HAPI within an application, and so modifying an existing project should be a trivial task.

Variable	Set by	Usage
`CPU`	`script.$CPU`	Machine/architecture identifier
`HAPIDIR`	user	HAPI home directory
`HAPIBIN`	user	Top level binary directory containing `bin.$CPU`
`HAPICC`	`script.$CPU`	Name of program to use as 'C' compiler
`HAPICF`	`script.$CPU`	'C' flags needed to compile HAPI applications
`HAPILF`	`script.$CPU`	Linker flags needed to compile HAPI applications

Having set up the environment correctly and created the directory `$HAPIBIN/bin.$CPU` the program is compiled using `make`.

```
> cd $HAPIDIR/Tutorial
> make basic
# $HAPICC $HAPICF -I$HAPIDIR/lib  -c basic.c -o basic.o
gcc  -ansi -g -O  -I/usr/local/HAPI/lib  -c basic.c -o basic.o
# $HAPICC basic.o $HAPIDIR/lib/HAPI_htk.$CPU.a -lm  $HAPILF
gcc  basic.o  /usr/local/HAPI/lib/HAPI_htk.linux.a -lm  -laudio
# mv a.out $HAPIBIN/bin.$CPU/basic
mv a.out ~/bin/bin.linux/basic
```

Under `Windows NT/95` the developer has the choice of compiling the tutorial using either a supplied batch file and makefile (using `nmake`), or by using the MSVC workspace called Tutorial, which contains the projects `basic`, `final`, `HAPIAdapt` and `HAPIVite`.

Running the tutorial

Following successful compilation of the basic program the recogniser can be tested. The program is set up to use the built-in audio input from the user's workstation so it is essential that this is working correctly. This is often easiest setup and tested with the aid of audio controls that come with the workstation but can be accomplished just using the dialer.

The default configuration used with the dialer is set up to read audio from the machine's line-in socket. This is the input that should be used when a preamplifier

is used in conjunction with the microphone. In general this gives better quality than the amplifier in the workstation and if available is preferred.

```
 HParm:   MICIN = F
 HParm:   LINEIN = T
```

However if it is necessary to use a microphone in conjunction with the machine's built in amplifier, it is necessary to alter the configuration as follows.

```
 HParm:   MICIN = T
 HParm:   LINEIN = F
```

Note that not all machines have a simple choice between microphone and line level input sockets and in these cases the audio control panel (or similar control) must be used to select the input source. For example, under Linux a mixer program must be used to select input levels for all recording sources (including both line and microphone inputs).

Once the user has connected the microphone (and possibly amplifier) to the computer the program can be started. When used with input from direct audio, the program takes a single argument which is the name of the configuration file to load. The basic application begins by loading the models, dictionary and network and then tries to calibrate the speech detector.

```
> set path = ( $path $HAPIBIN/bin.$CPU )
> basic basic_us.cfg
 Loading HMMs, dictionary and network
 Initialisation complete

 Recogniser calibration:

  If the mean speech level isn't 20-40dB higher than silence
   level please check the audio system and try again

Press return to measure speech levels
```

The program waits for the user to press return before measuring the input level. Although it prompts the user to say a particular sentence any utterance containing a small amount of silence can be used (including remaining silent although this means that the application cannot estimate the average speech level).

```
 Now say 'She sells sea shells by the sea shore'
[Speak for a few seconds]
 Thank you

  Mean speech level in dBs 63.96 (32.4%), silence 27.40

Press return to process utterance
```

Once the input level has been measured, estimates for the mean speech energy, peak-to-peak signal and background noise energy levels are printed. The three levels should be checked for the following features

- The speech signal to background noise ratio (SNR in the case above equal to $63.96 - 27.40 = 36.56dB$) should be checked to ensure that the microphone is working. Normally a signal to noise ratio of 20-40dB would be expected with higher values expected from better quality microphones. However, if there is a problem the speech and silence levels will differ by only a few dB and the calibration procedure will decide that no estimate of the speech level could be found (indicated by a speech level of -1.0).

  For example with no power connected to the microphone preamplifier the following levels were obtained.

  ```
  Mean speech level in dBs -1.00 (0.5%), silence 8.54
  ```

  Here no estimate of the speech level was obtained because there was no variation of the level over time and the calibration assumed that there was no speech present in the input signal.

- The peak to peak input range should be significantly less than 100% otherwise clipping will occur and the signal will be heavily distorted.

  With the preamplifier set to its highest gain the speech signal to background noise ratio may not be any lower but because the input is clipped, distortion could significantly degrade accuracy.

  ```
  Mean speech level in dBs 87.75 (100.0%), silence 48.31
  ```

If either of these situations occurs the problem should be investigated and the program restarted once audio input is working as expected. It should be noted that when the user does not speak after the prompt the first of these conditions will occur despite the fact that the system is set up correctly. For example with the microphone and preamplifier set up correctly but with no speech after the prompt the following levels were obtained.

```
Mean speech level in dBs -1.00 (0.7%), silence 25.55
```

Once satisfactory levels have been obtained (restarting the program if necessary) the user can try to recognise a few utterances. The program is configured to use the speech detector and so the user only has to press return for the recogniser to start listening. Once the user starts speaking, the recogniser should start processing the input data and when they stop the recogniser should finish automatically. The output generated for a single input utterance is illustrated by the following:

```
@50 !SIL
@100 !SIL DIAL
@150 !SIL DIAL OH ONE
@200 !SIL DIAL OH ONE TWO THREE
@250 !SIL DIAL OH ONE TWO THREE FOUR
@300 !SIL DIAL OH ONE TWO THREE FOUR NINE EIGHT
@350 !SIL DIAL OH ONE TWO THREE FOUR NINE EIGHT SEVEN
@400 !SIL DIAL OH ONE TWO THREE FOUR NINE EIGHT SEVEN SIX
@450 !SIL DIAL OH ONE TWO THREE FOUR NINE EIGHT SEVEN SIX FIVE FOUR

You said: DIAL OH ONE TWO THREE FOUR NINE EIGHT SEVEN SIX FIVE FOUR
 Dialing: 01234987654
```

During recognition the program shows the most likely hypothesis after every 50 frames (0.5s of speech) have been processed and once the end of the utterance has been reached the string of recognised words and the equivalent phone number are printed.

3.2 Program Description

The description of the initial program is split into several sections each corresponding to a section in the program itself.

- Initialisation

  The first part of the description deals with loading and initialising HAPI and the example recogniser as well as freeing up the recogniser components at the end of the program.

- Results processing

  This section deals with processing of results. Intermediate results are supplied in the form of a trace string displayed periodically. The final results are produced by scanning the results from the recogniser to produce the string of recognised words and the equivalent telephone number.

- User Interaction

  These functions deal with all user control actions. User interaction is limited to starting and stopping audio input, speaking the utterance and calibrating the speech/silence detector when this is used.

- Control

  The 'C' "main" function is where top level control of the program is defined. It ensures that all the various tasks proceed in the correct order, initialise the recogniser, calibrate the speech detector (when necessary) and then processes user commands.

As with most 'C' programs the program begins by including any necessary definitions which for HAPI are contained in a single file `HAPI.h` header file.

```
/* Standard include files */
#include <stdio.h>
#include <stdlib.h>
/* And the HAPI header */
#include "HAPI.h"
```

After this it is common to include any type definitions local to the application. To maintain a modular and self contained style, references to all the HAPI objects and other information needed by the application are contained in a single structure.

```
typedef struct ahi {
   /* These are set up by the application */
   char *config;               /* Configuration file name */
   hapiRecType type;           /* Recogniser type */
   hapiRecMode mode;           /* Recogniser mode */
   hapiSourceType src;         /* Source of input data */
   /* These HAPI Objects are available to the application */
```

```
   hapiHMMSetObject hmms;      /* HMMSet - audio models */
   hapiDictObject dict;        /* Dictionary */
   hapiNetObject net;          /* Recognition network */
   hapiSourceObject source;    /* Input data source */
   hapiCoderObject coder;      /* Coder converts waveform to observations */
   hapiResObject inter,final;  /* Recognition results */
   hapiRecObject rec;          /* Decoder */
   /* And recognition makes use of the following functions and storage */
   int (*finalFunc)(struct ahi *ahi);
   void (*traceFunc)(char *string);
   int abort;                  /* Set to non-zero for application to abort */
   int doneFinal;              /* Set non-zero to indicate recogniser done */
   char dialString[256];       /* The string to dial */
}
ApplicationHAPIInfo;
```

These declarations are followed by sections of code for initialisation, results processing, user interaction and finally the main recognition and control functions.

3.3 Initialisation

The first task of any application wishing to make use of HAPI is to initialise the library by having it read the appropriate configuration file. This is done with a single call to **hapiInitHAPI** passing the name of the configuration file to be read as the first argument and a pointer to the function to be called in case of a fatal error within HAPI as the second. Either argument can be `NULL`, in which case defaults are used. The configuration file name defaults to `"config"` and by default no application function is called in the case of a fatal error.

The configuration file will be read into memory (ready for parameters to be overridden) and HAPI's internal structures initialised before control is returned to the application.

As mentioned earlier full error recovery is not yet implemented within HAPI and many functions can fail within HTK producing unrecoverable errors. When this occurs control can be transfered back to the application but no more HAPI calls are permitted and often (certainly for the tutorial dialer) the application has no choice but to terminate as gracefully as possible.

The tutorial application just sends a message to stderr and exits.

```
void FatalHAPIError(int error)
{
   fprintf(stderr,"A fatal HAPI error has caused the application to quit\n");
   exit(error);
}
```

Once HAPI has been initialised the next task is normally to create and initialise the set of objects that the application requires. For simple static tasks one HMMSet, one dictionary, one network, one source and one coder are required by the recogniser. For more complex tasks, multiple networks and possibly multiple HMMSets and dictionaries can be required. For the initial application, the simple static setup is all that is required.

The example code below (a fragment of code taken directly from the tutorial application) sets up a single object of each type together with two results objects

to allow separate treatment of intermediate (part way through utterance) and final (whole utterance) results. Note how the return value of each function is checked to ensure that it worked as intended. When an error occurs, `fail` is set and because of the way in which 'C' short circuits the logical OR operation (||) no further HAPI calls are made. Also note that there is no attempt to retry operations and the application just exits if a problem occurs. Real applications should probably try to be more robust and cope with some types of errors and fail more gracefully with others.

```
ApplicationHAPIInfo *InitialiseRecogniser(char *config, hapiSourceType src,
                                          hapiRecType type,hapiRecMode mode)
{
   ApplicationHAPIInfo *ahi;
   int fail=0;

   /* Allocate the all encompassing info structure */
   if ((ahi=(void*)malloc(sizeof(ApplicationHAPIInfo)))==NULL) {
      fprintf(stderr,"Memory allocation problem");
      exit(1);
   }

   /* Set up input parameters */
   ahi->config = config;
   ahi->type = type;
   ahi->mode = mode;
   ahi->src = src;

   ahi->finalFunc=NULL; ahi->traceFunc=NULL;
   ahi->abort=0; ahi->doneFinal=0; ahi->dialString[0]='\0';

   /* Initialise everything */
   if (hapiInitHAPI(config,FatalHAPIError)!=0) return(NULL);

   /* Creation functions are quick */
   fail=fail || ((ahi->hmms=hapiCreateHMMSetObject())==NULL);
   fail=fail || ((ahi->dict=hapiCreateDictObject())==NULL);
   fail=fail || ((ahi->source=hapiCreateSourceObject(src))==NULL);
   fail=fail || ((ahi->coder=hapiCreateCoderObject(ahi->source))==NULL);
   fail=fail || ((ahi->rec=hapiCreateRecObject(ahi->type,HAPI_RM_callback_sync,
                                               ahi->hmms,ahi->net))==NULL);
   fail=fail || ((ahi->inter=hapiCreateResObject(ahi->rec,
                                                 HAPI_RI_def_htk))==NULL);
   fail=fail || ((ahi->final=hapiCreateResObject(ahi->rec,
                                                 HAPI_RI_def_htk))==NULL);

   /* Nice to be able to get from results objects to application data */
   fail=fail || (hapiResSetAttachment(ahi->inter,ahi)<0);
   fail=fail || (hapiResSetAttachment(ahi->final,ahi)<0);
```

These last two functions attach the application information to the result objects. This allows the results processing functions (which are only passed a reference to a results object) to obtain access to all the applications' HAPI objects.

Next (as part of the same function) comes the more costly (in terms of memory and time required) initialisation operations that actually load the necessary models, dictionaries and networks ready for recogniser operation. Since no override functions

are called, the application relies on the configuration file to define the location of HMMSet, dictionary and network files.

```
   /* Initialisation includes time consuming load */
   if (!fail) printf(" Loading HMMs, dictionary and network\n");
   fail=fail || (hapiHMMSetInitialise(ahi->hmms)<0);
   fail=fail || (hapiDictInitialise(ahi->dict)<0);
   fail=fail || ((ahi->net=hapiLoadNetObject(ahi->type,ahi->hmms,
                                             ahi->dict,NULL))==NULL);

   /* Initialise everything else */
   fail=fail || (hapiSourceInitialise(ahi->source)<0);
   fail=fail || (hapiCoderInitialise(ahi->coder,NULL)<0);
   fail=fail || (hapiRecInitialise(ahi->rec)<0);

   if (fail) {
      char8 buf[STR_LEN];

      fail=hapiCurrentStatus(buf);
      fprintf(stderr,"InitialiseRecogniser: %d, %s\n",fail,buf);
      free(ahi); ahi=NULL;
   }

   return(ahi);
}
```

Given that we have an initialisation function that creates lots of HAPI objects, there should also be a function to clean up at the end of the program, deleting all the allocated objects. Once again the return values of the functions are checked for errors although in this case the checks are less important as the program is about to exit. However, errors will highlight incorrect use of functions and attempts to delete objects in the wrong order (for example deleting an HMMSet when it is still being used by an active network).

```
int DeleteRecogniser(ApplicationHAPIInfo *ahi)
{
   int fail=0;

   fail=fail || (hapiDeleteResObject(ahi->final)<0);
   fail=fail || (hapiDeleteResObject(ahi->inter)<0);
   fail=fail || (hapiDeleteRecObject(ahi->rec)<0);
   fail=fail || (hapiDeleteCoderObject(ahi->coder)<0);
   fail=fail || (hapiDeleteSourceObject(ahi->source)<0);
   fail=fail || (hapiDeleteNetObject(ahi->net)<0);
   fail=fail || (hapiDeleteDictObject(ahi->dict)<0);
   fail=fail || (hapiDeleteHMMSetObject(ahi->hmms)<0);
   return(!fail);
}
```

3.4 User interaction

In the initial application user interaction is kept to a minimum. Apart from indicating that the recogniser should start processing the next utterance, the only interaction with the user is calibrating the speech detector (when it is used).

This calibration procedure will probably be the first time the user has interacted with the recogniser and it gives the user an opportunity to verify that the audio system is set up correctly.

There are two functions in HAPI for calibrating the speech detector. Both operate on a source object, when the source object is not a direct audio device or when the speech detector is not being used these functions do nothing. The functions differ in that one (**hapiSourceCalibrateSilence**) expects a few seconds of silence whilst the other (**hapiSourceCalibrateSpeech**) expects a mixture of speech and silence. Although using the second of these functions requires that the user say something, it does allow the average speech level to be measured. This allows the user to verify that audio system is functioning correctly and that the signal-to-noise ratio is high enough to ensure that the recogniser will operate accurately.

```
int CalibrateInput(ApplicationHAPIInfo *ahi, int wait)
{
   char buf[STR_LEN];
   int fail,needed;
   float levels[HAPI_CAL_size];

   fail = ((needed=hapiSourceCalibrateSpeech(ahi->source,NULL))<0);
   if (!fail && needed>0) {
      /* Now need to set up for interaction */
      /* This application doesn't have any fancy interface so its printf !! */
      printf(" Recogniser calibration:\n\n");
      printf("  If the mean speech level isn't 20-40dB higher than silence\n");
      printf("   level please check the audio system and try again\n\n");

      /* Now we will measure then display the speech levels */
      if (wait) {
         printf("Press return to measure speech levels\n");
         gets(buf);
      }
      printf(" Now say 'She sells sea shells by the sea shore'\n");
      fail = (hapiSourceCalibrateSpeech(ahi->source,levels)<0);
      printf(" Thank you\n\n");
      printf(" Mean speech level in dBs %.2f (%.1f%%), silence %.2f\n\n",
             levels[HAPI_CAL_speech],levels[HAPI_CAL_peak]*100.0,
             levels[HAPI_CAL_background]);
   }
   fail = ((needed=hapiSourceCalibrateSilence(ahi->source,NULL))<0);
   if (!fail && needed>0) {
      printf(" Silence measurement\n\n");

      /* Now we will measure then display the speech levels */
      if (wait) {
         printf("Press return to measure silence\n");
         gets(buf);
      }
      printf("Quiet please - measuring silence\n");
      fail = (hapiSourceCalibrateSilence(ahi->source,levels)<0);
      printf(" Thank you\n\n");
      printf(" Mean silence level in dBs %.2f\n\n",
             levels[HAPI_CAL_background]);
```

```
   }
   return(!fail);
}
```

Apart from this calibration procedure the only other user interaction is to start the recogniser listening for the start of speech. This is done in a similar manner in the `main` function using `gets` to wait for the user to press return.

3.5 Recognition

Prior to the recognition of each utterance the coder object must be prepared and the input source started. The utterance is then recognised by a single call to **hapiRecRecognise** in synchronous callback mode. This function call takes a pointer to the callback function **applicationInterfaceFunction** as a final argument. In this example, the callback function is responsible for retrieving and displaying both intermediate and final results. When the utterance recognition is complete we must tidy up by calling **hapiCoderComplete**.

In the tutorial application these steps are carried out by the **RecogniseWithTrace** function.

```
int RecogniseWithTrace(ApplicationHAPIInfo *ahi,
                       int (*finalFunc)(ApplicationHAPIInfo *ahi),
                       int traceInterval,void (*traceFunc)(char *string))
{
   int fail=0;
   /* Set up call back functions */
   ahi->finalFunc=finalFunc;
   ahi->traceFunc=traceFunc;

   /* Get the input channel ready */
   fail=fail || (hapiCoderPrepare(ahi->coder)<0);
   fail=fail || (hapiSourceStart(ahi->source)<0);

   /* Do the recognition - with trace and final callbacks */
   /*  Includes test for ahi->doneFinal so works in sync and async modes */
   ahi->doneFinal=0;
   fail=fail || (hapiRecRecognise(ahi->rec,ahi->net,ahi->coder,
                                  ahi->final,traceInterval,ahi->inter,
                                  applicationInterfaceFunction)<0);

   /* We have waited for final callback so can do this here rather than */
   /*  in final callback function or when it signals application that */
   /*  results ready */
   fail=fail || (hapiCoderComplete(ahi->coder)<0);
   return(!fail);
}
```

Note that audio input begins once `hapiSourceStart` is called but since `hapiCoderPrepare` takes very little time recognition effectively begins when `RecogniseWithTrace` is called.

3.6 Results Processing

Results processing can be split into two sections.

- Intermediate results.

  These are generated whilst recognition is proceeding. Often the only use of these results will be to let the user know that recognition is proceeding and maybe give an indication of the most likely hypothesis so far. In these cases (and in the case of our dialer) intermediate results processing will consist of a function that displays a trace string containing the current frame number and the most likely string of words up to that frame.

  In more complex applications more information may be needed, for instance the application may want to know when the user started and stopped speaking or what word the user is currently saying. These bits of information may not even be available directly from the results object but can be derived from other HAPI calls. For example one indication that the speech detector has triggered is that the first observation has been processed.

- Final results.

  Once the utterance has been processed and the final set of results has been generated, application processing of the results usually begins in earnest. The way in which the application processes the results is dependent upon the type of application. However in many simple cases the application will just need to know the string of words that were recognised. The dialer also needs to know the corresponding telephone number and it is easier to make use of some of HAPI's facilities than to parse the resulting string of words to generate the number.

For the dialer application the processing of the intermediate trace results is trivial, a trace string is displayed on `stdout` to indicate progress to the user. Processing of the final results is more complicated as the output of the recogniser should consist of two things.

- The sequence of words spoken

  To show the user the string of words that were recognised, thus allowing them to determine if the recogniser is operating correctly and accurately.

- The telephone number

  This is the sequence of digits that would be used by a real application to dial the required number.

The phone number is closely related to the sequence of spoken words; the spoken words are in the same order as the telephone digits and each word always corresponds to a particular digit. This is not always the case, with clock times the ordering of hours and minutes is undefined and even the actual numbers mean different things dependent upon the rest of the sentence (`QUARTER TO ONE` and `TWELVE FORTY FIVE` mean the same thing despite having no words in common). In these cases much more care will have to be put into the design of the results processing functions.

However our basic dialer can make use of the word names to represent the spoken words and the output symbols to represent the corresponding digits. Apart from a few special cases (most of which do not crop up until the input syntax is extended in

section 4.1), the string of spoken words can be found by concatenating the HAPI word names and the digit string by concatenating the output symbols. The one exception to this is the words representing silence which should not be included in the string of spoken words. In the dialer words that should be excluded from the spoken word string are marked by having names starting with the '!' character.

This gives rise to the dictionary shown in section 3.1 with the words representing numbers having output symbols corresponding to the appropriate digit whilst all other words have empty output symbols.

The dialer application makes the distinction between intermediate and final results explicit by actually using two different results objects which are processed by two different functions. Since there is only a single callback function for each invocation of the recogniser an extra interface function is needed to choose whether intermediate or final results processing is required.

```
int32 applicationInterfaceFunction(hapiResObject res)
{
   int fail=0;
   ApplicationHAPIInfo *ahi;

   /* Lets get the application information */
   fail=fail || ((ahi=hapiResGetAttachment(res))==NULL);
   /* If these are the final set of results */
   if (ahi->final==res) {
      /* We call the application program to let it process them */
      if (ahi->finalFunc!=NULL)
         fail = fail || ahi->finalFunc(ahi);
      ahi->doneFinal=1;
   }
   else {
      /* Otherwise we just give it the opportunity to print trace info */
      char string[80];  /* String for trace info - frame number + word */
      fail=fail ||
         (hapiResTraceInfo(res,HAPI_TF_frame+HAPI_TF_word,80,string)<0);
      if (ahi->traceFunc!=NULL)
         ahi->traceFunc(string);
   }
   /* Important to tidy up after ourselves */
   fail = fail || (hapiResClear(res)<0);

   return(fail?-1:0);
}
```

A return value of 0 indicates to HAPI that recognition is proceeding okay and that recognition should continue and the next call to the callback function should come after the same number of frames. The callback function can return a negative number if an error occurs to indicate that this recognition should be immediately aborted. It could also return a positive number indicating that the function should be called back after a different number of frames have been processed.

Intermediate results take the form of the trace string that is provided by the interface function and displayed to the user.

```
void printTraceResults(char *string) {
   printf("%s\n",string);
}
```

As explained above the final results from the recogniser consists of two items; the word sequence and the telephone number. Both of these pieces of information are found by calling **GetResultsString** described below and then displayed on `stdout`.

```
int printFinalResults(ApplicationHAPIInfo *ahi)
{
   int fail;
   char words[2048],number[2048];

   fail = GetResultStrings(ahi,ahi->final,words,number);
   if (!fail) {
      strcpy(ahi->dialString,number);
      printf("You said: %s\n Dialing: %s \n",words,number);fflush(stdout);
   }
   return(fail);
}
```

The following **GetResultStrings** function is used to extract the required two strings (`words` and `number`) from the HAPI results object.

```
int GetResultStrings(ApplicationHAPIInfo *ahi,hapiResObject res,
                     char *words, char *number)
{
   int fail=0,i,n,ok;
   char buffer[256],*ptr;

   /* Need to know how many words in the result */
   fail=fail || ((n=hapiResAllWordNames(res,NULL))<0);
   words[0]=0;number[0]=0;
   for(i=1;!fail && i<=n;i++) {
      /* Read name for each word in turn */
      ok=(hapiResWordName(res,i,buffer)>0);
      /*  and append to word string with a space between words and */
      /*  ignoring special words (starting with !) */
      if (ok && buffer[0]!='!') {
         if (words[0]!=0) strcat(words," ");
         strcat(words,buffer);
      }
      /* Read output symbol for each word in turn */
      ok=(hapiResOutSym(res,i,buffer)>0);
      /*  and append to number string (no need for spaces) */
      if (ok) strcat(number,buffer);
   }
   return(fail);
}
```

3.7 Program Control

The controlling `main` function is very simple and essentially consists of a single call to each of the functions which have been defined and described in the previous subsections. As described earlier the program is run from the command line with one required argument (the name of the configuration file to read) and optional arguments (which are the names of files to be processed when direct audio is not used).

```
main(int argc,char **argv)
{
   ApplicationHAPIInfo *ahi;   /* Everything we know is here */
   hapiRecType type;           /* Recogniser type */
   hapiRecMode mode;           /* Recogniser mode */
   hapiSourceType src;         /* Source type haudio/script */
   int wait,n;                 /* Wait for user */
   char cmd[STR_LEN];

   /* Command line can be used to switch to AUDIO or no wait mode */
   src=HAPI_SRC_haudio_with_det;
   type=HAPI_RT_htk;
   mode=HAPI_RM_callback_sync;
   wait=1; n=2;

    if (argc<2) {
      fprintf(stderr,"Usage: %s config [files to process if not audio]\n",
              argv[0]);
      exit(1);
   }
   if (argc>2) src=HAPI_SRC_file,wait=0;

   if ((ahi=InitialiseRecogniser(argv[1],src,type,mode))==NULL)
      exit(1);

  CalibrateInput(ahi,wait);

  do {
      if (src==HAPI_SRC_file) {
         if (n>=argc) break;
         hapiSourceOverrideFilename(ahi->source,argv[n]);
         n++;
      }
      else if (hapiSourceAvailable(ahi->source)==0)
         break;
      if (wait) {
         printf("Press return to process utterance\n");
         gets(cmd);
      }

      if (!RecogniseWithTrace(ahi,printFinalResults,50,printTraceResults)) {
         char8 buf[STR_LEN];
         int fail;

         fail=hapiCurrentStatus(buf);
         if (fail==0)
            fprintf(stderr,"User function aborted recognition\n");
         else {
            fprintf(stderr,"RecogniseWithTrace: %d, %s\n",fail,buf);
            break;
         }
      }
   }
   while(ahi->abort==0);
```

```
   if (!DeleteRecogniser(ahi)) {
      char8 buf[STR_LEN];
      int fail;

      fail=hapiCurrentStatus(buf);
      fprintf(stderr,"DeleteRecogniser: %d, %s\n",fail,buf);
   }

   exit(0);
}
```

Chapter 4

Using HAPI: Improving the Dialer

The previous chapter has described the development of a simple phone dialer application. Whilst this example has served well as a tutorial in the basic process of application development using HAPI it needs to be extended and improved in a number of key areas if it is to be used as a real application. The grammar is fairly restrictive in the form of telephone number input which is accepted. Presently only sentences consisting of a filler word followed by a sequence of digits is accepted. This chapter starts by describing the expansion of the input syntax to allow more natural input, including the use of the words *DOUBLE*, *TREBLE* and the ability to dial by name. The dial by name facility requires that the user has a personal phone book of names and numbers. The chapter continues with functions that allow the user's phone book to be modified and extended. Finally it describes how the application can be further extended to provide the user with alternative hypotheses for mis-recognised utterances to allow their correction.

4.1 Expanded Input Syntax

The first changes to the program will be to expand the input syntax to allow more natural input including dialing by name[1].

Five specific improvements will be included

- Allow and produce correct output for words *DOUBLE* or *TREBLE*.
- Allow for phone numbers including the words *HUNDRED* or *THOUSAND*.
- Allow the use of numbers between *TEN* and *NINETY NINE*.
- Allow names to be used for common numbers.
- Add an access code so that no extra digits are required for outside lines.

[1] From HAPI's point of view networks are just one of the resources needed for a recognition system. They can be generated using the *grapHvite* netbuilder or HTK and this book does not describe how they are produced.

The original top level network is modified to contain two sub-lattices (see figure 4.1) in place of the simple digit loop. One of these holds an expanded digit dialing network (`number`) whilst the other holds the network containing the names included in the phone book (`phone_book`).

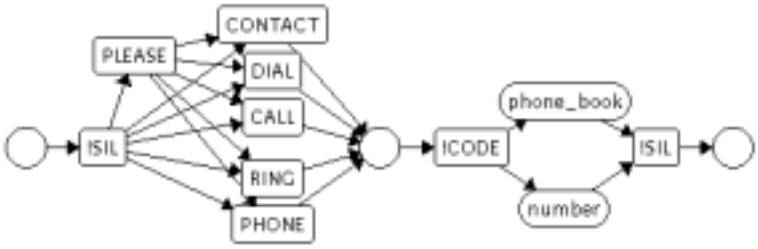

Fig. 4.1 The final top level network

In the final network the `phone_book` sub-lattice holds the system default phone book. This contains only two entries as it is intended to be supplemented by the user's personal phone book.

Fig. 4.2 The system default phone book

The user's phone book is stored in the form of a HAPI dictionary. Each word in the dictionary takes its name by concatenating the words making up the phrase identifying the telephone number. The output symbol is the actual telephone number and the phone sequence is the pronunciation of the whole phrase. This pronunciation is in terms of phone models and so the user's phone book can be expanded to include any arbitrary name. Note that the optional short pause model (`sp`) should be placed at the end of each spoken word. For example

```
ENTROPIC        [01223302651]  eh n t r aa p ih k sp
ENTROPIC_FAX    [01223324560]  eh n t r aa p ih k sp f ae k s sp
```

Entries for the words in this system default phone book are held in the final dictionary. They are as follows

```
EMERGENCY_SERVICES   [999]  ih m er jh ax n s iy sp s er v ih s ih z sp
DIRECTORY_ENQUIRIES  [192]  d ih r eh k t ax r iy sp ih n k w ay r iy z sp
```

Note that the changes to the dictionary and lattice mean that the configuration file needs to be updated. In order to avoid the hard coding of file names into the program, the name of the user's dictionary is entered as a HAPI configuration variable which although it is not used by HAPI itself can actually be read (and even overridden) by the application.

```
# File locations
HAPI:   PHONEBOOK     = ./phone_book_us.dct
HAPI:   DICTFILE      = ./final_us.dct
HAPI:   NETFILE       = ./final.lat
HAPI:   HMMLIST       = ./ecrl_us.list
HAPI:   MMF           = ./ecrl_us.mmf
```

The example program implements user phone books in the following manner. During initialisation the name of the file containing the user's phone book is obtained from the configuration file and this is read in as a dictionary. The top level lattice is then scanned to identify the sub lattice representing the phone book. The user dictionary is then 'inserted' into the initial system default phone book lattice by adding a node for each word in the phone book dictionary with corresponding links from the start node and to the end node. A reference to this lattice is included in the application information to allow later modification and addition to the phone book. Finally the recognition network is built from the top level lattice.

To implement this the `ahi` structure has been expanded and the `InitialiseRecogniser` function has been updated as follows:

```
typedef struct ahi {
.
.
+  hapiLatObject lat;          /* Word level lattice of recognition network */
+  hapiLatObject book;         /* Lattice holding phone book */
+  char8 *userName;            /* File name of user dictionary */
+  hapiDictObject user;        /* User Phone book as dictionary */
.
.
}
ApplicationHAPIInfo;

ApplicationHAPIInfo *InitialiseRecogniser(char *config, hapiSourceType src,
                                          hapiRecType type,hapiRecMode mode)
{
   ApplicationHAPIInfo *ahi;
   int fail=0;
+  hapiDictObject user;
+  char *sub,buffer[256],*ptr;
+  int i,nn,na;
.
.
.
   fail=fail || ((ahi->rec=hapiCreateRecObject(ahi->type,HAPI_RM_callback_sync,
                                               ahi->hmms,ahi->net))==NULL);
+  ahi->user=NULL;
+  if ((ptr=hapiOverrideConfStr("PHONEBOOK",NULL))==NULL)
+     user=NULL;
+  else {
+     fail=fail || ((user=hapiCreateDictObject())==NULL);
+     hapiDictOverrideFileName(user,ptr);
+  }
   fail=fail || ((ahi->inter=hapiCreateResObject(ahi->rec,
                                                 HAPI_RI_def_htk))==NULL);
```

```
.
.
.
   fail=fail || (hapiDictInitialise(ahi->dict)<0);
+  if (user!=NULL) {
+     fail=fail || (hapiDictInitialise(user)<0);
+     fail=fail || ((ahi->user=DuplicateDictionary(user,ahi->dict))==NULL);
+     fail=fail || (hapiDeleteDictObject(user)<0);
+  }
+  /* Network is also a lattice */
+  fail=fail || ((ptr=hapiOverrideConfStr("NETFILE",NULL))==NULL);
+  fail=fail || ((ahi->lat=hapiLoadLatObject(ahi->dict,ptr,
+                                            "DialLattice"))==NULL);
+  /* Scan to find phone_book lattice */
+  fail=fail || (hapiLatSize(ahi->lat,&nn,&na,NULL)<0);
+  for (i=1;!fail && i<=nn;i++) {
+     sub=hapiLatNodeSublatId(ahi->lat,i);
+     if (sub!=NULL) {
+        strcpy(buffer,sub);
+        for(ptr=buffer;*ptr;ptr++) *ptr=toupper(*ptr);
+     }
+     if (sub!=NULL && strcmp(buffer,"PHONE_BOOK")==0) {
+        ahi->book=hapiLatNodeSublatObject(ahi->lat,i,NULL);
+        break;
+     }
+  }
+  /* Remember that ahi->book and/or ahi->user could be NULL if not found */
+  if (ahi->book!=NULL && ahi->user!=NULL)
+     fail=fail || (AddPhoneBook(ahi->user,ahi->dict,ahi->book)<0);
+  else {
+     if (ahi->user!=NULL)
+        fail=fail || (hapiDeleteDictObject(ahi->user)<0);
+     ahi->user=NULL; ahi->book=NULL;
+  }
+
+  /* Now make the network */
!  fail=fail || (hapiNetBuild(ahi->net,ahi->lat)<0);

   /* Initialise everything else */
   fail=fail || (hapiSourceInitialise(ahi->source)<0);
.
.
.
}
```

This extra initialisation code includes two new functions. The first **DuplicateDictionary** is included to allow the user's phone book to omit the `!PHONE_SET` pseudo word (which should be included in proper HAPI dictionaries) and duplicates the dictionary entries using the proper phone set upon initialisation. This duplication function skips invalid words (which contain the wrong phones, word names or output symbols) thus ensuring that the erroneous entries in the user's phone book are just ignored rather than stopping the program working correctly.

```
hapiDictObject DuplicateDictionary(hapiDictObject user,hapiDictObject dict)
```

```
{
   hapiDictObject new;
   hapiPhoneId pron1[HAPI_MAX_PRON_LEN],pron2[HAPI_MAX_PRON_LEN];
   char *ptr,*out,buf[STR_LEN],buf1[STR_LEN]="<null>",buf2[STR_LEN];
   int i,j,n,p,phid,fail=0,okay;

   new=hapiEmptyDictObject(dict);
   fail=fail || ((n=hapiDictAllWordNames(user,NULL))<0);
   for (i=2;!fail && i<=n;i++) {
      /* Get word info */
      ptr=(hapiDictWordName(user,i,buf1)<0)?NULL:buf1;
      out=hapiDictPronOutSym(user,i,0,NULL);
      p=hapiDictWordPron(user,i,1,pron1);
      /* Transfer word to new dictionary */
      for (j=0,okay=1;okay && j<p;j++) {
         /* Need to munge phones */
         okay=okay && (hapiDictPhoneName(user,pron1[j],buf)>=0);
         okay=okay && ((phid=hapiDictPhoneId(new,buf))>0);
         pron2[j]=phid;
      }
      pron2[p]=0;
      /* Transfer access code from user dictionary to system */
      if (okay && ptr!=NULL && out!=NULL && strcmp(ptr,out)!=0 && p==0 &&
          (j=hapiDictWordId(dict,ptr))!=0 &&
          hapiDictPronOutSym(dict,j,0,NULL)==NULL)
         hapiDictPronOutSym(dict,j,0,out);
      else if (!okay || out==NULL || ptr==NULL || strcmp(ptr,out)==0 ||
               p<=0 || hapiDictWordId(dict,buf1)!=0) {
         fprintf(stderr,"Word %s skipped\n",buf1);
         continue;
      }
      fail=fail || (hapiDictAddItem(new,ptr,pron2,out)<0);
   }
   fail=fail || (hapiDictFinalise(new)<0);
   return(fail?NULL:new);
}
```

The second **AddPhoneBook** is used to insert the user's phone book into the syntax network. Once initialisation is complete, the `phone_book` lattice (see figure 4.2) is expanded by adding each word in the user's phone book dictionary between the start and end nodes of the lattice.

Note that this function assumes that the user dictionary and the main dictionary share a common phone set (because it uses the array of `hapiPhoneId` returned by **hapiDictWordPron** to directly in **hapiDictAddItem** rather than converting all the ids as **DuplicateDictionary** does). The initial duplication of the dictionary in **DuplicateDictionary** ensures that this is the case.

```
int AddPhoneBook(hapiDictObject user,hapiDictObject dict,hapiLatObject book)
{
   hapiPhoneId pron[HAPI_MAX_PRON_LEN];
   int32 nn,na,st,en,n,i,j,p,wid,nid,fail=0;
   char8 buf1[STR_LEN],buf2[STR_LEN],*out,*ptr;

   /* Find start and end nodes */
```

```
   fail=fail || (hapiLatModify(book)<0);
   fail=fail || (hapiDictModify(dict)<0);
   fail=fail || (hapiLatSize(book,&nn,&na,NULL)<0);
   for (i=1,st=en=-1;!fail && i<=nn;i++) {
      if (hapiLatNodePredArc(book,i,NULL)==0) st=i;
      if (hapiLatNodeFollArc(book,i,NULL)==0) en=i;
   }
   fail=fail || (st<0) || (en<0);
   fail=fail || ((n=hapiDictAllWordNames(user,NULL))<0);
   for (i=2;!fail && i<=n;i++) {
      /* Transfer word to main dictionary */
      fail=fail || (hapiDictWordName(user,i,buf1)<0);
      fail=fail || ((out=hapiDictPronOutSym(user,i,0,NULL))==NULL);
      fail=fail || ((p=hapiDictWordPron(user,i,1,pron))<0);
      /* Skip empty pronunciations */
      if (!fail && p==0) continue;
      fail=fail || (hapiDictAddItem(dict,buf1,pron,out)<0);
      /* Add node into phone_book lattice */
      fail=fail || ((wid = hapiDictWordId(dict,buf1))<0);
      fail=fail || ((nid = hapiLatAddNode(book,wid,0))<0);
      fail=fail || (hapiLatAddArc(book,st,nid)<0);
      fail=fail || (hapiLatAddArc(book,nid,en)<0);
   }
   fail=fail || (hapiDictFinalise(dict)<0);
   fail=fail || (hapiLatFinalise(book,st,en)<0);

   return(fail==0?0:-1);
}
```

As well as allowing new phone book entries to be added to the word level syntax, the way in which telephone numbers can be said has been expanded (see figure 4.3).

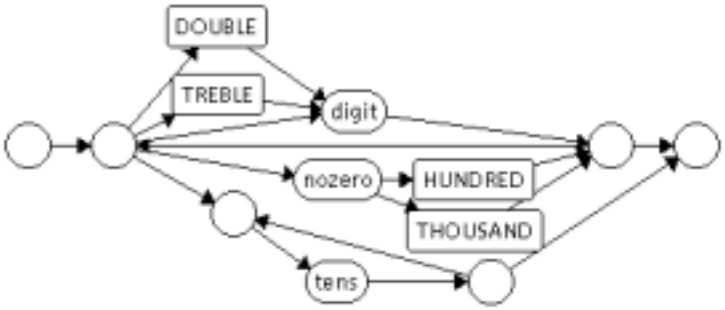

Fig. 4.3 The numbers network

The syntax has been expanded to allow the use of the words *DOUBLE* and *TREBLE*; a non-zero digit followed by the word *HUNDRED* or *THOUSAND* i.e. *TWO THOUSAND*; the numbers between 10 and 99 i.e. *TWENTY FIVE*.

The expanded input syntax requires extensions to the results processing function. With suitable network structure the output symbols could correspond directly to

the phone number. However this would require substantial duplication in both the dictionary and network. To avoid this two tricks will be used.

An additional symbol (`X`) will be used to indicate that the following digit should be repeated (so the sequence of output symbols `4XX321` becomes the telephone number `433321`). This makes handling of the words `DOUBLE` and `TREBLE` much simpler (they have output symbols of `X` and `XX` respectively) although the actual results processing function has to be modified to convert the `X`'s to the correct digit.

```
int GetResultStrings(ApplicationHAPIInfo *ahi,hapiResObject res,
                     char *words, char *number)
{
   int fail=0,i,n,ok;
   char buffer[256],*ptr;
+  int c=0,l;

   /* Need to know how many words in the result */
   fail=fail || ((n=hapiResAllWordNames(res,NULL))<0);
   words[0]=0;number[0]=0;
   for(i=1;!fail && i<=n;i++) {
      /* Read name for each word in turn */
      ok=(hapiResWordName(res,i,buffer)>0);
      /*  and append to word string with a space between words and */
      /*  ignoring special words (starting with !) */
      if (ok && buffer[0]!='!') {
         if (words[0]!=0) strcat(words," ");
         strcat(words,buffer);
      }
      /* Read output symbol for each word in turn */
      ok=(hapiResOutSym(res,i,buffer)>0);
+     /*  and append to number string (no need for spaces) */
+     ptr=buffer;
+     while(ok && *ptr!=0) {
+        c++;
+        if (ptr[0]!='X') { /* X repeats next number */
+           for (l=strlen(number);c>0;c--,l++)
+              number[l]=*ptr;
+           number[l]=0;
+           c=0;
+        }
+        ptr++;
+     }
   }
   return(fail);
}
```

Also two `NULL` words (words with an empty pronunciations) will be used. One will be used to insert an extra `0` into the telephone number in the case of strings such as `TWENTY`, `FIFTY` etc. This extra word allows a unified treatment for this part of the network since each two digit string can consist of the multiples of ten (twenty-ninety with output symbols `2-9`) followed by either a digit (between one and nine with output symbols `1-9`) or this special empty word (with output symbol `0`). This means if the user says "Twenty twenty five thirty" this should be recognised as the word sequence `TWENTY !0ZERO TWENTY FIVE THIRTY !0ZERO` which has the correct

sequence of output symbols 202530. Numbers between ten and nineteen can be handled by setting the output symbol to the appropriate two digit number. The other NULL word will be used to insert a user definable access code at the start of the phone number.

```
DOUBLE       [X]   d ah b ax l sp
TREBLE      [XX]   t r eh b ax l sp

HUNDRED      [00]  hh ah n d r ih d sp
THOUSAND    [000]  th aw z ih n d sp

TEN          [10]  t eh n sp
ELEVEN       [11]  ih l eh v ih n sp
TWELVE       [12]  t w eh l v sp
THIRTEEN     [13]  th er t iy n sp
FOURTEEN     [14]  f ao r t iy n sp
FIFTEEN      [15]  f ih f t iy n sp
SIXTEEN      [16]  s ih k s t iy n sp
SEVENTEEN    [17]  s eh v ih n t iy n sp
EIGHTEEN     [18]  ey t iy n sp
NINETEEN     [19]  n ay n t iy n sp
TWENTY       [2]   t w eh n t iy sp
THIRTY       [3]   th er t iy sp
FORTY        [4]   f ao r t iy sp
FIFTY        [5]   f ih f t iy sp
SIXTY        [6]   s ih k s t iy sp
SEVENTY      [7]   s eh v ih n t iy sp
EIGHTY       [8]   ey t iy sp
NINETY       [9]   n ay n t iy sp
!OZERO        [0]

!CODE                  []
EMERGENCY_SERVICES   [999]  ih m er jh ih n s iy sp s er v ih s ih z sp
DIRECTORY_ENQUIRIES  [192]  d ih r eh k t ax r iy sp ih n k w ay r iy z sp
```

The expanded dictionary resulting from these changes is shown above.

4.2 Modification of User Phone Book

Having implemented a phone book for the user it is very desirable to be able to add names and numbers to it whilst the program is running. As the phone book is a dictionary and there are functions within HAPI to add new words to a dictionary, this can be accomplished relatively easily.

The only real complication is that the main dictionary and lattice are dependent upon the user's phone book (they are altered when the user's phone book is added) and so it is necessary to modify these as well as the user's phone book. In practice it is easiest to undo the modifications made to the main network and dictionary, modify the user dictionary and rebuild the network rather than to try and modify all three in parallel.

This is accomplished by a function that removes from the phone book lattice and the main dictionary all items that occur in the user dictionary. Provided that the user dictionary is not modified between calls to **AddPhoneBook** and **DelPhoneBook** this function will undo all the additions by **AddPhoneBook** and leave the lattice and main dictionary in the same state as after loading.

```
int DelPhoneBook(hapiDictObject user,hapiDictObject dict,hapiLatObject book)
{
   int32 nn,na,i,wid,uid,fail=0;
   char8 buf1[STR_LEN],*ptr;

   /* Find start and end nodes */
   fail=fail || (hapiLatModify(book)<0);
   fail=fail || (hapiDictModify(dict)<0);
   fail=fail || (hapiLatSize(book,&nn,&na,NULL)<0);
   for (i=1;!fail && i<=nn;i++) {
      wid = hapiLatNodeWordId(book,i,NULL);
      if (wid>0) ptr=hapiDictWordName(dict,wid,buf1);
      uid=((wid>0 && ptr!=NULL)?hapiDictWordId(user,ptr):0);
      if (uid>1) {
         fail=fail || (hapiLatDeleteNode(book,i)<0);
         fail=fail || (hapiDictDeleteItem(dict,wid,0)<0);
      }
   }
   fail=fail || (hapiDictFinalise(dict)<0);
   fail=fail || (hapiLatFinalise(book,0,0)<0);

   return(fail==0?0:-1);
}
```

Every change to the user dictionary starts by deleting the current recognition network, undoing the changes to the phone book lattice and the main dictionary and marking the user dictionary as modifiable. Once the necessary changes have been made the dictionary is finalised, added to the main dictionary and phone book lattice before the recognition network is rebuilt.

```
   fail=fail || (hapiDeleteNetObject(ahi->net)<0) || (ahi->net=NULL);
   fail=fail || (DelPhoneBook(ahi->user,ahi->dict,ahi->book)<0);
   fail=fail || (hapiDictModify(ahi->user)<0);
   /* Can now make changes to user phone book dictionary */
.
.
.
   /* Once have made all necessary changes rebuild everything */
   fail=fail || (hapiDictFinalise(ahi->user)<0);
   fail=fail || (AddPhoneBook(ahi->user,ahi->dict,ahi->book)<0);
   fail=fail || ((ahi->net=hapiCreateNetObject(ahi->type,ahi->hmms,
                                               ahi->dict))==NULL);
   fail=fail || (hapiNetBuild(ahi->net,ahi->lat)<0);
```

Since the user enters the pronunciation as a string of phones delimited by white space it is necessary to convert this into the array of `hapiPhoneIds` required by the **hapiDictAddItem** call. This is done by parsing the string one phone at a time, checking that the phone is valid and copying its `hapiPhoneId` into the pronunciation array.

```
int MakePron(hapiDictObject user, hapiPhoneId *pron, char8 *phones)
{
   char *ptr,*nxt=NULL;
   int j,phid,okay=1;

   for (j=0,ptr=phones;okay && *ptr!=0;j++,ptr=nxt) {
      /* Find end of phone */
      for (nxt=ptr;*nxt!=0 && !isspace(*nxt);nxt++);
      /* And replace white space with 0 */
      while (isspace(*nxt)) *nxt++=0;
      okay=okay && ((phid=hapiDictPhoneId(user,ptr))>0);
      pron[j]=phid;
   }
   pron[j]=0;
   return(okay?0:-1);
}
```

Putting these operations together produces a function that interacts with the user to find out what modifications they require and then makes the necessary changes. Note that the access code is modified as a special case. Because it does not add or subtract anything from the dictionary it does not need to be wrapped as described above and is implemented by just altering the output symbol associated with the access code word `!CODE`.

```
int ModifyPhoneBook(ApplicationHAPIInfo *ahi)
{
   hapiPhoneId pron[HAPI_MAX_PRON_LEN];
   char buf1[STR_LEN],buf2[STR_LEN],buf3[STR_LEN],buf[STR_LEN],*ptr;
   int fail=0,wid,did,del;

   printf("Name ? : "); fflush(stdout);
   gets(buf1);
   for (ptr=buf1;*ptr!=0;ptr++) if (isspace(*ptr)) *ptr='_';
   if ((did=hapiDictWordId(ahi->dict,buf1))!=0 &&
       ((wid=hapiDictWordId(ahi->user,buf1))!=0) &&
       hapiDictWordPron(ahi->user,wid,1,NULL)==0) {
      printf("Access code ? : "); fflush(stdout);
      gets(buf2);
      /* Can chnage output symbol with altering network */
      hapiDictPronOutSym(ahi->dict,did,0,buf2);
      hapiDictPronOutSym(ahi->user,wid,0,buf2);
      return(0);
   }
   printf("Phone number [blank to delete] ? : "); fflush(stdout);
   gets(buf2);
   if (!isspace(buf2[0]) && buf2[0]!=0) {
      del=0;
      printf("Pronunciation ? : "); fflush(stdout);
      gets(buf3);
   }
   else del=1;
   /* Before we modify anything need to delete old stuff */
   fail=fail || (hapiDeleteNetObject(ahi->net)<0) || (ahi->net=NULL);
   fail=fail || (DelPhoneBook(ahi->user,ahi->dict,ahi->book)<0);
```

```
   fail=fail || (hapiDictModify(ahi->user)<0);
   /* Can now make changes to user phone book dictionary */
   if ((wid=hapiDictWordId(ahi->user,buf1))!=0) {
      if (del) printf("Word %s already in dictionary; Delete ? : ");
      else printf("Word %s already in dictionary; Replace ? : ");
      fflush(stdout);
      gets(buf);
      if (toupper(buf[0])=='Y')
         fail=fail || (hapiDictDeleteItem(ahi->user,wid,0)<0);
   }
   if (!fail && !del) {
      if (hapiDictWordId(ahi->dict,buf1)!=0 ||
          MakePron(ahi->user,pron,buf3)<0)
         fprintf(stderr,"Word %s could not be added\n",buf1);
      else
         fail=fail || (hapiDictAddItem(ahi->user,buf1,pron,buf2)<0);
   }
   /* Once have made all necessary changes rebuild everything */
   fail=fail || (hapiDictFinalise(ahi->user)<0);
   fail=fail || (AddPhoneBook(ahi->user,ahi->dict,ahi->book)<0);
   fail=fail || ((ahi->net=hapiCreateNetObject(ahi->type,ahi->hmms,
                                               ahi->dict))==NULL);
   fail=fail || (hapiNetBuild(ahi->net,ahi->lat)<0);
   return(fail?-1:0);
}
```

This function is called when the user indicates that they want to modify the phone book rather than recognise an utterance. The main function is modified to take notice of what the user types before pressing return when waiting for an utterance. This is then used as a command to control the next action.

```
.
.
      printf("Commands available\n");
      printf(" l - List entries in phone book\n");
      printf(" n - Add new word to phone book\n");
      printf(" s - Save phone book\n");
      printf(" q - Quit application\n");
      while (wait) {
         hapiPhoneId pron[HAPI_MAX_PRON_LEN];
         char *ptr,*num,buf[STR_LEN];
         int fail=0,e,p,i,j;

         printf("Type command or press return to process utterance\n");
         gets(cmd);
         switch(cmd[0]) {
          case 'n':
            if (ModifyPhoneBook(ahi)<0) {
               char8 buf[STR_LEN];
               int fail;

               fail=hapiCurrentStatus(buf);
               fprintf(stderr,"ModifyPhoneBook: %d, %s\n",fail,buf);
            }
```

```
            break;
          case 'l':
            if (ahi->user==NULL) break;
            if ((e=hapiDictAllWordNames(ahi->user,NULL))<2) break;
            printf(" Phone book has %d entries:\n",e-1);
            for (i=2;i<=e;i++) { /* Skip !NULL word */
               ptr=hapiDictWordName(ahi->user,i,buf);
               if (ptr==NULL) continue; /* Deleted */
               num=hapiDictPronOutSym(ahi->user,i,1,NULL);
               printf(" %-12s -> %-15s (",num,ptr);
               if ((p=hapiDictWordPron(ahi->user,i,1,pron))<0)
                  printf("<unknown>");
               else
                  for (j=0;j<=p;j++)
                     if (hapiDictPhoneName(ahi->user,pron[j],buf)>=0)
                        printf(" %s",buf);
               printf(" )\n");
            }
            printf("\n");
            break;
          case 's':
            if (ahi->user!=NULL && ahi->userName!=NULL)
               hapiDictSave(ahi->user,ahi->userName);
            break;
          case 'q':
            cmd[0]='q';
            break;
          default:
            cmd[0]=0;
         }
         if (cmd[0]==0 || cmd[0]=='q') break;
      }
      if (cmd[0]=='q') break;
.
.
```

4.3 Alternative Hypotheses

One of the more useful facilities provided within the HAPI library is the ability to generate alternative hypotheses for each utterance. This is accomplished in two stages. Firstly it is necessary to expand the recognition search to ensure that alternatives are generated, in the form of a lattice of word hypotheses. This is accomplished by using multiple **tokens** in each part of the network to represent different hypotheses. As usual the number of tokens is read in as a configuration parameter.

```
HAPI: NTOKS = 4
```

Once the output lattice has been recovered from the final recognition results, the alternatives can be found in several ways.

- Alternatives for a certain period of time of the utterance.
- Alternatives between two specified nodes.

- Alternatives to a particular path through the lattice.

  This is probably the most useful way of discovering multiple hypotheses or for simplifying the process of the user correcting the recogniser results. Three functions are available which find alternatives for a particular portion of a route through the lattice passed to the function.

  - Different words. The returned alternative routes will each consist of a different word sequence. Where multiple alternatives map to a single word sequence (for example if they contained different pronunciations of the same word) only the most likely is returned.
  - Different pronunciations. The returned alternative routes will each consist of a different pronunciation sequence. Only the most likely alternative mapping to a particular sequence of pronunciations will be returned.
  - Different output symbols. Where the output symbols represent the underlying result this function can be used to find alternatives.

For the dialer, the final of these functions can be used to present the user with alternative phone numbers in cases where the first choice is incorrect. By ensuring that the lattice of possibilities from the final results are available the user can be presented with a confirmation dialog in which they can either accept the first choice answer or one of a list of hypotheses.

Also a function has been added to play back the current utterance to the user (just to demonstrate how this is done) and then present the user with an opportunity to accept the recognised string, select an alternative, request more options or proceed to the next utterance.

```
int VerifyResult(ApplicationHAPIInfo *ahi,char *words,char *number,
                 hapiLatObject lat)
{
   hapiLatArcId **alt;
   char buf[STR_LEN],ans[NALT][STR_LEN];
   int i,j,k,n,m,ok,c,maxAlt;

   printf("\nRecognised words: %s\n\n",words); fflush(stdout);
   if (ahi->play!=NULL && ahi->nw>0 && ahi->wave!=NULL) {
      hapiSourceSamplePeriod(ahi->play,&ahi->sampPeriod);
      hapiSourcePlayWave(ahi->play,ahi->nw,ahi->wave);
   }
   maxAlt=NALT;
   do {
      if ((alt=malloc(sizeof(hapiLatArcId*)*maxAlt))==NULL) return(-1);
      n=hapiLatOutSymAlternatives(lat,0,-1,NULL,maxAlt,alt);
      m=n-maxAlt+NALT;
      for (i=0,k=maxAlt-NALT;k<n;i++,k++) {
         for(j=0,ans[i][0]=0;alt[k][j]!=0;j++) {
            ok=hapiLatArcOutSym(lat,alt[k][j],buf);
            if (ok>0) strcat(ans[i],buf);
         }
         for (j=strlen(ans[i])-1,c=0;j>=0;j--)
            if (ans[i][j]=='X') ans[i][j]=c;
            else c=ans[i][j];
      }
```

```
      free(alt);

      printf(" Is number %s correct ",number);
      for (i=1;i<m;i++)
         printf("\n  or %d for %s ",i+maxAlt-NALT,ans[i]);
      printf("  [y n # m] ? ");
      fflush(stdout);
      gets(buf);
      switch(buf[0]) {
       case '0': case '1': case '2': case '3': case '4':
       case '5': case '6': case '7': case '8': case '9':
       case 'y':
         /* Select one of the alternatives */
         i=atoi(buf);
         if (buf[0]=='y' || (i>maxAlt-NALT && i<maxAlt))
            strcpy(ahi->dialString,ans[i-maxAlt+NALT]);
         else
            ahi->dialString[0]=0;
         buf[0]='y'; break;
       case '\n':
       case '\0':
         /* Default action is 'y' first time then 'm' */
         if (n<=NALT) {
            strcpy(ahi->dialString,ans[0]);
            buf[0]='y';
            break;
         }
       case 'm':
         /* If we can get more alternatives get them */
         if (n==maxAlt) {
            maxAlt+=NALT; break;
         }
       case 'n':
         /* Otherwise give up */
         ahi->dialString[0]=0; buf[0]='y'; break;
      }
   }
   while(buf[0]!='y');
}
```

To make the waveform available to this function it must be stored before use of the coder is completed. The following fragment of code is added to **RecogniseWithTrace** to store the waveform for later playback immediately before the call to complete the utterance.

```
+  if ((ahi->nw=hapiCoderAccessWave(ahi->coder,1,-1,NULL))>0) {
+     ahi->wave=malloc(sizeof(short)*ahi->nw);
+     hapiCoderAccessWave(ahi->coder,1,-1,ahi->wave);
+     ahi->sampPeriod=hapiSourceSamplePeriod(ahi->source,NULL);
+  }
   fail=fail || (hapiCoderComplete(ahi->coder)<0);
```

The verification function is called immediately after recognition to allow the user to verify and possibly correct the recognised number. If the user rejects the suggested number an empty string is returned in `ahi->dialString` and no number is dialed (well printed to the screen).

```
      if (!RecogniseWithTrace(ahi,printFinalResults,50,printTraceResults)) {
.
.
      }
+     else if (ahi->ans!=NULL) {
+        VerifyResult(ahi,ahi->wordString,ahi->dialString,ahi->ans);
+        if (strlen(ahi->dialString)>0)
+           printf("\n Dialing: %s \n\n",ahi->dialString);fflush(stdout);
+        if (ahi->wave!=NULL) {
+           free(ahi->wave);
+           ahi->wave=NULL;ahi->nw=0;
+        }
+        if (ahi->ans!=NULL)
+           hapiDeleteLatObject(ahi->ans),ahi->ans=NULL;
+     }
   }
   while(ahi->abort==0);
.
.
```

4.4 Running the extended dialer

The final version of the program is supplied in `final.c` and can be compiled in the same way as the initial basic program. It is also invoked in the same way as the basic program but because of all the extra features, a different configuration file needs to be used.

```
> final final_us.cfg
 Loading HMMs, dictionary and network
 Initialisation complete

 Recogniser calibration:

  If the mean speech level isn't 20-40dB higher than silence
   level please check the audio system and try again

Press return to measure speech levels

 Now say 'She sells sea shells by the sea shore'
 Thank you

 Mean speech level in dBs 62.39 (34.0%), silence 29.70
```

The initialisation and calibration looks much like the original program. However rather than just being able to recognise an utterance the user now has a few commands at their disposal.

```
Commands available
 l - List entries in phone book
 m - Modify phone book
 s - Save phone book
 q - Quit application
```

```
Type command or press return to process utterance
l
 Phone book has 2 entries:
 9            -> !CODE           ( )
 01223302651  -> ENTROPIC        ( eh n t r aa p ih k sp sp )
 01223324560  -> ENTROPIC_FAX    ( eh n t r aa p ih k sp f ae k s sp sp )

Type command or press return to process utterance
m
Name ? : FRED SMITH
Phone number [blank to delete] ? : 01234987654
Pronunciation ? : f r eh d sp s m ih th sp
```

Note that the system default phone book entries (`DIRECTORY_ENQUIRIES` and `EMERGENCY_SERVICES`) are not shown, only the entries that appear in the user's phone book.

```
Type command or press return to process utterance

@50 !SIL
@100 !SIL CALL !CODE
@150 !SIL CALL !CODE
@200 !SIL CALL !CODE FRED_SMITH

Recognised words: CALL FRED_SMITH

 Is number 901234987654 correct   [y n # m] ? y

 Dialing: 901234987654
```

Here there is very little confusability and the only number that appears in the list of options is the new one added for `FRED_SMITH`. When real numbers are used there is more possibility for error.

```
Commands available
 l - List entries in phone book
 m - Modify phone book
 s - Save phone book
 q - Quit application
Type command or press return to process utterance
s
 Phone book saved to example.pbk

Type command or press return to process utterance

@50 !SIL
@100 !SIL CALL !CODE ONE TWO
@150 !SIL CALL !CODE ONE TWO THREE
@200 !SIL CALL !CODE ONE TWO THREE FOUR OH
@250 !SIL CALL !CODE ONE TWO THREE FOUR OH SIX
```

```
Recognised words: CALL ONE TWO THREE FOUR OH SIX

 Is number 9123406 correct
  or 1 for 9123456
  or 2 for 9123506
  or 3 for 9123556
  or 4 for 91234006
  or 5 for 912346   [y n # m] ? 1

 Dialing: 9123456
```

Here the first alternative was the spoken number (and `FIVE` was misrecognised as `OH`). The number was corrected by just typing in the number of the alternative.

Chapter 5

Using HAPI: Speaker Adaptation

5.1 Introduction

This chapter provides a tutorial on the adaptation facilities in HAPI to improve the accuracy of the recognition. It uses the supplied HAPI application, HAPIADAPT, to illustrate many of the examples.

5.1.1 An introduction to adaptation

Many Entropic products are supplied with speaker independent (SI) models. These acoustic models are trained so that any speaker from a generic model type (i.e. UK or US) will achieve good performance. The speaker independent model is very versatile, because it generalises to a wide spectrum of speaker accents and variability. However if only one speaker were to use the system, a speaker dependent model (SD), specifically trained on that one speaker, would significantly improve performance (in terms of recognition speed and accuracy) when compared with the speaker-independent model. This is because the speaker dependent model can account for the individual traits of that one speaker resulting in an accurate model representation of that speaker's voice.

The drawback of speaker dependent models (apart from the fact that only one person can use it) is that, if constructed from scratch, they require a large amount of speech data (typically hours of recorded data). On top of this the model construction is a complex and lengthy process.

Adaptation provides a compromise solution. By taking a speaker independent model and treating it as a seed model, together with some adaptation speech data (typically minutes rather than hours) a new acoustic model can be estimated by transforming the parameters in the speaker independent model so that they more closely resemble the adaptation material. In this way the new acoustic model will have captured some of the key characteristics of the new speaker, resulting in a more accurate model.

5.1.2 Types of adaptation

The adaptation process usually falls into a number of categories:-

Supervised A detailed transcription of the speech is available.

Unsupervised A transcription of the speech is not available. In this case it is possible to use the recogniser to generate a hypothesis, which is assumed to be the correct transcription in the adaptation process.

Online The speech application adapts to the new speaker and/or environment during the course of an interactive session with the user.

Offline The speech application adapts to the user using a collection of speech files recorded previously. The adapted models or transformations are then stored on disk, ready for use at a later stage by other applications.

5.1.3 Adaptation in HAPI

Independent of the mode of adaptation used, the adaptation process broadly consists of the following stages

1. The accumulation stage collects the speaker's data, storing statistics based on the "adaptation" data. This accumulation is performed for each enrollment utterance and can itself be broken into two stages.
   - In the alignment stage, frames of speech data are assigned to parts of the acoustic model set.
   - Once the frames of speech have been assigned, the statistics necessary for the next stage are accumulated.
2. Using the statistics accumulated thus far, a transformation can be estimated, which can be used to refine the acoustic model set.

When the adaptation process is completed the new model can be stored on disk in two different formats (as a transform model file or as an acoustic model set). These formats are described in section 10.4.2.

This tutorial will concentrate on offline supervised or unsupervised adaptation. As the main difference between online and offline is the mode of interaction with the user this does not limit the generality of the description. Also, an example of the online adaptation process is presented in section 10.4. The following sections outline key components of the example adaptation application, called HAPIADAPT, together with its use.

5.2 Application design

The remaining parts of this chapter will describe the key elements of a prototype adaptation tool, referred to as HAPIADAPT. The tool will be designed to take a set of acoustic models together with some adaptation material and produce a new model which is adapted to the new speaker and the environment.

5.2.1 Key Design

To begin with, the following key components will be required from the outset by the application.

- A set of acoustic models (HMMs) for the task.
- A dictionary mapping the required words into these models. This is necessary for the alignment process.
- Either a set of speech files or a list of speech data files used as the adaptation material (referred to as a script file), stored in a file.
- A configuration file containing the location of the some of the above components together with other application specific setups.

Supervised adaptation design

For supervised adaptation, a transcription of the adaptation data is required. For HAPIADAPT, the transcription is stored in a file (which will be referred to as a "prompt" file), with a single transcription per line, in the same order as the script file. Each transcription should contain the the spoken words in their dictionary form. An example is given below, which also includes some cases of verbalised punctuation:-

```
/* the prompt file */

THE ENROLLMENT PROCESS ENABLES THE RECOGNITION SYSTEM TO ADAPT !SEMI-COLON
IT ADAPTS TO YOUR VOICE !COMMA AND PRESENT ACOUSTIC ENVIRONMENT !PERIOD
THIS IS ACCOMPLISHED BY COLLECTING SPEECH DATA FROM THE USER !DASH
MODIFYING THE CURRENT ACOUSTIC MODELS USED BY THE RECOGNISER !PERIOD
THE UTTERANCE CURRENTLY RECORDED IS USED TO PERFORM ADAPTATION !PERIOD
THE RESULTING RECOGNITION SYSTEM BECOMES MORE ACCURATE
IT ALSO RECOGNISES SPEECH ROUGHLY FIFTY PERCENT MORE QUICKLY


/* the script file */

w:/data/sessions/bod/bod1.wav
w:/data/sessions/bod/bod2.wav
w:/data/sessions/bod/bod3.wav
w:/data/sessions/bod/bod4.wav
w:/data/sessions/bod/bod5.wav
w:/data/sessions/bod/bod6.wav
w:/data/sessions/bod/bod7.wav
```

As explained earlier, part of the adaptation process requires an alignment of the speech data to the acoustic models. This leads to a problem when a word in a transcription is outside of the vocabulary covered by the dictionary. In HAPIADAPT a background (garbage) model is used to skip the portion of the utterance corresponding to the out of vocabulary (OOV) word.

Unsupervised adaptation design

For unsupervised adaptation, a transcription of the speech data is generally not available. In this case the recogniser will be used to transcribe the speech, and provide the reference transcription required by the adaptation process. The recogniser will require a network to specify the task grammar in addition to a dictionary.

Performing adaptation

Generally, to achieve the most accurate adapted models, it is common to process the adaptation material more than once. During the first pass a single global transform is calculated and applied, to produce an adapted acoustic model. During the second pass the adapted models are used during the alignment stage, giving an improved alignment of the frames of speech to the models. At the end of the second pass more detailed transformations are produced, either class-based transforms or a fully transformed model set. These three types of adaptation transform (global, class and full) are described in section 10.2.2.

Configuration

Every application using HAPI requires a configuration file. This specifies where to find the required resources and also configures the HAPI library. Example configuration files can be found in the `UK` or `US Tutorial` directory. The example shown below is taken from the file `adapt_uk.cfg`, in the `UK` directory.

```
##
##  UK Configuration for HAPIAdapt
##  ==============================
##
##  This config provides the default settings for the HAPI adaptation
##  application.
##

# Coding and recogniser set ups + file locations
#include "final_uk.cfg"                 /* standard final config */

## configs for the alignment and adaptation procedure
##
HAPI:     ALIGNBEAM     = 320.0
HAPI:     INITBEAMADPT  = 320.0
HAPI:     INCBEAMADPT   = 160.0
HAPI:     LIMBEAMADPT   = 481.0
HAPI:     MINFRWDADPT   = 10.0
HAPI:     FULLADPTSCALE = 12.0
HADAPT:   BLOCKS        = 3
HADAPT:   OCCTHRESH     = 1000.0
HNET:     ADDSILPHONES  = "sp sil"
```

The first part of this configuration file includes the `final_uk.cfg` configuration file as used in section 4.4, telling the application where to find the UK HMM set, dictionary (and secondary dictionary) and the network for unsupervised adaptation, as well as all the recogniser and coding set-ups. The final block of definitions are for the alignment and adaptation processes. These configuration settings are all explained in chapter 10.

5.2.2 Application code definitions

To maintain a modular and self contained style, references to HAPI objects and other information necessary to the application are contained in the single structure

Application.

```
typedef struct _SpecialWords {

  char8 *startWord;
  char8 *endWord;
  char8 *bckgrdWord;

} SpecialWords;

/* application structure */
typedef struct _Application {

  hapiHMMSetObject hmms;           /* HMM object */
  hapiTransformObject trans;       /* transform object (for adaptation) */
  hapiDictObject dict;             /* dictionary object */
  hapiDictObject dictBack;         /* 2ndry dict containing the bckgrd word */
  hapiNetObject net;               /* net object */

  hapiRecType type;                /* define an LVR recogniser */
                                   /* for unsupervised adaptation mode */
  hapiTransformType transType;     /* type of transform object to create */
  hapiTransformType applyType;     /* type of transformation to apply */

  int32 supervised;                /* flag supervised/unsupervised */
  int32 useBckgrdModel;            /* flag use a background model */
  int32 firstPass;                 /* flag first pass of adaptation */
  char8 *scriptFn;                 /* Script filename */
  SpecialWords *special;           /* special important adaptation words */

} Application;
```

The application starts by allocating memory and initialising a variable of type `Application`, parsing the command-line, and checking whether a background (garbage) word model is available for supervised adaptation. If so, the background word is added to the main dictionary.

Before the adaptation process begins, the transform is created with a call to the `CreateTransform()` function below. The transform type `app->transType` is used to control the type of transform that will be generated. When a transform model file (TMF) name is supplied, it is loaded and applied to the model set, otherwise a new transform object is created and initialised.

```
/* Create a transform object for the adaptation process */
static void CreateTransform(Application *app, char8 *tmfFn) {

  int32 fail=false;

  /* check to see if tmf to load, or create a new transform object */
  if (tmfFn != NULL) {
    fail=fail || ((app->trans=hapiTransformLoad(app->hmms, tmfFn,
                                                app->transType)) == NULL);
    /* now apply the transform to the model set */
    app->applyType = (HAPI_TRANS_adapt_sil+HAPI_TRANS_use_variance+
                      HAPI_TRANS_global);
```

```
    ApplyTransform(app);
  }
  else {       /* create and initialise a transform object */
    fail=fail || ((app->trans=
hapiCreateTransformObject(app->hmms,
                                                app->transType)) == NULL);
    fail=fail || (hapiTransformInitialise(app->trans)<0);
  }

  if (fail) {
    fprintf(stderr, "Failed to create and initialise the transform object\n");
    exit(1);
  }

}
```

5.2.3 Application overview

The following pseudo-code outlines the structure of a typical multi-pass offline supervised adaptation application, and forms the basis for the code examples taken from HAPIADAPT.

```
foreach pass through the adaptation data
       foreach speech data file
              parameterize data into observations
              read in next transcription from prompt file
              create a word level lattice from the transcription
              accumulate statistics for this file
       done
       set the transform apply type
       calculate and apply the transformation to the model set
done
save the transform and/or adapted model set
```

The example code below performs two passes through the data, (mimicking the outer for loop in the pseudo-code above) and adapting the acoustic model by making calls to the function `Adapt()`. This function is explained in more detail in the following section. Notice that the only difference between the first and second pass is the apply 'type' setting. In the first pass a global transformation is applied, while in the second pass both a class-based and full transformation is applied.

```
    /* -------- first pass -- and apply global transform --------- */

    app->applyType = (HAPI_TRANS_use_variance+HAPI_TRANS_adapt_sil+
                      HAPI_TRANS_global);
    totFrames = Adapt(app,promptFn);

    /* -------------------- end first pass ----------------------- */

    /* ------- second pass - apply class+full transform ---------- */

    /* clear the accumulated statistics */
    hapiTransformInitialise(app->trans);
    totFrames = 0;
```

```
    app->applyType = (HAPI_TRANS_use_variance+HAPI_TRANS_adapt_sil+
                      HAPI_TRANS_class+HAPI_TRANS_full);
    totFrames = Adapt(app,promptFn);

    /* ------------------- end second pass ----------------------- */
```

Finally the models are saved. In this example, the final transformation is class-based as well as full, and this allows for the saving of a class-based transform model file (TMF) as well as a more detailed fully transformed model set. The model set is written to disk in the form of an MMF. The TMF can be applied to the original model set using the the HAPI function `hapiTransformApply()`.

5.2.4 The application in depth

The function `Adapt()` is called for each adaptation pass. It calls two functions:-

- `PrepareAdaptation()` which reads in the prompts and the speech data and accumulates the statistics.
- `ApplyTransform()` which calculates a new transformation based on the accumulated statistics, and applies it to the model set.

.

```
static int32 Adapt(Application *app, char *promptFn) {

  int32 totFrames=0;

  /* Collect statistics for the adaptation process */
  /* set a first update for unsupervised mode after having
     seen only UNSUP_FRAMES_ADPT frames, otherwise the
     first pass adapts on all the data */
  if (app->firstPass) {
    if (app->supervised)
      totFrames = PrepareAdaptation(app, promptFn, 0);
    else
      totFrames = PrepareAdaptation(app, promptFn, UNSUP_FRAMES_ADPT);
    app->firstPass = false;
  }
  else
    totFrames = PrepareAdaptation(app, promptFn, 0);

  /* after collection is complete the new transformation can be applied */
  ApplyTransform(app);

  return totFrames;
}
```

Depending on the request for supervised or unsupervised adaptation, the first pass has slightly different functionality. Unsupervised adaptation requires recognition to be performed; hence each pass through the data can take some time. In order to speed this process up, the first pass ends prematurely, allowing an earlier update of

the model set. Recognition with an adapted model set during the second pass is likely to be much quicker.

The function `PrepareAdaptation()` is responsible for initiating the process of reading speech data in via a script file or from the command line, and is comparable to the inner loop of the pseudo-code shown earlier. `PrepareAdaptation()` calls `ProcessFile()` for every utterance (either in the script or as specified on the command line). `ProcessFile()` is firstly responsible for reading in speech data and coding it into observations.

```
/* first read in the data into the coder buffer from the source */
while(hapiCoderRemaining(coder) != HAPI_REM_done) {
  if (hapiCoderReadObs(coder, obs) == HAPI_REM_error) {
    fprintf(stderr, "ProcessFile: Problem coding data from source!\n");
    exit(1);
  }
  nFrames += 1;
}
```

Next `ProcessFile()` must load in the corresponding transcription from the prompt file. The string buffer `buff` is now passed to a function called `GenerateLattice()`, which will generate a word level lattice object from a string. The lattice object provides the spoken word sequence required by the adaptation process.

The word level lattice is initialised with a NULL node. Its construction is continued by creating a new node for each new word in the transcription, and connecting it to the lattice by adding an arc. If a word is not found in the dictionary then either a background word model is used for the missing word, or a NULL lattice object is returned, and the utterance skipped. The function `ProcessFile()`, continues as follows.

```
/* build a word graph lattice with HAPI calls, and then accumulate
   statistics for adaptation purposes */
 nFrames = 0;
 lat = GenerateLattice(app->dict, app->special, app->useBckgrdModel,
 buff);

/* if lattice object is valid proceed with the accumulation of statistics,
   otherwise skip the utterance */
 if (lat != NULL) {
   nFrames = hapiTransformAccumulate(app->trans, lat, coder);
   hapiDeleteLatObject(lat);
 }
 else if (trace&T_TOP) {
   printf("Utterance is not being used for adaptation purposes\n");
   fflush(stdout);
 }
```

So, for each utterance in the speech data source, we continue to accumulate statistics. Once all the speech data has been accumulated we can calculate and apply the transform, noting that the transform apply type must be set before the following function is called.

```
static void ApplyTransform(Application *app)
{
   int32 fail = false;
```

```
   fail=fail || (hapiTransformApply(app->trans, app->applyType)<0);

   if (fail) {
     fprintf(stderr, "Failed to apply the transform to the model set\n");
     exit(1);
   }
}
```

The transform that is calculated is always with respect to the original (pre-transformed) model set. This means that if a speaker independent model was loaded into HAPIADAPT, any TMFs that are produced can (and should) be applied to this speaker independent model. The `hapiTransformApply()` function automatically checks what state the acoustic model set is in – i.e. whether a transform has been applied to it. If so `hapiTransformApply()` reverts the model back to its original form before creating and applying the new transform.

Finally the application saves a TMF and an MMF, and finishes up by deallocating the used objects and exits.

5.3 Running the adaptation tool

This part of the adaptation tutorial describes how to go about adapting the model sets provided in the `Tutorial` directory. The adapted model sets can then be tested within the extended dialer example from section 4.4. In order to adapt the model sets, some adaptation data must be collected. Once data collection is completed, adaptation is performed and the new model sets can be tested. Note that all of the following command lines must be run from within the "language directory" (either UK or US) in the `HAPI/Tutorial` directory.

5.3.1 Data collection

First of all move to the "language directory" (i.e. `HAPI/Tutorial/UK`) and create a new directory or folder in the `adapt` directory. This directory will hold all the waveforms, script files and TMFs for a particular user. In this example we are assuming a directory has been created for user `djk`.

The tool HAPIADAPT features a very simple data collection routine, that only allows you to collect speech data in single session. It is invoked using the following command line.

```
> HAPIAdapt -u djk -p phone_prompts.pmt -T 1 -C adapt_uk.cfg -c adapt/djk
```

Here the prompts for the data collection are taken from the file `phone_prompts.pmt`. It contains 25 simple telephone number prompts. The `-c` option tells HAPIADAPT to collect data and put it in the `adapt/djk` directory. Like the extended dialer application, the data collection begins with initialisations and a speech calibration.

```
Performing data collection only

 Data collection calibration:

  If the mean speech level isn't 20-40dB higher than silence
```

```
   level please check the audio system and try again

Press return to measure speech levels

 Now say 'She sells sea shells by the sea shore'
 Thank you

 Mean speech level in dBs 59.43 (6.2%), silence 21.74

Calibration completed -- ready to start data collection
```

The calibration is followed by the data collection. The user is asked to press enter and then speak the prompt shown.

```
 1) DIAL OH ONE EIGHT ONE   EIGHT OH SIX OH EIGHT SIX
When ready to speak press enter and say prompt 1)...
Thank you
```

Once the utterance has been spoken the user is asked to select a command from a menu...

```
Choices available
 n - move to next prompt
 p - play back utterance
 q - quit data collection
 r - re-record utterance
Type command
n
Saving file to adapt/djk/adapt_djk01.wav

 2) PLEASE PHONE ZERO ONE SIX FOUR SEVEN   THREE ONE TWO FOUR NINE SIX
When ready to speak press enter and say prompt 2)...
Thank you
Choices available
 n - move to next prompt
 p - play back utterance
 q - quit data collection
 r - re-record utterance
Type command
n
Saving file to adapt/djk/adapt_djk02.wav

 3) CONTACT ZERO ONE TWO SEVEN SIX   ZERO SIX ZERO FIVE ZERO NINE
When ready to speak press enter and say prompt 3)...
Thank you
Choices available
 n - move to next prompt
 p - play back utterance
 q - quit data collection
 r - re-record utterance
Type command
q
Saving file to adapt/djk/adapt_djk03.wav

Data collection completed -- 3 utterances collected
```

There are 25 prompts available, and unless a minimum of 10 utterances are collected, little impression in speed and accuracy is likely. A script file, as described in section 5.2.1, called `adaptation.scp` is also produced by HAPIADAPT in the `adapt/djk` directory.

5.3.2 Performing adaptation

Supervised adaptation

After the data was collected using HAPIADAPT as described in section 5.3.1, (and assuming transcriptions exist in the form of prompts) the following command line can be used to perform supervised adaptation, where the prompt file is specified using the `-p` option. Note that superior performance is generally achieved using supervised adaptation (as opposed to unsupervised).

```
> HAPIAdapt -t -B -T 1 -C adapt_uk.cfg -p phone_prompts.pmt
  -u djk -S adapt/djk/adaptation.scp -M adapt/djk

Using HMM Set       : ./ecrl_uk.mmf
and HMM List        : ./ecrl_uk.list
Using Dictionary    : ./final_uk.dct
Performing supervised adaptation...
        If a prompt contains a word not in the dictionary,
        the prompt will be skipped
File   1: adapt/djk/adapt_djk01.wav
File   2: adapt/djk/adapt_djk02.wav
File   3: adapt/djk/adapt_djk03.wav

First Pass
Update and apply global transform using 1750 frames!

File   1: adapt/djk/adapt_djk01.wav
File   2: adapt/djk/adapt_djk02.wav
File   3: adapt/djk/adapt_djk03.wav

Second Pass
Update and apply class transform using 1750 frames!

Saved tmf to adapt/djk/djk.tmf
```

Unsupervised adaptation

When data is available, but there are no transcriptions, the following command line can be used to perform the unsupervised adaptation, where each data file is first recognised, and the hypothesis is used as the transcription for the adaptation process.

```
> HAPIAdapt -t -B -T 1 -C adapt_uk.cfg -u djk -m
  -S adapt/djk/adaptation.scp -M adapt/djk

Using HMM Set       : ./ecrl_uk.mmf
and HMM List        : ./ecrl_uk.list
Using Dictionary    : ./final_uk.dct
Using Network/LM    : ./final.lat
Performing unsupervised adaptation...
```

```
File   1: adapt/djk/adapt_djk01.wav
!SIL DIAL !CODE OH ONE EIGHT ONE EIGHT OH SIX OH EIGHT SIX !SIL
File   2: adapt/djk/adapt_djk02.wav
!SIL PLEASE PHONE !CODE ZERO ONE SIX FOUR SEVEN THREE ONE TWO FOUR NINE SIX !SIL
File   3: adapt/djk/adapt_djk03.wav
!SIL CONTACT !CODE ZERO ONE TWO SEVEN SIX ZERO SIX ZERO FIVE ZERO NINE !SIL

First Pass
Update and apply global transform using 1750 frames!

File   1: adapt/djk/adapt_djk01.wav
!SIL DIAL !CODE OH ONE EIGHT ONE EIGHT OH SIX OH EIGHT SIX !SIL
File   2: adapt/djk/adapt_djk02.wav
!SIL PLEASE PHONE !CODE ZERO ONE SIX FOUR SEVEN THREE ONE TWO FOUR NINE SIX !SIL
File   3: adapt/djk/adapt_djk03.wav
!SIL CONTACT !CODE ZERO ONE TWO SEVEN SIX ZERO SIX ZERO FIVE ZERO NINE !SIL

Second Pass
Update and apply class transform using 1750 frames!

Saved tmf to adapt/djk/djk.tmf
```

5.3.3 Testing the adapted models

A small amount of code (shown below) has been added to the `final.c` program to allow it to read in a transform model file (TMF) specified in the configuration and apply it to the model set before recognition begins.

```
   /* Load transform if in the hapi config file and transform model set */
   if ((tmfFile=hapiOverrideConfStr("TMF",NULL))==NULL)
     ahi->trans=NULL;
   else {
     transType = (HAPI_TRANS_adapt_sil+HAPI_TRANS_use_variance);
     fail=fail || ((ahi->trans=hapiTransformLoad(ahi->hmms, tmfFile,
 transType))==NULL);
     if (ahi->trans==NULL) {
       printf("Transform %s not suitable for loaded HMM set\n", tmfFile);
       exit(1);
     }
     fail=fail || (hapiTransformApply(ahi->trans,transType)<0);
     printf(" Loaded and applied TMF to model set\n");
   }
```

To run `final` with a TMF, you will need to amend the `final_uk` or `final_us` configuration file. Alter the line:-

```
 # HAPI:   TMF           = adapt/uid/uid.tmf
```

Remove the comment marker and change `uid` to reflect your personal identifier that you have been using. Then run the dialer as before, noting the extra line of dialogue feedback.

```
> final final_uk.cfg
 Loading HMMs, dictionary and network
 Loaded and applied TMF to model set
 Initialisation complete
```

The chapter has given a brief overview of how to collect data, construct a TMF and use it to adapt a simple model set for running in a simple recognition application. For such a simple task it may be difficult to notice much improvement in terms of speed and accuracy. A further test can be carried out by re-running the adaptation phase. This time, construct a TMF for the other model set, i.e. an American constructs a TMF for the UK system, and then test by using the UK system with and without the transform. The difference in speed and accuracy should be more apparent.

Chapter 6

Interfacing to a source driver

The standard HAPI library includes built-in support for direct audio input for several machine types as well as the ability to process HTK parameterised and several formats of waveform files. Input from these sources does not require any application support beyond selecting the appropriate source type and ensuring the configuration file is set up correctly.

It also includes the ability to use application defined drivers for processing data from other sources. In order to make use of this facility the application programmer must write the set of functions defined below and create a `hapiSrcDriverDef` structure containing pointers to these driver functions. This structure is then passed to the **hapiRegisterSrcDriver** function which (if the operation is successful) returns a newly allocated `hapiSourceType`. To process data collected using this driver the application creates a source object of the new type and rather than using the audio or file driver built into HAPI the application driver will be used to provide data.

This chapter describes an example driver which reads data from a stream. The driver utilises an internal buffer to hold data that has been read from a stream before it is read by HAPI. Although the example just reads data from a file it has been designed so that it can be modified to interface to almost any data source without needing to change the interface to HAPI. To demonstrate its versatility the final section of this chapter describes how the driver can be modified to read from a stream based audio device.

6.1 An Example Driver

The example driver just reads data from a stream (which can be sourced from a file, a pipe or even an audio device). However it is structured in a way that insulates the input interface (which reads from the stream) from the output interface (which supplies the data to HAPI). This allows the input interface to be easily changed to read data from an alternative source or in a different way.

Unlike most of the HAPI functional interface (which uses simple data types and passes several arguments to a function rather than encapsulating them in a predefined structure), each source definition is held in a structure containing both the driver specification and pointers to the functions called by HAPI to control the driver.

The example driver definition is shown below

```
static hapiSrcDriverDef ubs = {
```

```
    NULL,               /* Place holder for global hook */
    0.0,                /* Any sample frequency may be okay */
    0,                  /* Just use config to choose detector */
    HAPI_SDT_none,      /* Gets filled in by the register function */
    HAPI_SDT_none,      /* No obvious playback device */
    uBufferRegister,    /* Read config parameters and set up global info */
    uBufferShutdown,    /* Free resources allocated in Register */
    uBufferCreate,      /* Allocate object specific info */
    uBufferDelete,      /* Free resources allocated by Create */
    uBufferInitialise,  /* Could be NULL as it just notes the coder */
    uBufferSrcAvailable, /* Could be NULL as files are always available */
    uBufferPlayWave,    /* Could be NULL as there is no playback method */
    uBufferPrepare,     /* Set up for later start */
    uBufferComplete,    /* Clear up after utterance */
    uBufferStart,       /* Open file and skip header */
    uBufferStop,        /* Could be NULL as files stop themselves */
    uBufferNumSamples,  /* Return number of samples readable now */
    uBufferGetSamples,  /* Get certain number of samples */
};
```

The new driver is used by first creating the above structure then registering it with HAPI. Registration is accomplished with a single call that returns a new source type. This is used as the argument to **hapiCreateSourceObject** when a source object reading from the driver is required.

Since the example source driver will be able to read a number of different types of data (depending upon the format of the data read from the stream) a number of registration functions are used to register a driver for each supported source data type.

```
hapiSourceType uBufferHRegisterULaw(void)
{
   /* Set format to 8 bit ulaw */
   ubs.srcType=HAPI_SDT_mu8;
   return(hapiRegisterSrcDriver(&ubs));
}

hapiSourceType uBufferHRegisterALaw(void)
{
   /* Set format to 8 bit alaw */
   ubs.srcType=HAPI_SDT_alaw8;
   return(hapiRegisterSrcDriver(&ubs));
}

hapiSourceType uBufferHRegisterLinear(void)
{
   /* Set format to 16 bit linear */
   ubs.srcType=HAPI_SDT_lin16;
   return(hapiRegisterSrcDriver(&ubs));
}
```

The application uses the new driver to capture data by selecting the appropriate data format, registering the appropriate driver and then creating a new source object of the returned type.

```
fail=fail || ((ahi->src=uBufferRegisterLin16())<0);
```

Example value	Definition
NULL	`void *gInfo;` Initial value for a global driver hook that is passed to each driver function
0.0	`hapiTime sampPeriod;` Sample period of the data sourced from the driver. Zero indicates that the driver will operate at the configured sampling rate.
0	`int32 forceDetUse;` Determines the use of the speech detector with source objects using this driver. The value zero indicates the driver can be used with the speech detector and the channel configuration determines whether it is used.
HAPI_SDT_none	`hapiSrcDriverDataType srcType;` Defines the type of data sourced from the driver. In the example the type of data is set within the call that registers the device with HAPI to either `HAPI_SDT_mu8`, `HAPI_SDT_alaw8` or `HAPI_SDT_lin16`.
HAPI_SDT_none	`hapiSrcDriverDataType playType;` Defines the type of data that can be played by the driver. The example driver does not support playback.

Table. 6.1 The Extended Source Definition (a)

```
.
.
.
   fail=fail || ((ahi->source=hapiCreateSourceObject(ahi->src))==NULL);
```

For example the tutorial dialer program can be modified to use the new driver just by including a line to register the source with HAPI.

```
main(int argc,char **argv)
{
+  hapiSourceType uBufferHRegisterULaw(void);
.
.
+  src=uBufferHRegisterULaw();

   if ((ahi=InitialiseRecogniser(argv[1],src,type,mode))==NULL)
      exit(1);
```

As explained above the source definition is held in a structure which holds driver parameters and functions. The first few members of the structure define the capabilities of the source driver. An explanation of the individual fields and the values in the example driver are shown in table 6.1.

The remaining members of the definition are function pointers that are called by the HAPI library when it needs information from the driver, wishes to communicate information to the driver or even just wishes to give the driver an opportunity to do some processing. During processing of an utterance every frame the driver is given at

Example value	Definition
`uBufferRegister`	`void (*esRegister) (hsi,hsd);` After the source driver has been successfully registered with HAPI (but before **hapiRegisterSrcDriver** returns) this function is called.
`uBufferShutdown`	`void (*esShutdown) (hsi);` Called when the HAPI is (and therefore the driver should be) shutdown. *Currently HAPI cannot be shutdown and so this call is never made.*
`uBufferCreate`	`hapiSrcDriverObjectInfo *(*esCreate) (hsi);` Called when a new `hapiSourceObject` of the driver's type is created.
`uBufferDelete`	`void (*esDelete) (hsi, oInfo);` When a source object using this driver is deleted this function is called to allow the driver to free resources allocated in the **esCreate** call.
`uBufferInitialise`	`void (*esInitialise) (hsi, oInfo, coder);` This function is called when the source object is initialised.
`uBufferSrcAvailable`	`hapiRemainingStatus (*esSrcAvailable) (hsi, oInfo);` Return a value indicating how many more utterances are available for processing.
`uBufferPlayWave`	`void (*esPlayWave)(hsi, oInfo, n, data);` Some drivers also have the ability to play an audio message to the user. When they can this function is used to play back a waveform buffer.

	`hapiSrcDriverStatus *hss;`
	`hapiSrcDriverDef *hsd;`
	`hapiSrcDriverObjectInfo oInfo;`
	`hapiCoderObject coder;`
	`int n;`
	`void *data;`
	Argument type definitions for above functions.

Table. 6.2 The Extended Source Definition (b)

least one opportunity to do some processing. In general this will occur many times each second.

The first group of functions (see table 6.2) provide registration, creation and ancillary functions that are used to set up the new source driver before it is used for data capture and to shut it down once data capture has finished.

The remainder of the functions (shown in table 6.3) are used to process each utterance. HAPI uses an opportunistic read method of taking data from the driver. Each time the coder object is accessed during data capture (usually twice per frame - once to determine the coder status and once to actually read the observation) the driver is queried to determine how much data is available. Any data available immediately is read and buffered within the coder straight away.

In general each utterance will be processed with the following sequence of function

calls.

- `esPrepare` Prepare for start of utterance
- `esStart` Start data collection
  - `esNumSamples` Query number of sample available
  - * `esGetSamples` Read those samples
  - `[esStop]` May be called to stop collection
- `esComplete` Utterance processing complete

The core of the driver (and the only part which which must be implemented by every new source extension) are the final two calls **esNumSamples** and **esGetSamples**.

As mentioned above HAPI opportunistically reads data from the driver. It uses the **esNumSamples** function to poll the state of the driver during the processing of an utterance. The number returned is used to determine how many frames of data can be read immediately. During data capture HAPI does not want to block unless it must, this is only the case if all previous data has already been processed and the application tries to read a frame of data. To facilitate this method of operation the **esNumSamples** function returns a value that indicates whether data capture is in progress as well as the number of samples available for reading without the need to block. Then the **esGetSamples** function is used to read the available data which is buffered within HAPI until it is processed. However, if the buffer runs out of data to process, it will call the **esGetSamples** function to read data that the **esNumSamples** function has indicated is not yet available. In this case the call should wait (preferably blocking rather than looping and using the CPU unnecessarily) until data is available before returning all the requested samples. The only time that **esGetSamples** should return with less than the number of samples requested is when the end of the input data has been reached.

It should also be noted that if blocking needs to occur during the set-up for each utterance it should occur in **esStart** rather than in the **esPrepare** function. This will be the case if the driver reads data from an input source not under the application's direct control. For example, if the application is a server daemon processing data from several clients, input can only begin when the client starts sending data. The daemon can prepare for recognition and then have the prepared recogniser/coder/source wait in the **hapiSourceStart** call for the client to begin data transmission.

Since HAPI ensures that driver functions are called frequently (at least once per frame read/processed), they can be used to provide an opportunity for the driver to make use of the CPU (instead of using multiple threads or processes). However, since such time-slicing is co-operative, the driver must ensure that any such processing occurs in relatively small blocks to allow HAPI to continue to provide relatively 'real-time' performance.

Each of the functions is passed a `hapiSrcDriverStatus` stucture which is used pass the global driver hook to each function and to return status information from each call to the driver. The members of the function are shown in table 6.4. Since the members of this structure are initialised to default values which indicate the operation was successful the driver code only needs to update them when an error occurs or if the global driver hook needs to be updated.

6.2 Implementation

The code for the driver can be split three section

- The buffer used to hold data read from the input channel but not yet read by HAPI.
- The interface to HAPI which reads the data from the buffer and controls driver operation.
- The driver that reads data from the input channel and adds it to the buffer.

Only the final section is dependent upon the actual data source with the buffer and HAPI interface code usable with almost any device.

Example value	Definition
`uBufferPrepare`	`void (*esPrepare)(hsi, oInfo, name, sampPeriod);` During preparation of the coder for the next utterance this function is called to allow the source to perform any computationally expensive initialisation.
`uBufferComplete`	`void (*esComplete) (hsi, oInfo);` This function is called when the current utterance has been processed and the associated coder is being completed.
`uBufferStart`	`void (*esStart) (hsi, oInfo);` When **hapiStartSource** is called by the application, HAPI calls this function to start data capture.
`uBufferStop`	`void (*esStop) (hsi, oInfo);` Called to stop data capture.
`uBufferNumSamples`	`hapiRemainingStatus (*esNumSamples) (hsi, oInfo);` This function is used by HAPI to determine the input status of the driver.
`uBufferGetSamples`	`int32 (*esGetSamples) (hsi, oInfo, n, data);` Return captured samples to HAPI.

	`hapiSrcDriverStatus *hss;`
	`hapiSrcDriverObjectInfo oInfo;`
	`hapiTime sampPeriod;`
	`int n;`
	`void *data;`
	Argument type definitions for above functions.

Table. 6.3 The Extended Source Definition (c)

`hapiStatusCode status;`
Status code for operation. This is initialised to `HAPI_ERR_none` before each driver function call and so only needs to be updated in case of error.
`char8 statusMessage[STR_LEN];`
Status message for operation. This is initialised to an empty string but may be used to pass a textual description of any error to the user since this buffer is appended to a generic description and is made available via the **hapiCurrentStatus** function.
`void *gInfo;`
Value of global driver hook passed to each driver function.

Table. 6.4 The `hapiSrcDriverStatus` structure.

6.2.1 The Buffer

The driver uses an internal buffer to hold samples that have been read by the driver but have not yet been requested by HAPI. In this example the buffer is implemented as a self contained buffer object with methods to add samples, remove samples and query status.

```
typedef struct sbuffer {
   /*  mutex;                    /* Place holder for read/write mutex */
   int n;                        /* Number of valid samples held in buffer */
   int in;                       /* Position marker for input to buffer */
   int out;                      /* Position marker for output from buffer */
   int total;                    /* Total number of samples in buffer */
   int ss;                       /* Sizeof each sample in buffer */
   void *data;                   /* Buffered data */
}
SBuffer;
```

If the driver is implemented using multiple threads it will be necessary to use the commented out field as a mutex. This would be used to ensure that the driver thread filling the buffer and the HAPI thread reading from the buffer do not try to access the structure at the same time. If this occurred it could potentially produce inconsistent results or leave the buffer in an invalid state.

The first three fields describe the current state of the buffer in terms of the number of samples currently held in the buffer and the current read and write positions. The final three fields define the data buffer itself. With the `data` area holding `total` samples each of which is `ss` bytes in size.

There are two main routines used to access the buffer. The fist is used to extract data from the buffer;

```
static int SBufferGetSamples(SBuffer *buf,int n,void *data)
{
   int m;

   /* CLAIM BUFFER MUTEX */
   if (buf->out<0 || n<0) n=-1;
```

```
   else {
      if (buf->n<n) n=buf->n;
      if (buf->out+n>buf->total) {
         m=(buf->total-buf->out);
         memcpy(data,((char*)buf->data)+buf->out*buf->ss,buf->ss*m);
         memcpy(((char*)data)+m*buf->ss,buf->data,buf->ss*(n-m));
      }
      else
         memcpy(data,((char*)buf->data)+buf->out*buf->ss,buf->ss*n);
      buf->out=(buf->out+n)%buf->total;
      buf->n-=n;
      if (buf->in<0 && buf->n<=0) buf->out=-1;
   }
   /* FREE BUFFER MUTEX */

   return(n);
}
```

With the second used to add samples to the buffer

```
static int SBufferAddSamples(SBuffer *buf,int n,void *data)
{
   int m;

   /* CLAIM BUFFER MUTEX */
   if (n<0) buf->in=-1;
   else if (n==0) {
      if (buf->out<0) buf->out=0;
   }
   else if (buf->in<0)
      n=-1;
   else {
      if (buf->total-buf->n<n) n=buf->n;
      if (buf->in+n>buf->total) {
         m=(buf->total-buf->in);
         memcpy(((char*)buf->data)+buf->in*buf->ss,data,buf->ss*m);
         memcpy(buf->data,((char*)data)+m*buf->ss,buf->ss*(n-m));
      }
      else
         memcpy(((char*)buf->data)+buf->in*buf->ss,data,buf->ss*n);
      buf->in=(buf->in+n)%buf->total;
      buf->n+=n;
   }
   /* FREE BUFFER MUTEX */

   return(n);
}
```

As well as indicating the point in the buffer at which samples should be extracted or entered, the `in` and `out` fields are used to hold status information as well. When the `in` field is negative this indicates that the input channel is inactive and samples cannot be added to the buffer. If the `out` field is negative all the buffer is complete and all the samples have been read.

As well as transferring data into the buffer, the **SBufferAddSamples** function is used to inform the buffer of changes to the input state of the buffer. The function is

called with a negative number of samples to transfer to indicate the end of the source data and to mark data input as complete. Then, once all the samples have been read from the inactive buffer, the `out` field is set to `-1` to indicate the both input and output are complete.

Two further functions are used to test for these conditions and to find the amount of free space available for new samples.

```
static int SBufferAvailSamples(SBuffer *buf)
{
   int n;

   /* CLAIM BUFFER MUTEX */
   if (buf->out<0) n=-1;
   else n=buf->n;
   /* FREE BUFFER MUTEX */

   return(n);
}

static int SBufferComplete(SBuffer *buf)
{
   int n;

   /* CLAIM BUFFER MUTEX */
   if (buf->in<0) n=-1;
   else n=0;
   /* FREE BUFFER MUTEX */

   return(n);
}
```

Another couple of functions are used to determine how much free space is available in the buffer and the total size of the buffer.

```
static int SBufferFreeSpace(SBuffer *buf)
{
   int n;

   /* CLAIM BUFFER MUTEX */
   if (buf->n<0) n=-1;
   else n=buf->total-buf->n;
   /* FREE BUFFER MUTEX */

   return(n);
}

static int SBufferSize(SBuffer *buf)
{
   int n;

   /* CLAIM BUFFER MUTEX */
   n=buf->total;
   /* FREE BUFFER MUTEX */

   return(n);
```

```
}
```

The final functions are used to manage, allocated and free the buffer.

```
static void SBufferInitialise(SBuffer *buf)
{
   /* CLAIM BUFFER MUTEX */
   buf->in=buf->out=buf->n=0;
   /* FREE BUFFER MUTEX */
}

void SBufferDelete(SBuffer *buf)
{
   buf->in=buf->out=buf->n=-1;
   free(buf->data); buf->data=NULL;
   free(buf);
}

static SBuffer *SBufferCreate(int total,int ss)
{
   SBuffer *buf;

   buf = malloc(sizeof(SBuffer));
   buf->total = total; buf->ss = ss;
   buf->data = malloc(ss*total);
   buf->in=buf->out=buf->n=-1;

   return(buf);
}
```

6.2.2 The HAPI Interface

The second section of the device driver is the actual interface to HAPI. This has to implement the set of functions defined in the `hapiSrcDriverDef` structure which not only read the data (from the above buffer) but also control the operation of the driver.

For this example the application will directly control the data capture and so the actions on the stream need to be controlled by the HAPI interface. This is implemented by using an *event* handler which uses messages from the HAPI interface functions to control the stream driver. In the example code the messages are passed via function calls (although this could make use of a different operation paradigm).

The set of events generated by HAPI are quite small (and some of these can be ignored by most types of driver).

```
typedef enum {
   NONE,
   INITIALISE,
   CREATE,
   DELETE,
   PREPARE,
   COMPLETE,
   START,
   STOP,
   BLOCK
}
HEvent;
```

Each of these corresponds to a specific call in the `hapiSrcDriverDef` structure except the final `BLOCK` event . This is called when HAPI requests data that has not yet been buffered. The value of the event is used to pass the minimum number of extra samples that are needed to fulfill a **esGetSamples** request. In this case the stream driver should block until the required number of samples have been read in.

When the function associated with each event occurs it calls

```
static int eventProcess(bInputInfo *bii,HEvent event);
```

to pass the event on to part of the code that control the input stream.

Associated with each object that will use the *uBuffer* source driver is a structure that holds the information needed by the HAPI interface.

```
typedef struct ubufferobjectinfo {
   SBuffer *buf;
   int option;
   bInputInfo *bii;
   hapiCoderObject coder;
   hapiTime sampPeriod;
   struct ubufferobjectinfo *next;
   struct ubufferobjectinfo *prev;
}
uBufferObjectInfo;
```

The `bInputInfo` structure holds the corresponding information needed by the data capture code whilst the `SBuffer` structure is the data buffer defined above.

In addition to the object dependent information a small amount of global information is needed. This consists of a copy of the `hapiSrcDriverDef` and a pointer to the start of a linked list of objects using this driver.

```
typedef struct ubufferglobalinfo {
   hapiSrcDriverDef hsd;
   uBufferObjectInfo *list;
}
uBufferGlobalInfo;
```

Many of the HAPI driver functions in the example have little to do. The **esRegister** and **esShutdown** functions are a good example. These just allocate and initialise or free the `uBufferGlobalInfo` structure associated with the driver instance. The **esRegister** function is passed a copy of the `hapiSrcDriverDef` for this driver and so pieces of code that are shared several different drivers can take note of the particular configuration being used. For example, the buffered stream driver can be registered to read different data types and so a copy of the `hapiSrcDriverDef` is used to determine the current input sample type (and size).

```
static void uBufferRegister(hapiSrcDriverStatus *hss,hapiSrcDriverDef *hsd)
{
   uBufferGlobalInfo *gi;
   int *swap;
   short x, *px;
   unsigned char *pc;

   /* Allocate info needed by particular source object */
   gi=malloc(sizeof(uBufferGlobalInfo));
```

```
   gi->list=NULL;

   /* Set global info */
   hss->gInfo=gi;
   gi->hsd=*hsd;
}

static void uBufferShutdown(hapiSrcDriverStatus *hss)
{
   uBufferGlobalInfo *gi=hss->gInfo;

   /* Free up global info */
   hss->gInfo=NULL;
   free(gi);
}
```

The **esCreate** and **esDelete** functions perform the corresponding allocation and disposal when each object is created. The create function also creates the structure that holds information about the input stream. This probably should be done by the event handler despite needing a different functional interface that allows it to return the allocated structure for future use by the event hander.

```
static hapiSrcDriverObjectInfo *uBufferCreate(hapiSrcDriverStatus *hss)
{
   uBufferObjectInfo *bi;
   uBufferGlobalInfo *gi=hss->gInfo;
   int32 *ip;

   /* Allocate info needed by particular source object */
   bi=malloc(sizeof(uBufferObjectInfo));
   /* Add to linked list of objects using this driver */
   bi->next=gi->list;gi->list=bi;bi->prev=NULL;
   if (bi->next!=NULL) bi->next->prev=bi;

   /* Illustrate use of config variables */
   if ((ip=hapiOverrideConfInt("BUFFERSIZE",NULL))==NULL)
      bi->buf=SBufferCreate(BUFFERSIZE,(gi->hsd.srcType&HAPI_SDT_bytes));
   else
      bi->buf=SBufferCreate(*ip,(gi->hsd.srcType&HAPI_SDT_bytes));
   if ((ip=hapiOverrideConfInt("INPUTOPTION",NULL))==NULL)
      bi->option=0;
   else
      bi->option=*ip;

   /* Send message that new object has been created */
   /*  also gives CPU time to driver */
   bi->bii=bInputCreate(bi->buf,gi->hsd.srcType,
                        bi->option,gi->hsd.sampPeriod);
   eventProcess(bi->bii,CREATE);

   return((void*) bi);
}

static void uBufferDelete(hapiSrcDriverStatus *hss,
                          hapiSrcDriverObjectInfo oInfo)
```

```
{
   uBufferObjectInfo *bi=oInfo;
   uBufferGlobalInfo *gi=hss->gInfo;

   /* Send message that object has been deleted */
   /*  also gives CPU time to driver */
   eventProcess(bi->bii,DELETE);

   /* Remove from linked list */
   if (bi->next!=NULL) bi->next->prev=bi->prev;
   if (bi->prev!=NULL) bi->prev->next=bi->next;

   /* Free up resources */
   SBufferDelete(bi->buf);
   free(bi->bii->temp);
   free(bi->bii);
   free(bi);
}
```

The create function also illustrates how configuration variables can be used by the driver to control operation in the same way that they are used internally within HAPI. Also note how the **esDelete** function (in common with all object specific functions) is passed the `uBufferObjectInfo` pointer returned by the **esCreate** function.

The initialisation function also does very little apart from note the coder in the object specific information structure.

```
static void uBufferInitialise(hapiSrcDriverStatus *hss,
                              hapiSrcDriverObjectInfo oInfo,
                              hapiCoderObject coder)
{
   uBufferObjectInfo *bi=oInfo;
   uBufferGlobalInfo *gi=hss->gInfo;
   int fail=0;

   /* Keep record of coder (although it is not used) */
   bi->coder=coder;

   /* Send message that object has been initialised */
   /*  also gives CPU time to driver */
   fail = fail || (eventProcess(bi->bii,INITIALISE)<0);

   /* Error - cannot open audio */
   if (fail) hss->status=HAPI_ERR_generic;
   else
      printf(" uBuffer initialised\n");
}
```

Since this driver cannot play waveforms back to the user there is no need to define a **esPlayWave** function (the corresponding function pointer in the source definition can be set to `NULL`). However, to keep the definition complete the example driver has a function which just sets the status to indicate an error occurred.

```
static void uBufferPlayWave(hapiSrcDriverStatus *hss,
                            hapiSrcDriverObjectInfo oInfo,
                            int n,void *data)
```

```
{
   /* uBuffer cannot play audio to user */
   hss->status=HAPI_ERR_generic;
}

static hapiRemainingStatus uBufferSrcAvailable(hapiSrcDriverStatus *hss,
                                               hapiSrcDriverObjectInfo oInfo)
{
   /* uBuffer source is always available */
   return(HAPI_REM_more);
}
```

Similarly there is no need for a **esSrcAvailable** function as the default if the function is not implemented is to assume that the driver can produce more utterances upon demand.

The remaining functions actually implement the utterance by utterance processing. Although the **esPrepare**, **esComplete**, **esStart** and **esStop** functions appear to do little the events they generate actually begin and end data capture from the stream.

```
static void uBufferPrepare(hapiSrcDriverStatus *hss,
                           hapiSrcDriverObjectInfo oInfo,
                           char8 *name,hapiTime sampPeriod)
{
   uBufferObjectInfo *bi=oInfo;
   uBufferGlobalInfo *gi=hss->gInfo;
   int fail=0;

   /* Set up more info */
   bi->bii->sampPeriod=sampPeriod;
   if (name!=NULL)
      strcpy(bi->bii->fnBuf,name),bi->bii->fn=bi->bii->fnBuf;
   else
      bi->bii->fn=NULL;

   /* Send message that object should be prepared for utterance */
   /*  also gives CPU time to driver */
   fail = fail || (eventProcess(bi->bii,PREPARE)<0);

   /* Error - cannot open audio */
   if (fail) hss->status=HAPI_ERR_generic;
   else
      printf(" uBuffer prepared\n");
}

static void uBufferComplete(hapiSrcDriverStatus *hss,
                            hapiSrcDriverObjectInfo oInfo)
{
   uBufferObjectInfo *bi=oInfo;
   uBufferGlobalInfo *gi=hss->gInfo;
   int fail=0;

   /* Send message that utterance processing is complete */
```

```
   /*  also gives CPU time to driver */
   fail = fail || (eventProcess(bi->bii,COMPLETE)<0);

   /* Error - cannot open audio */
   if (fail) hss->status=HAPI_ERR_generic;
   else
      printf(" uBuffer completed\n");
}

static void uBufferStart(hapiSrcDriverStatus *hss,
                         hapiSrcDriverObjectInfo oInfo)
{
   uBufferObjectInfo *bi=oInfo;
   uBufferGlobalInfo *gi=hss->gInfo;
   int fail=0;

   /* Send message that data capture should begin NOW */
   /*  also gives CPU time to driver */
   fail = fail || (eventProcess(bi->bii,START)<0);

   /* Error - cannot start audio */
   if (fail) hss->status=HAPI_ERR_generic;
   else
      printf(" uBuffer started\n");
}

static void uBufferStop(hapiSrcDriverStatus *hss,
                        hapiSrcDriverObjectInfo oInfo)
{
   uBufferObjectInfo *bi=oInfo;
   uBufferGlobalInfo *gi=hss->gInfo;
   int fail=0;

   /* Send message that data capture should stop NOW */
   /*  also gives CPU time to driver */
   fail = fail || (eventProcess(bi->bii,STOP)<0);

   /* Error - cannot stop audio */
   if (fail) hss->status=HAPI_ERR_generic;
   else
      printf(" uBuffer stopped\n");
}
```

Unlike all of the above functions (which accomplish their actions elsewhere through the event processor) the functions that actually check the driver status and read samples are self contained.

```
static hapiRemainingStatus uBufferNumSamples(hapiSrcDriverStatus *hss,
                                             hapiSrcDriverObjectInfo oInfo)
{
   uBufferObjectInfo *bi=oInfo;
   uBufferGlobalInfo *gi=hss->gInfo;
   SBuffer *buf=bi->buf;
   int fail=0,n,c;
```

```
   /* Give CPU time to driver */
   /*  only do this every time if EventProcess is guaranteed to be quick */
   fail = fail || (eventProcess(bi->bii,NONE)<0);

   n=SBufferAvailSamples(buf);
   c=SBufferComplete(buf);
   if (n<0) n=0;                /* Not started */
   else if (!c) n=-1-n;         /* Input continues */
   /* Input complete */

   if (fail) hss->status=HAPI_ERR_generic;
   else
        printf(" uBuffer : Status - %d samples to read\n",n);
   return(n);
}
```

The `hapiRemainingStatus` returned indicates both the amount of data in the buffer and whether data capture is still actively occurring. Whilst data capture is happending (and **SBufferCompete** returns zero) the function returns `HAPI_REM_more` minus the number of samples already in the buffer. Once data capture has finished the returned number is just the number of samples in the buffer.

The **esGetSamples** function begins by checking if there are enough samples currently in the buffer to fulfill the request. If there aren't the function stays in a loop processing events (and therefore giving CPU time to the data capture side of the driver). This actually blocks waiting for enough data to fulfill the request (the type of the event indicates the number of samples needed). Once there are enough samples available (or if the driver knows that the request will never be fulfilled) the function reads the appropriate number of samples and returns the number correctly read to HAPI.

```
static int32 uBufferGetSamples(hapiSrcDriverStatus *hss,
                               hapiSrcDriverObjectInfo oInfo,
                               int n,void *data)
{
   uBufferObjectInfo *bi=oInfo;
   uBufferGlobalInfo *gi=hss->gInfo;
   SBuffer *buf=bi->buf;
   int fail=0,c,r;

   /* Give CPU time to driver */
   fail = fail || (eventProcess(bi->bii,NONE)<0);

   /* Check that there are sufficient samples */
   while (r=SBufferAvailSamples(buf), c=SBufferComplete(buf),
          !c && (r>=0 && r<n)) {
      /* Should block and wait for data */
      /*  which in this case consists of */
      /*  giving CPU time to driver */
      fail = fail || (eventProcess(bi->bii,BLOCK+n-r)<0);
   }
   /* Actually read frame of data */
   r=SBufferGetSamples(buf,n,data);

   if (fail) hss->status=HAPI_ERR_generic;
   else
```

```
        printf(" uBuffer : Read %d/%d samples in frame %d\n",
               r,n,nf);
   return(r);
}
```

Since this driver does all message passing using function calls, the implementation of the event handler is very simple. When it is called with a particular event it just calls the appropriate function in the data capture driver code.

```
static int eventProcess(bInputInfo *bii,int event)
{
   if (event>BLOCK)
      /* Told to wait for a specific number of samples */
      bInputGetAvailSamples(bii,event-BLOCK);
   else
      /* Opportunistic catch up with input data */
      bInputGetAvailSamples(bii,0);
   /* Then process the event */
   switch(event) {
    case CREATE:     /* Don't need to do anything when object created. */
       break;
    case DELETE:     /*  or when it is deleted. */
       break;
    case INITIALISE: /* or when it is initialised */
       break;
    case PREPARE:    /* Sample period and filename are now available. */
       SBufferInitialise(bii->buf); break;
    case COMPLETE:   /* Free utterance resources */
       bInputComplete(bii); break;
    case START:      /* Begin data capture - NOW */
       bInputStart(bii); break;
    case STOP:       /* End data capture - NOW */
       bInputStop(bii); break;
   }
   /* And just in case that took a long time ...*/
   /*  Opportunistic catch up with input data */
   bInputGetAvailSamples(bii,0);
   /* No error checking in driver */
   return(0);
}
```

6.2.3 Data Capture

For this example the source of the samples is a stream (a combination of a file descriptor and a file pointer is used by the data capture code).

The stream is opened when a `START` event is received and closed when it has been exhausted (either in the function called when the `COMPLETE` event occurs or possibly in the function associated with `STOP`).

Whilst the input stream is active the function **bInputGetAvailSamples** is used to read as much data as possible from the stream and add it to the buffer. Normally this function does not block and just reads until there is no more data available. However, if the function is called with a positive second argument this is interpreted as a request to block until that many samples have been read. Every time an event

is received **bInputGetAvailSamples** is called to read the available data into the buffer. This ensures that latency is kept to a minimum and that the state of the buffer accurately reflects the state of the input. The buffer must be an accurate reflection of the input since it is used by the HAPI interface for this purpose.

In common with the HAPI interface to the driver the data capture code utilises one structure, which holds the details of the current status of the stream, per source object.

```
typedef struct binputinfo {
   hapiSrcDriverDataType type;    /* Type of data being read */
   int option;              /* Input option */
   char *fn;                /* File name to read */
   char fnBuf[STR_LEN];     /*  and buffer to store it in */
   hapiTime sampPeriod;     /* Requested sample rate */

   FILE *stream;            /* File pointer for input device */
   int fd;                  /* File descriptor for input device */
   int eof;                 /* End of input flag */

   SBuffer *buf;            /* Sample buffer */

   int tcnt;                /* Number of samples in temp buffer */
   int tss;                 /* Sizeof samples in temp buffer */
   void *temp;              /* temp buffer for samples read in */
}
bInputInfo;
```

This structure can be broken into three sections

- Task description

  The type of the data samples, the filename, sample period and appropriate configuration variables appear first in the structure. These are set up when the `binputinfo` object is created.

- Stream information

  The next three fields hold the necessary information about the input stream. Both a file pointer and file descriptor are used to control the input stream as well as an end-of-file flag indicating when data capture has ended.

- Buffering

  Samples are first read into a temporary buffer before being transfered into the buffer which the HAPI interface reads from.

When a source object using the example driver is created a buffer input object is created by the driver using the following call.

```
static bInputInfo *bInputCreate(SBuffer *buf,hapiSrcDriverDataType type,
                                int option,hapiTime sampPeriod)
{
   bInputInfo *bii;

   /* Record sampling period */
   bii=malloc(sizeof(bInputInfo));
```

```
   bii->type=type; bii->option=option;
   bii->buf=buf; bii->sampPeriod=sampPeriod;
   bii->stream=NULL; bii->fd=-1; bii->eof=-1;

   bii->tcnt=1024;
   bii->tss=(type&HAPI_SDT_bytes)*((type&HAPI_SDT_chans)>>8);
   bii->temp=malloc(bii->tcnt*bii->tss);

   return(bii);
}
```

This allocates and initialises the `bInputInfo` structure before returning its address to the HAPI interface. This pointer provides the link between the HAPI object and the stream and is passed to the event handler each time that it is called.

Input is started by opening the stream and then informing the buffer that the input channel is active.

```
static int bInputStart(bInputInfo *bii)
{
   SBuffer *buf=bii->buf;
   int fail=0;

   fail = fail || (bii->fn==NULL);
   fail = fail || ((bii->fd=open(bii->fn,O_RDONLY))<0);
   fail = fail || ((bii->stream=fdopen(bii->fd,"r"))==NULL);
   if (!fail) setbuf(bii->stream,NULL);
   fail = fail || (fseek(bii->stream,(long)bii->option,SEEK_SET)<0);

   /* Mark data capture started */
   fail = fail || (SBufferAddSamples(buf,0,NULL)<0);
   bii->eof=0;

   if (!fail)
      printf(" bInput started\n");
   return(fail?-1:0);
}
```

Stopping the data is accomplished by reading pending samples from the data stream and then closing the stream. Finally the buffer is marked to indicate the input channel is no longer active. This call will only normally be useful for 'real-time' sources, which are reading from a real device (such as the workstation's audio input) or pipe generating data synchronously with the application's operation.

```
static int bInputStop(bInputInfo *bii)
{
   SBuffer *buf=bii->buf; int fail=0;

   /* Get remaining data */
   bInputGetAvailSamples(bii,0);
   /* Close file now */
   if (bii->stream!=NULL) fclose(bii->stream),bii->stream=NULL;
   if (bii->fd>=0) close(bii->fd),bii->fd=-1;
   /* Mark no more data available */
   SBufferAddSamples(buf,-1,NULL);
```

```
   if (!fail)
      printf(" bInput stopped\n");
   return(fail?-1:0);
}
```

The previous function makes use of the **bInputGetAvailSamples** function to mop-up any remaining data in the pipe. This function reads data from the input stream and adds it into the buffer. As outlined above it needs to operate two different modes.

- Non-blocking

  The function tries to read as many samples as it has space for. However, non-blocking read calls are used and the function ends when no more data can be read.

- Blocking

  The function tries to read a specific number of samples. A blocking read call is used and so the function only returns when the stream is closed, its data finished or the required number of samples has been read in and added to the buffer.

Choice of operation mode is determined by the second parameter passed to the function. If this is zero the first mode of operation is required and the function just reads any data available without blocking. This occurs in several places in the code and allows the stream driver to read data 'in the background' whilst other processing occurs. In the second mode the function is called directly from the event handler in response to an event requesting that it block and read the specified number of samples.

```
static int bInputGetAvailSamples(bInputInfo *bii,int min)
{
   SBuffer *buf=bii->buf;
   char *p;
   int c,r,n,b,t=0;

   /* Check if data capture is happening */
   c=SBufferFreeSpace(buf);
   if (bii->fd<0 || c<0 || bii->eof) return(-1);

   /* Set correct blocking mode and size for the read */
   r=fcntl(bii->fd, F_GETFL);
   if (min>0) /* Block and read minimum number of samples */
      r &= ~(O_NONBLOCK|O_NDELAY),n=min;
   else  /* Non-blocking read of up to size of free space in buffer */
      r |= O_NONBLOCK, n=c;
   fcntl(bii->fd, F_SETFL, r);

   while(t<n) {
      /* Calc size of block to read */
      p=bii->temp; b=c=n-t; if (c>bii->tcnt) b=c=bii->tcnt;
      while(b>0) {
         r=fread(p,bii->tss,b,bii->stream);
         if (r<0 && errno==EAGAIN) r=0;
```

```
         else
            bii->eof = bii->eof || feof(bii->stream);
         if (r<=0 || (r!=b && bii->eof)) break;
         p+=r*bii->tss;b-=r;
      }
      /* Update total read so far */
      b=c-b; t+=b;
      /* Add these to buffer */
      if (b<=0 || SBufferAddSamples(buf,b,bii->temp)!=b || b<c) break;
   }
   /* Mark buffer complete when input exhausted */
   if (bii->eof = bii->eof || feof(bii->stream))
      SBufferAddSamples(buf,-1,NULL);

   /* Information */
   if (t>0 && t==n)
      fprintf(stderr," bInputGetAvailSamples: Read %d samples%s\n",
              t,(bii->eof?" with EOF":""));
   else if (bii->eof || t>0)
      fprintf(stderr," bInputGetAvailSamples: Only read %d/%d samples%s\n",
              t,n,bii->eof?" with EOF":"");
   return(t);
}
```

The final function is called once the utterance processing is complete. This closes the stream if necessary but does not deallocate any memory as this occurs when the HAPI object is deleted.

```
static int bInputComplete(bInputInfo *bii)
{
   SBuffer *buf=bii->buf;
   int fail=0;

   if (bii->stream!=NULL) fclose(bii->stream),bii->stream=NULL;
   if (bii->fd>=0) close(bii->fd),bii->fd=-1;
   bii->eof=-1;

   if (!fail)
      printf(" bInput completed\n");
   return(fail?-1:0);
}
```

6.3 Variations

The above driver is relatively versatile and the implementation quite general. Normally minor modifications are all that is required to read from different types of stream with slighltly more substantial changes required for devices that use different low level interface. For example, the following changes are sufficient to convert the driver to read from a Sun/Solaris type audio device.

```
static int bInputStart(bInputInfo *bii)
{
   SBuffer *buf=bii->buf;
```

```
int fail=0,f,g,i;

audio_info_t info;

fail = fail || ((bii->fd = open(AUDIO_IO, O_RDONLY + O_NDELAY)) < 0);
fail = fail || ((bii->stream = fdopen(bii->fd,"r"))==NULL);
if (!fail) setbuf(bii->stream,NULL);

fail = fail || (ioctl(bii->fd, AUDIO_GETINFO, &info) < 0);
AUDIO_INITINFO(&info);
f = 1.0E+07 / bii->sampPeriod;
info.record.sample_rate = f; info.record.pause = 1;
info.record.port = (((bii->option&1)?AUDIO_LINE_IN:0)+
                        ((bii->option&2)?AUDIO_MICROPHONE:0));
switch (bii->type) {
 case HAPI_SDT_lin16: /* 16 bit linear */
    info.record.channels = 1;
    info.record.precision = 16;
    info.record.encoding = AUDIO_ENCODING_LINEAR;
    break;
 case HAPI_SDT_mu8: /* 8 bit mulaw */
    info.record.channels = 1;
    info.record.precision = 8;
    info.record.encoding = AUDIO_ENCODING_ULAW;
    break;
 default:
    fail=1;
    break;
}
if (!fail && ioctl(bii->fd, AUDIO_SETINFO, &info) < 0) {
   static float sampFreqs[] = {
      8000, 9600, 11025, 16000, 18900, 22050, 32000, 37800, 44100, 48000, 0
   };
   for(i=0,f=g; sampFreqs[i]>0 && sampFreqs[i]<f; i++) g=sampFreqs[i];
   printf("Trying alternate frequency %d vs %d\n",f,g);
   info.record.sample_rate = f = g;
   fail = fail || (ioctl(bii->fd, AUDIO_SETINFO, &info) < 0);
}

/* Convert stream for blocking reads */
fail = fail || ((g=fcntl(bii->fd, F_GETFL)) < 0);
g = g & ~(O_NONBLOCK|O_NDELAY);
fail = fail || (fcntl(bii->fd, F_SETFL, g) < 0);

/* Flush everything collected so far */
fail = fail || (ioctl(bii->fd, I_FLUSH, FLUSHR)<0);
fail = fail || (ioctl(bii->fd, AUDIO_GETINFO, &info)<0);
if (!fail) info.record.pause=0;
fail = fail || (ioctl(bii->fd, AUDIO_SETINFO, &info)<0);

/* Mark data capture started */
fail = fail || (SBufferAddSamples(buf,0,NULL)<0);
bii->eof=0;
```

```
   if (!fail)
      printf(" bInput started\n");
   return(fail?-1:0);
}

static int bInputStop(bInputInfo *bii)
{
   SBuffer *buf=bii->buf;
   int fail=0;
   audio_info_t info;

   /* Stop data collection */
   fail = fail || (ioctl(bii->fd, AUDIO_GETINFO, &info)<0);
   if (!fail) info.record.pause=1;
   fail = fail || (ioctl(bii->fd, AUDIO_SETINFO, &info)<0);

   /* Get remaining data */
   bInputGetAvailSamples(bii,0);
   /* Close file now */
   if (bii->stream!=NULL) fclose(bii->stream),bii->stream=NULL;
   if (bii->fd>=0) close(bii->fd),bii->fd=-1;
   /* Mark no more data available */
   SBufferAddSamples(buf,-1,NULL);

   if (!fail)
      printf(" bInput stopped\n");
   return(fail?-1:0);
}
```

These changes only effect the setup for each utterance (which is more specialised for the audio driver) and a minor change to the code that stops data capture which is meant to synchronise the call more accurately with "real-time".

The changes required to read from a different type of device (that does not implement a stream interface) are more substantial. However, as long as the underlying device supports either non-blocking reads or a method of determining the amount of data currently available.

For example the **InputGetAvailSamples** function can be implemented without non-blocking reads for stream input by making use of an `ioctl` implemented for both pipes and the audio device (although not files themselves). This `ioctl` is used to determine the number of bytes waiting in the input queue that can be read without blocking before actually reading the data.

```
static int bInputGetAvailSamples(bInputInfo *bii,int min)
{
   SBuffer *buf=bii->buf;
   char *p;
   int c,r,n,b,t=0;

   /* Check if data capture is happening */
   c=SBufferFreeSpace(buf);
   if (bii->fd<0 || c<0 || bii->eof) return(-1);

   if (min>0) /* Block and read mimimum number of samples */
```

```
      n=min;
   else  { /* Determine how many samples can be read NOW */
      if (ioctl(bii->ctld, FIONREAD, &r) < 0) n=0;
      else n=r/2;
   }

   while(t<n) {
      /* Calc size of block to read */
      p=bii->temp; b=c=n-t; if (c>bii->tcnt) b=c=bii->tcnt;
      while(b>0) {
         r=fread(p,bii->tss,b,bii->stream);
         bii->eof = bii->eof || feof(bii->stream);
         if (r<=0 || (r!=b && bii->eof)) break;
         p+=r*bii->tss;b-=r;
      }
      /* Update total read so far */
      b=c-b; t+=b;
      /* Add these to buffer */
      if (b<=0 || SBufferAddSamples(buf,b,bii->temp)!=b || b<c) break;
   }
   /* Mark buffer complete when input exhausted */
   if (bii->eof = bii->eof || feof(bii->stream))
      SBufferAddSamples(buf,-1,NULL);

   /* Information */
   if (t>0 && t==n)
      fprintf(stderr," bInputGetAvailSamples: Read %d samples%s\n",
              t,(bii->eof?" with EOF":""));
   else if (bii->eof || t>0)
      fprintf(stderr," bInputGetAvailSamples: Only read %d/%d samples%s\n",
              t,n,bii->eof?" with EOF":"");
   return(t);
}
```

Part II

HAPI in Depth

Chapter 7

Developing HAPI Applications

The second part of this book will describe HAPI in depth.

The following chapters describe the programming interface in depth with examples of the use of each function in the library and how to use each object (both by itself and as part of a complete system).

This chapter starts with a discussion of HAPI's use in applications. It then tries to outline the jobs of both the designer of the recognition system and the applications programmer who must incorporate the system into the final application. Finally it describes the relationship (both the similarity and differences) between HTK and HAPI to allow current HTK users to quickly become familiar with HAPI and port their recognition systems to HAPI applications.

7.1 System Design

The design of a speech enabled system can be divided into two stages.

- Recogniser design.

  This process involves the specification of the individual recogniser components. Deciding which acoustic models are needed, specifying the task grammar and the dictionary pronunciations. Often specifying the task grammar is the most complicated of these tasks and will be done in an incremental manner as testing reveals the range of expressions produced by typical users.

- Application design.

  Once the recognition task has been specified the application code to initialise, operate and process the results from the recogniser is needed. When complex semantic processing of the results is required extra information may be included in the network and dictionary in order to simplify the application program. However, this tends to take the form of extra information added to the current resources rather than modifications to their underlying form.

The recognition components (acoustic models, dictionaries and syntax networks) specified by the first of these tasks can be obtained from several sources.

The dictionary is dependent upon the underlying form of the acoustic models. If these are word or phrase models (where a single acoustic model represents the whole realisation of a particular output token) then the information content in the dictionary is minimal and it can be easily specified. However when sub-word acoustic models are used the dictionary needs to specify pronunciations for each word which are both accurate and representative. Each pronunciation must represent a reasonable way of saying the word and the range of pronunciations must cover all the different ways in which the word can be said. Fortunately there are a variety of sources for phonetic dictionaries. Probably the simplest is to use a set of acoustic models supplied by Entropic together with a dictionary in a format suitable for use with HAPI. When this is not convienient a dictionary must be either prepared in-house, a public domain one used or rights acquired to use one of the many commercial ones available. The LDC and ELRA are well-known sources of speech recognition data, both for dictionaries and acoustic data.

As stated above acoustic models can be obtained from Entropic or task specific ones can be generated using the HTK toolkit together with suitable acoustic training data. The exact form these models should take will depend upon the task, the performance required and the target system. For simple tasks on limited platforms a small set of uncomplicated models (such a continuous density monophones with only a few Gaussians per state) may be most suitable. Whereas, for a more complex task where there is more acoustic confusability more detailed and complex models (which will require more storage space and computational resources) may be needed to maintain a suitable level of accuracy.

Finally the task grammar is needed. This will always have to be produced by the customer although Entropic can supply tools to make the design easier. HTK includes facilities for generating networks from a textual grammar and from statistical language models. However, probably the easiest way to generate networks is to use the *grapHvite* package. *grapHvite* contains a graphical network builder, a set of acoustic models and a dictionary allowing the user to build up a network graphically whilst testing the recogniser direct from the workstation's built in audio input.

Once the basic recogniser and syntax has been created the system designer must interact with the application designer to modify the dictionary and network definitions to incorporate any extra information needed in the application's post-processing of the results. Part III of this book explains some of the issues that arise when real applications are written and how these effect the detailed design of the network and addition of output symbols to the dictionary.

The application programmer then designs the recogniser in terms of HAPI objects and the applications results processing. Most applications will use HAPI in three stages

- Initialisation.

  HAPI objects are allocated and initialised (loaded from disk). These are then kept in memory ready for quick access when needed. Normally the memory required to store several networks (and maybe dictionaries) will be small compared to the acoustic models and so this potential inefficiency will not be a concern (the memory used can be swapped to disk by the OS if memory is scarce).

- Use.

The application runs making use of HAPI to perform recognition when necessary. More complex applications may need to construct new networks from lattices built or modified on the fly but many will just run a single recogniser using a single HMMSet and dictionary possibly using multiple networks.

- Completion.

  Once the application has finished it should tidy up and delete the various objects it has allocated. Care must be taken when freeing up objects because the hierarchical interdependencies will prevent freeing of objects used by others. The higher level objects must be freed first. Note that the application may need to close down because of an internally generated HAPI error and in these cases the programmer may wish to avoid further HAPI calls which could just exasperate the problem.

Creation and initialisation of objects is relatively simple when they can be read from files on disk (as they will be in most cases). However, as the tutorial example has indicated, modification of objects and use of multiple dictionaries, networks and lattices can quickly become more complicated. Often it is easiest to start with a simple application and add the complex frills later.

7.2 HAPI for HTK users

A user familiar with HTK should be able to get their recognition systems working under HAPI with the very minimum of fuss. The tutorial example from the first part of the book and code fragments in this part should provide a starting point for simple applications. The actual compilation environment and software structure is close to that used for the HTK tools themselves and so users familiar with compiling the tools should be able to start with HAPI almost immediately.

The use of HTK style configuration files means that initially a simple HVite style application can be produced from the generic `main.c` program and a slightly extended configuration file with the minimum of effort. This application needs the command line parameters from HVite to be converted to configuration parameters in the manner described in the next few chapters. Once this is done the application can be run to provide a similar utterance by utterance single network recogniser that approximates HVite being used direct from audio.

When providing HAPI users with recognition resources the HTK user should bear in mind the following points

- HMMs.

  HAPI prefers models in MMF files and insists that each HMMSet contains at least one MMF file. Consequently the HTK user should in general use the HMM definition editor to merge model files into a small number of MMF files.

  They must also generate a configuration file containing all the parameters that they used whilst encoding data. Many HTK users will be in the habit of coding both training and test data initially and then running all experiments from the coded parameter files. In general the HAPI user will want to run direct from audio so they will need a full data coding setup. Care must also be taken when choosing parameterisation to ensure that the one chosen can be used with direct audio. Things to avoid include energy normalisation and CMN (unless the dynamic version is used; see section 11.4).

- Dictionaries.

  HTK dictionaries can be used without change. However, HAPI expects phonetic dictionaries and if modification of sparse dictionaries is required the `!PHONE_SET` word must be added to define the phone set. Expanded triphone dictionaries in which each pronunciation is expressed in terms of triphone model names can be used for network generation but will not give access to pronunciations because of the number of 'phones' in the phone set.

- Network definitions.

  HTK network definitions can also be used unchanged. These can be loaded in as lattices and then converted to networks or can be used directly to create a network. Any of the tools that generate networks for HVITE can be used to generate a lattice/network definition for HAPI.

  However none of the HTK tools for generating lattices is particularly intuitive and so the primary method of generating word level syntaxes will probably the graphical network builder that comes as part of the *grapHvite* package.

Apart from these minor points HTK users should find it easy to use their recognition systems in HAPI applications and can use the new programming interface to try them in real applications.

Chapter 8

HAPI, Objects and Configuration

The next few chapters describe the individual HAPI objects. Each HAPI object has an chapter devoted to it, with the exception of the coder and source objects which are described together in a single chapter. The chapters describe the general structure and purpose of the object, its creation, any specific types associated with the object, its configuration and how it is used, including code fragments illustrating the use of each function associated with the object.

Chapter 9 concerns the **hapiHMMSetObject** which provides the application and other HAPI objects with the database of acoustic models needed for recognition. The source of the acoustic data **hapiSourceObject** and the object that converts this (normally waveform level) data into the observations required by the recogniser (the **hapiCoderObject**) are described together in chapter 11 because they cannot operate independently of each other. Use of the **hapiDictObject** is explained in chapter 12 which gives examples of how the dictionary can be modified and added to within the application. Word level lattices (**hapiLatObjects**), used to specify syntaxes for recognition networks and to hold results when NBest alternatives are required, are described in chapter 13. Creation and initialisation of the actual decoder dependent recognition network stored as a **hapiNetObject** is explained in chapter 14. However, its use is described chapter 15 which deals with the top level **hapiRecObject** that brings all the recognition components together for a decoder to produce results (in the form of **hapiResObjects**) for each utterance.

Some HAPI functions are not specifically associated with any one object. These generic functions are mainly concerned with initialisation, error handling and configuration overrides. This chapter describes these generic functions and gives examples of their use.

8.1 Initialisation and status checks

The first task for any application that wishes to use HAPI is to initialise and configure the library. These operations are achieved with a single HAPI function call.

```
if (hapiInitHAPI(config,errFunc)<0)
   printf("HAPI failed to initialise\n\n");
```

```
   else printf("HAPI Initialised\n\n");
```

The first argument to **hapiInitHAPI** is the filename of the configuration file to load. If this is `NULL` a default name of `"config"` is assumed.

The second argument specifies an application function to be called in case of an unrecoverable internal HAPI error. Normally this will just be used to warn the user that the application is about to shut down and to allow them to write out any unsaved work but some applications may be able to continue without needing to use HAPI.

In simple console applications the following function could be used to print a message to `stderr` and end the application.

```
void errFunc(int32 n)
{
   fprintf(stderr,"hapiError: Code %d\nAborting\n\n",n);
   exit(n);
}
```

Non fatal errors are reported back to the application via function return values. When these are integers the actual error code is often returned but other functions can only return a special value indicating that an error occurred but not the actual code itself. For functions that return pointers `NULL` is used to indicate an error (although it may be a legitimate return value for some functions). Functions returning floating point values return a special value that should be checked with the macro `IS_HAPI_NAN`. If this is true then an error has occurred. In these cases when errors occur the application needs to call a separate function to determine the exact error code.

```
   hapiStatusCode
   char buf[STR_LEN]="";

   st=hapiCurrentStatus(buf);
   if (st!=0)
      fprintf(stderr,"Status %d - %s\n",st,buf);
```

This function can also be used to obtain a textual description of the problem whenever errors occur.

Configuration variables can be set to control the behaviour when errors occur. The default action is to issue a warning message to the console whenever a function returns a failure code, however these warnings can be disabled (for final applications) or used to force program termination (during initial testing).

It is also possible to have internal HTK and HAPI errors cause the program to abort (and dump core)

Module	Name	Default	Description
HAPI	ABORTONERR	F	Causes HError to abort rather than exit.
	WARNONERR	T	Display a warning every time an error condition is returned by a HAPI function.
	ENDONERR	F	Display a warning and exit application every time an error condition is returned by a HAPI function.
	SHOWCONFIG	F	Print the current configuration to stdout.

Setting the final of these parameters `T` causes the printing of the read in configuration to `stdout`. This allow the user to check this matches what was expected and preserves a permanent record of the setup in a log file.

8.2 Configuration in general

In HTK configuration parameters are used to control the detailed operation of individual library modules. Configuration parameters are used in HAPI for this purpose as well as specifying file names, pruning and other parameters that would be specified on the command line for HTK tools.

Module name	Function
`HAPI`	The HAPI interface. For example, resource filenames (such as HMM list files, dictionaries and default networks), recognition scaling and pruning parameters.
`HSHELL`	File IO and configuration handling For example, specifies filters used to read files and low level file format specification.
`HWAVE`	Waveform file handling Parameters effecting reading of waveform files from disk.
`HAUDIO`	Direct audio input Parameters controlling reading of waveform from direct audio input.
`HPARM`	General coder behaviour Specifies detailed signal processing used to generate observations from waveform sources (although these can be overridden by use of multiple channels the `HParm` parameters are the defaults).
`HADAPT`	Adaptation operation A few parameters to control the adaptation functionality that are not selected by choosing different adaptation transform types.
`HNET`	Recognition network expansion Specifies how the word level syntax is expanded into a model level network suitable for use by the decoder.
`HREC`	Recogniser operation A few parameters that control the more esoteric recogniser features that are not selected by choosing different recogniser types and modes.

Consequently each HAPI application configuration file does not only include the parameters controlling the functioning of HTK library modules (principally the HPARM module which controls how waveform data is coded). The configuration also contains HAPI parameters which specify the make-up of the HMMSet, the location of the dictionary and syntax and the general pruning parameters for the recogniser used by the application.

Although many configuration parameters associated with each HAPI object can be overridden by object and parameter specific function calls some cannot. In order

to override these values generic parameter override function are provided which can override parameters within the actual configuration itself rather than for individual objects. These functions should be called before the object in question is created to alter any of the configuration parameters effecting that object[1].

These generic functions can also be used to read arbitrary parameters from the configuration. This gives the application a way to store information (such as file names) in the configuration file rather than including them in the source code or needing a separate application implemented configuration file.

For example the tutorial in part one describes how the name of the users phone book is stored in the configuration and then accessed in the application.

```
char8 *userName;
userName=hapiOverrideConfStr("PHONEBOOK",NULL);
```

This use of configuration parameters allows all system dependent parameters and file names to be changed without recompilation of the application.

There are four generic override functions, each specific to a particular type of configuration variable (although the numeric types are freely convertible).

```
char8 *oldStr,  *newStr="New String", *parameterName="PARAMNAME";
int32 oldInt,    newInt=12345;
float 32 oldFlt, newFlt=12.345;
int32 oldBool,   newBool=1;

oldStr=hapiOverrideConfStr(parameterName,newStr);
oldInt=hapiOverrideConfInt(parameterName,&newInt);
oldFlt=hapiOverrideConfFlt(parameterName,&newFlt);
oldBool=hapiOverrideConfBool(parameterName,&newBool);
```

One of the most useful facilities inherited from HTK is the ability to define filters through which different files are read. This feature is accessed by setting a filter configuration parameter to a system command that should be used to access files of a particular type.

These filters can be used to accomplish a vast array of transformations. For example, one could be a simple on the fly expansion of networks to allow them to be used in compressed format

```
HNETFILTER = "gzip -d -c $.gz"
```

And another to perform complex file format conversion and signal processing whilst reading in a waveform file

```
HWAVEFILTER = "btosps -f 16000 -S 1024 -n 1 -c "Downsampled file" $ - | \
         rem_dc - - | esfconvert -s 8000  -c 3800 -t 400 -R 96 - - | \
         e2sphere - -"
```

When the filter is set the file is opened as a pipe. For reading, rather than opening the named file directly, the command given in the filter definition (with the `$` replaced by the file name) is executed and its output read instead of the contents of the file. For writing, the data that would normally have been written directly to the file is passed a input to the command. For example the following filter could be used to compress all output network/lattice files as they are written to disk

[1] Currently altering HTK configuration parameters is ineffective since they are only read once at initialisation and are not rescanned each time they are used. This will change in subsequent versions of HAPI.

```
HNETOFILTER = "compress > $.Z"
```

Each major file type has its own input and output filters with the following names.

Input filter name	Output filter name	Module
HMMLISTFILTER	HMMLISTOFILTER	Filter used to access HMM list files.
HMMDEFFILTER	HMMDEFOFILTER	HMM definition files.
HWAVEFILTER	HWAVEOFILTER	waveform data files.
HPARMFILTER	HPARMOFILTER	parameterised data files.
HLABELFILTER	HLABELOFILTER	label files.
HDICTFILTER	HDICTOFILTER	dictionary files.
HNETFILTER	HNETOFILTER	network/lattice files.
HLANGMODFILTER	HLANGMODOFILTER	language model files.

Chapter 9

hapiHMMSetObject

A `HMMSetObject` represents a collection of HMMs. It provides the necessary acoustic models for recognition. Since there is no specific requirement for API facilities to access the individual models or their parameters, the object is essentially a database which is used by the network and recogniser objects. Consequently the functions associated with the HMMSet object only concern its specification, its creation and its deletion and not access to the models.

In general a single HMMSet can contain several different types of model (provided that the model names are unique) and can be shared among many recognisers. Most applications will therefore only require a single HMMSet object. However each recogniser is given the opportunity to take control of the HMMSet and modify it in ways specific to its operation. If this occurs or if different types of model with the same name (for example both telephone and wideband versions of the same models) are required then multiple HMMSet objects must be used.

The remainder of this chapter discusses the contents of the HMMSet in more detail, how it is defined to HAPI and also how it is created and used within an application program.

9.1 Definition

The definitions of the HMMs are stored off-line in files on disk. Typically these are stored in master model files (MMFs) which hold multiple definitions in a single file. Although the use of individual HMM definition files is supported by HAPI it is discouraged. In fact HAPI, unlike HTK, requires that each HMMSet contains at least one MMF file. There is no real advantage to using individual HMM definition files, MMF files provide a simpler and more compact method for storing sets of models.

The format of HMM definition files (both as MMFs and as individual models) is described in the HTK Book. However, it is unlikely that the HAPI user will ever have to be concerned with the details of the format as HTK will normally be used to generate the models. As mentioned above the one significant difference in the definition of HMMSets between HAPI and HTK is that HAPI expects that each model set should include at least one MMF file. However, since this can be empty all model sets accepted by HTK can also be used with HAPI.

A single HMMSet can contain models of several different types as long as the name of each model is unique within the HMMSet. For example a single set could

contain both whole word models as well as monophones allowing the use of a mixed dictionary in which the most common words are represented by a single whole word model whereas more obscure words are represented by a sequence of sub-word models. However all models within a single HMMSet must share a common data parameterisation. When systems require recognition of data with different parameterisations (for example both telephone bandwidth and wideband data) multiple HMMSets must be used.

It is also worth making clear that although the format of the model file is independent of the type of acoustic data used to produce the models, the actual parameters of those models are not. Their values depend upon both the type of the acoustic data (such as the sampling frequency and resolution as well as the acoustic channel used) and also the way in which the data is encoded (which is outlined in the description of the coder object later in this book). Consequently each model set will be matched to only one particular acoustic setup and the system designer must ensure that the configuration and acoustic input of their particular application matches closely the one in which the models were trained.

The file(s) containing the HMMSet can be specified either in the configuration file or via configuration override function calls. When multiple model sets are required only the first can be specified in the configuration file and subsequent sets must be specified directly by the application. Remember however that it is possible to use configuration variables to pass information into the application so all file names can be stored in the configuration file rather than in the application source code.

The configuration variables relevant to the HMMSet object are shown in the table below.

Module	Name	Default	Description
HAPI	`HMMLIST`	`"list"`	List of names of acoustic HMMs.
	`HMMDIR`	`"."`	Directory in which to find acoustic HMMs.
	`HMMEXT`	`""`	Extension for HMM files.
	`MMF`	`"MODELS"`	Name of single master model file or
	`MMF[1-9]`		of multiple master model files.
HMODEL	`ALLOWOTHERHMMS`	`T`	Allow MMFs to contain HMM definitions which are not listed in the HMM List

The HAPI module parameters are concerned with specifying the files that define and make up the HMMSet. When HMM definitions are loaded the configuration parameters are used in the following manner.

- Scan all MMF files in order (these will either consist of the single `$MMF` file or the series `$MMF1 ... $MMFN`). Read in all low level macros plus any models that appear in `$HMMLIST`.

- Scan `$HMMLIST` and for any models not yet loaded read the definition from the corresponding file. (For model named `$hmm` the definition is loaded from `$HMMDIR/$hmm` if `$HMMEXT` is an empty string or from `$HMMDIR/$hmm.$HMMEXT` otherwise).

This method of specifying and loading HMMSets corresponds closely with that used for the HTK tools and consequently each of the above configuration parameters

is equivalent to a command line option to the HTK decoder HVITE. This correspondence is shown in the examples below (the examples assume that the command line option $NAME is the value of the configuration variable NAME)

So a HAPI configuration file of the form

```
HAPI:  HMMLIST = list
HAPI:  HMMEXT = ""
HAPI:  HMMDIR = .
HAPI:  MMF = MODELS
```

is equivalent to the following command line for HVITE

```
HVite ... -d $HMMDIR -x $HMMEXT -H $MMF ... $HMMLIST ....
```

And when multiple MMF files are needed,

```
HAPI:  HMMLIST = list
HAPI:  HMMEXT = ""
HAPI:  HMMDIR = .
HAPI:  MMF1 = mmf1
HAPI:  MMF2 = mmf2
HAPI:  MMF3 = mmf3
```

is equivalent to

```
HVite ... -d $HMMDIR -x $HMMEXT -H $MMF1 -H $MMF2 -H $MMF3 ... $HMMLIST ....
```

The one HTK configuration parameter (`ALLOWOTHERHMMS`) that affects the HMMSet object controls the behaviour when unknown models are encountered in MMF files. By default no warnings will be issued if the master model files contain HMMs which do not appear in the list. These definitions will be quietly skipped. By setting `ALLOWOTHERHMMS` to `F` unknown models appearing in MMF files will generate error messages making it possible to check that the MMF files correspond exactly with the list being used.

9.2 Usage

This section uses example code to illustrate how to use the HMMSet object within an application program. Since the application does not use the contents of the HMMSet object directly the only tasks that it will ever need to perform is to specify and create the object, which is then passed to other HAPI functions that require HMM definitions, and delete the object when it is no longer required.

The example code fragment below defines a simple structure will be used to hold information related to the HMMSet. This structure will be used both to specify the HMMSet as well as to pass information between functions in our example code.

```
/* ---------------------  HMMSet example code --------------------- */

typedef struct applhmminfo {
   hapiHMMSetObject hmms; /* Reference to HMMSet object */
   char8 *list;           /* HMMSet list file name */
   char8 *dir;            /* HMM file directory */
   char8 *ext;            /* HMM file name extension */
```

```
   char8 **mmfs;          /* HMMSet MMF file names (NULL terminated array) */
}
ApplHMMInfo;
```

Our first function will create and initialise a HMMSet. The function is passed a pointer to the above structure which is used to both specify the HMMSet and to store a reference to the newly created object.

```
int32 exHMMSetObjectInit(ApplHMMInfo *ahi)
{
   hapiHMMSetObject hmms;
   char8 *str;
   int32 i,n,fail=0;

   /* Create HMMSet and attach application info to hapiObject */
   fail = fail || ((ahi->hmms=hmms=hapiCreateHMMSetObject())==NULL);
   fail = fail || (hapiHMMSetSetAttachment(hmms,ahi)<0);
```

Note that the `ApplHMMInfo` structure is attached to the general purpose attachment hook in the HAPI object. This allows the application to recover the structure from a reference to the HAPI object.

The first three strings in the `ahi` structure correspond to the configuration parameters `HMMLIST`, `HMMDIR`, and `HMMEXT`. If, upon initialisation, they are `NULL` the actual settings need to be queried from the configuration, otherwise they override the values specified in the configuration file.

This can be accomplished by the following code

```
   if (ahi->list!=NULL)
      /* Need to set HMMSet list file */
      fail = fail || (hapiHMMSetOverrideList(hmms,ahi->list)==NULL);
   else {
      /* Need to query HMMSet list file */
      fail = fail || ((str=hapiHMMSetOverrideList(hmms,NULL))==NULL);
      if (!fail) ahi->list=StrAllocCopy(str);
   }

   if (ahi->dir!=NULL)
      /* Need to set HMMSet directory */
      fail = fail || (hapiHMMSetOverrideDir(hmms,ahi->dir)==NULL);
   else {
      /* Need to query HMMSet directory */
      fail = fail || ((str=hapiHMMSetOverrideDir(hmms,NULL))==NULL);
      if (!fail) ahi->dir=StrAllocCopy(str);
   }

   if (ahi->ext!=NULL)
      /* Need to set hmm filename extension */
      fail = fail || (hapiHMMSetOverrideExt(hmms,ahi->ext)==NULL);
   else {
      /* Need to query hmm filename extension */
      fail = fail || ((str=hapiHMMSetOverrideExt(hmms,NULL))==NULL);
      if (!fail) ahi->ext=StrAllocCopy(str);
   }
```

This code makes use of the following macro to allocate and initialise the strings that are queried from the configuration.

```
#define StrAllocCopy(str) (((str)==NULL) ? ((char*)NULL) : \
                           ((char*)strcpy((char*)malloc(strlen(str)+1),(str))))
```

This is necessary because all of the **HMMSetOverride** functions share a single buffer which holds returned values. In other words calls to **hapiHMMSetOverrideList**, **hapiHMMSetOverrideDir** and **hapiHMMSetOverrideExt** will all return the same value. However after each call the contents of the returned string buffer will be different.

Handling the list of MMF files is slightly more complex. The final member of the structure is an array of strings used to hold the list of MMF file names. Again if this is `NULL` the initialisation queries the configuration to determine the set of MMF files specified otherwise it clears the current set and adds the MMF files appearing in the `NULL` terminated array.

```
if (ahi->mmfs!=NULL) {
   /* Need to set HMMSet MMF files to load */
   fail = fail || (hapiHMMSetOverrideClearMMF(hmms)<0);
   for (i=0;ahi->mmfs[i]!=NULL;i++)
      /* Return value normally NULL so don't bother checking for error */
      hapiHMMSetOverrideAddMMF(hmms,ahi->mmfs[i]);
}
else {
   /* Need to query HMMSet MMF files to load */
   /* First find a NULL in the sequence of MMF files */
   while (hapiHMMSetOverrideAddMMF(hmms,NULL)!=NULL);
   /* Then count up total number of MMF files */
   n=0;
   while (hapiHMMSetOverrideAddMMF(hmms,NULL)!=NULL) n++;
   /* Allocate memory */
   ahi->mmfs=malloc(sizeof(char8*)*(n+1));

   /* Assign in the array */
   i=0;
   while ((str=hapiHMMSetOverrideAddMMF(hmms,NULL))!=NULL) {
      ahi->mmfs[i]=StrAllocCopy(str);
      i++;
   }
   ahi->mmfs[i]=NULL;
}
```

Finally once all the querying and overriding is complete the HMMSet can be loaded and initialised with the following

```
/* Load HMMSet and initialise */
fail = fail || (hapiHMMSetInitialise(hmms)<0);

/* We have one function not used yet so we need to demonstrate it */
fail = fail || (ahi!=hapiHMMSetGetAttachment(hmms));

if (fail && ahi->hmms!=NULL)
   hapiDeleteHMMSetObject(ahi->hmms),ahi->hmms=NULL;
```

```
   return(fail?-1:0);
}
```

Note that all the return values of all functions and the object attachment are checked to enuse correct operation and when an error occurs the HMMSet object is deleted and a failure code returned.

Once the application has finished using the HMMSet object it should be deleted and the following function illustrates how this can be done.

```
int32 exHMMSetObjectDel(ApplHMMInfo *ahi)
{
   int fail=0;
   fail = fail || (hapiDeleteHMMSetObject(ahi->hmms)<0);
   ahi->hmms=NULL;
   return(fail?-1:0);
}
```

Chapter 10

hapiTransformObject

A `hapiTransformObject` is responsible for handling all transformation and adaptation operations. The transforms can be applied to the `HMMSetObject` to adapt the acoustic HMMSet models to a particular speaker, a new environment or both. In general, the outcome of this transformation is faster, more accurate recognition performance. The transform object can be used to adapt an HMMSet with a previously estimated set of transforms, or create a set of transforms to adapt the HMMSet given some new speech data. Each transform object is uniquely associated with a single HMMSet object, with various API facilities for creation, loading transform sets, transform set application (to the HMMSet), adaptation (in online or offline, supervised or unsupervised modes) deletion, and transformation set and HMMSet saving (to disk).

10.1 Transforms and adaptation

The definitions of the transforms are stored off-line in files on disk. Typically these are stored in transform model files (TMFs). Many TMFs can be applied to the same HMMSet, but each TMF can only be applied to only one particular HMMSet. As an example, a transform set used to transform a set of monophone HMMs cannot be used to transform a more complex triphone HMM set. Once a transformation has been calculated, it can either be saved in a TMF or the transformed HMM set can be saved in a master model file[1] (MMF) instead. A TMF can also be saved together with the statistics necessary to allow further adaptation to continue from this saved point, rather than throwing away all the adaptation data seen so far and starting afresh.

One possible use of multiple TMFs and a single HMMSet could be to transform a speaker independent model[2] using an accompanying repository containing individual user transforms. A user could then log on to the recognition system and the HMMSet would be automatically transformed using the TMF associated with the user. As this transformed HMMSet would better represent the characteristics of a user's speech, recognition would be faster and more accurate.

There are two modes that the transform object can operate under. In the first mode there is *no adaptation* – a TMF can be loaded and applied to/removed from

[1] A master model file or MMF is the file format used within HTK for saving HMMSet models.

[2] This kind of acoustic model is trained on a variety of speakers, making it independent of the user, unlike a speaker dependent system.

an HMMSet object. In the second mode the transform object provides adaptation functionality. Several categories of adaptation are available:-

Supervised
A full transcription of the speech is available.

Unsupervised
A transcription is not available. In this case it is usual to use the recogniser to generate a hypothesis, which is then used as the transcription in the adaptation process.

Online
The speech application can adapt to the new speaker and/or environment as it recognises/gathers the speech, by generating a new transformation incrementally or whenever the user requests an update.

Offline
When a block of data from a particular user is already available, further refinement is possible by iterating (repeating) the model adaptation process.

Further sections introduce some example code together with descriptions to highlight the major components of the transform object and its usage.

10.2 Introduction to the transform object

Example code illustrates how the transform object is typically used within an application program. The following simple data structure `ApplTrInfo` is defined and used to hold information related to the transform set, (and the lattice objects necessary for the alignment process[3] in adaptation). This structure will be used both to specify the transform set, as well as to pass information between functions in the example code.

10.2.1 Application initialisation

```
/* ------------------  transform example code --------------------- */

typedef struct appltrinfo {

   hapiTransformObject trans;     /* Reference to the transform object */
   hapiTransformType   type;      /* Type of transform */
   hapiTransformType   applyType; /* Type of transform to apply
                                     to the HMMSet object */
   char8 *tmf;                    /* input/load TMF file name */
   char8 *outtmf;                 /* output/save TMF file name */
   char8 *outmmf;                 /* output/save an HMMSet MMF file */
   char8 *uid;                    /* User identifier */
   char8 *uname;                  /* Full user name */
   int32 adapt;                   /* Flag for adaptation */

   /* objects required for adaptation */
```

[3] Essentially the alignment process associates each incoming speech frame with a particular model in the HMMSet, based on the transcription (or best hypothesis for unsupervised adaptation).

```
   hapiLatObject lat;             /* Reference to the lattice object */
   int32 adapted;                 /* Flag indicating whether the
                                     models have been transformed */

}
ApplTrInfo;
```

After having first created and initialised and loaded an HMMSet, a transform object can either be created and loaded in a single load call or it can be created and initialised ready for adaptation from scratch. This is demonstrated by the function, `exTransformObjectInit()`, which shows the typical HAPI calls used during the initialisation process. It also shows how various transform information settings can be overridden using the function `hapiTransformOverrideInfo()`.

```
int32 exTransformObjectInit(ApplTrInfo *ati, hapiHMMSetObject hmms)
{
   int32 fail=0;

   /* set the transform type */
   if (ati->adapt) {
      ati->type = (HAPI_TRANS_dynamic+HAPI_TRANS_use_variance+
                   HAPI_TRANS_adapt_sil);
   }
   else {
      ati->type = (HAPI_TRANS_use_variance+HAPI_TRANS_adapt_sil);
   }

   /* create a new transform object */
   if (ati->tmf != NULL) {
      /* load a transform object from file */
      fail = fail || ((ati->trans = hapiTransformLoad(hmms,
                                                      ati->tmf,
                                                      ati->type)) == NULL);

      /* Provide some feedback to the user */
      ati->uname = hapiTransformOverrideInfo(ati->trans, HAPI_TINFO_UNAME, NULL);
      ati->uid   = hapiTransformOverrideInfo(ati->trans, HAPI_TINFO_UID, NULL);

      if (ati->tmf != NULL) {
         printf("Transforms for %s (%s) loaded from file %s\n",
                ati->uname, ati->uid, ati->tmf);
         fflush(stdout);
      }
   }
   else {
      /* create and initialise transform object from scratch */
      fail = fail || ((ahi->trans = hapiCreateTransformObject(hmms,
                                                              ati->type)) == NULL);
      fail = fail || (hapiTransformInitialise(ati->trans));
   }
   fail = fail || (hapiTransformSetAttachment(ati->trans, ati) < 0);

   /* initialise the adapted flag */
   ati->adapted = false;
```

hapiTransformType	HAPI Functions		
	`CreateTransform`	`TransformLoad`	`TransformApply`
`HAPI_TRANS_dynamic`	—	X	—
`HAPI_TRANS_use_variance`	X	X	X
`HAPI_TRANS_adapt_sil`	X	X	X
`HAPI_TRANS_global`	—	—	X
`HAPI_TRANS_class`	—	—	X
`HAPI_TRANS_full`	—	—	X

Table 10.1: Functional use of `hapiTransformType`.

```
   return(fail?-1:0);
}
```

A description of the settings shown in the above example code for `ati->type` can be found in section 10.2.2. Once the application has finished using the transform object, the transform object should be deleted. The following function illustrates how this can be accomplished.

```
int32 exTransformObjectDel(ApplHMMInfo *ati)
{
   int fail=0;
   fail = fail || (hapiDeleteTransformObject(ati->trans)<0);
   ati->trans=NULL;
   return(fail?-1:0);
}
```

10.2.2 Transform types

From the example application structure `ApplTrInfo` there are two functionally important `hapiTransformType` members – `type` and `applyType`. The initialisation function `exTransformObjectInit` demonstrates some of the settings for `type` that are used by the HAPI functions `hapiTransformLoad()` and `hapiCreateTransformObject()`. Another function responsible for (possibly updating and) applying a transform called `hapiTransformApply()` also has a parameter of type `hapiTransformType`, and this function is demonstrated later on. By varying the settings of `type` and `applyType`, the developer gains control of subtle functional differences within the transform object. The full range of meaningful settings is shown in table 10.1[4].

The meaningful switches are marked with crosses, while the unused switches are marked with dashes. The different transform types are described below. Although using variance and silence transforms is not the default, the developer is strongly urged to use variance transforms and silence transforms for increased speed and accuracy of the resulting transformed HMMSet. Further information about `hapiTransformApply()` functionality is given in section 10.3.

dynamic

This specifies that adaptation is to take place, and that the transform will

[4]The function names in the table are shortened for compactness.

be updated (if necessary). When calling `hapiTransformLoad()`, setting the dynamic switch allows statistics stored in the TMF to be loaded (if they are present). This enables the adaptation process to continue from a previously saved point. When calling the function `hapiCreateTransformObject()` the dynamic switch is implicitly set.

variance
By default, the transform object creates/loads/applies (and updates as part of the `hapiTransformApply()` function) the mean transforms. This setting also creates/loads/applies the variance transformation.

silence
This setting also creates/loads/applies a transformation for silence.

global
A single transformation is applied globally to the parameters of the HMMSet. (The linear transform is also applied to silence, whether the apply type silence switch is set or not).

class
Similar HMMSet parameters are grouped together to form a class. The model set may contain many different classes, with each parameter belonging only to one class. In the class-based approach, each class has a linear transformation associated with it. Hence, a whole series of class-based linear transforms is applied to the model set, with each transform being responsible for transforming a part of the model set. In general this transformation results in a model set that is superior, in terms of speed and accuracy, to the globally transformed model set. This is also the default application type.

full
This transformation is only available when adaptation is being performed, and is not stored in the TMF. Further information on the full transformation can be found in section 10.4.2.

10.3 Transform object without further adaptation

This section highlights when and how a system developer might use the transform object without performing any adaptation. No transform specific configuration parameters are necessary for this mode of operation. As an example, a system could contain a core speaker independent HMMSet and a large group of speaker specific transforms. Figure 10.1 shows the operation of such a system.

Since this application does not make any extensive use of the contents of the transform object directly, the only tasks the object will need to perform is to specify and create the object, transform the HMMSet and then delete the object when it is no longer required. Note that it is the HMMSet object that is changed when a transform is applied. It is this object which is then passed to other HAPI functions that require HMM definitions. Once a transform set has been loaded, it can be applied to an HMMSet object. How the transform object is applied to the HMMSet can be controlled by the `hapiTransformType` argument to the application function. The example below applies a class-based transformation to the HMMSet, also applying a variance transformation and transforming the silence parameters in the HMMSet.

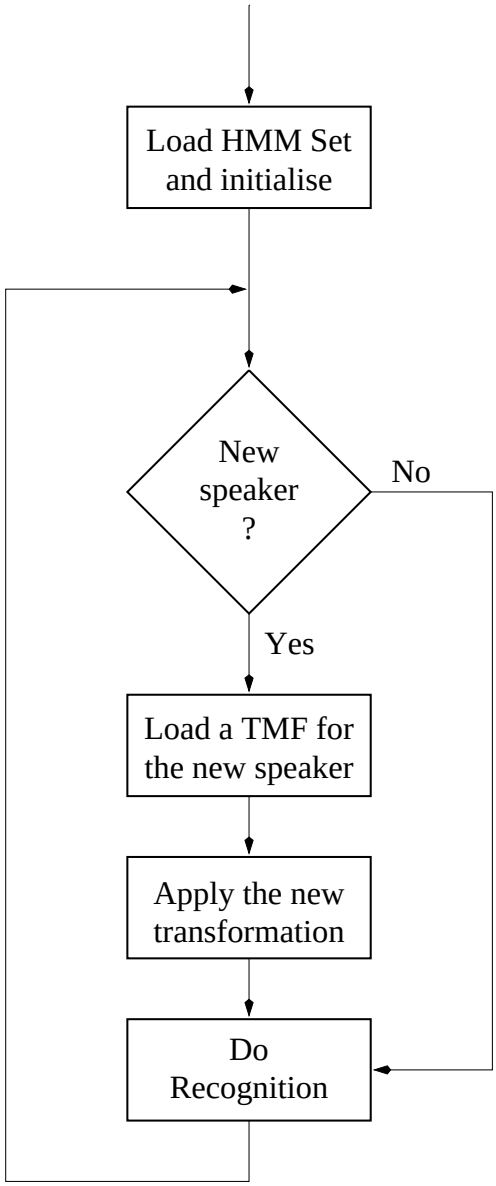

Fig. 10.1 Transform operational order

```
ati->applyType = (HAPI_TRANS_class + HAPI_TRANS_use_variance +
                  HAPI_TRANS_adapt_sil);
fail = fail || (hapiTransformApply(ati->trans, ati->applyType)<0);
```

One final operation allows the developer to transform a **transformed** HMMSet back to its original state before the current transform object was applied. Since this reversal is performed by estimating an inverse transform (and applying this), the HMMSet reversal is dependent on the current transform object. Once the reversion

is complete, the developer can make use of the speaker-independent model set, or apply another transform to it. Note that it is not necessary to revert a transformed model set before applying a new transform or deleting a transform, since the reversion process is handled automatically by HAPI.

```
fail = fail || (hapiTransformRevert(ati->trans) < 0);
```

10.4 Adaptation

Adaptation is the process by which some speech adaptation data (for a user and/or environment) is used to estimate a transformation which when applied to the model set adapts it so that is more representative of the speech adaptation data (and hence the user and/or environment). It is important for the system developer to be able to distinguish between estimating a transform, and adapting the model set by applying the newly estimated transform.

Within HAPI the whole process of estimating the transforms falls into two distinct stages:-

1. The first stage accumulates the statistics necessary for generating a new set of transforms. Computationally this stage is relatively inexpensive, normally taking around a second to process 20 seconds of speech. Consequently it's normally performed after processing[5] each utterance.

2. The second stage uses the accumulated statistics to generate a new set of transforms. This stage can be performed at any time, as long as some new statistics have been gathered in stage one. Hence new transforms can be estimated as often as every recognised sentence and as little as every session. When to update the transform set is entirely up to the system developer (or possibly under the user's control depending on the system design). This process is a little more compute intensive than stage one, taking a few seconds to calculate the transform parameters.

Configuration variables relevant to the transform object have no effect when adaptation is not occurring, as all transform configuration settings are stored in the TMF, and are automatically assigned by HAPI when the transforms are loaded from disk. However some configuration settings are necessary for the *adaptation* mode. These configuration settings are necessary for aligning the incoming speech frames to models in the HMMSet, during the statistics accumulation stage. Once accumulated, the statistics can be used at any stage to generate the transforms (which also requires a couple of configuration settings). The transform object relevant configuration settings are shown in table 10.2.

Before some example code is shown, the following is a typical set of entries one might find in the configuration file with respect to adaptation.

```
# Transform object adaptation parameters
HAPI:   ALIGNBEAM    = 250.0
HAPI:   INITBEAMADPT = 250.0
```

[5] In supervised adaptation processing is merely reading in the data, whereas in unsupervised adaptation, processing requires recognition of the utterance to generate a transcription.

Module	Name	Default	Description
HAPI	ALIGNBEAM	400.0	General beam width controlling which hypotheses are pursued during the alignment of speech to models stage. Larger values can prevent certain types of errors at the expense of greater computation.
	INITBEAMADPT	400.0	Initial backward (beta) pruning beam width for the frame to state alignment. Larger values can prevent pruning errors or utterance alignment rejection. Rejection can also be avoided by using an incremental threshold (INCBEAMADPT and LIMBEAMADPT).
	INCBEAMADPT	0.0	If a pruning error occurs, then the initial backward (beta) pruning beam width is increased incrementally by the INCBEAMADPT setting, and the utterance is reprocessed.
	LIMBEAMADPT	400.0	Repeated pruning errors cause the backward beam to be increased, and this continues until the upper beam limit LIMBEAMADPT is reached.
	MINFRWDADPT	10.0	The forward (alpha) pruning beam width.
	FULLADPTSCALE	12.0	Scaling used between the current model parameters and the adaptation data for "full" adaptation. The larger the number is, the less influence the adaptation material has when adjusting the model parameters. It is advisable to choose a number greater than 5.0.
HADAPT	BLOCKS	1	Calculate and store the transforms in block diagonal format. For static, delta and acceleration parameters in the feature vector, it is generally best to use 3 blocks, since this requires less computation and storage space, with no significant loss in accuracy.
	OCCTHRESH	700.0	The class occupation threshold controls the generation of transforms for particular classes (groupings of the model parameters). The lower this number, the more transforms are generated. However this can lead to poorly estimated transforms due to insufficient data.

Table 10.2: Configuration settings relevant to the `hapiTransformObject`.

```
HAPI:   INCBEAMADPT  = 250.0
HAPI:   LIMBEAMADPT  = 750.1
HAPI:   MINFRWDADPT  = 10.0
HAPI:   FULLADPTSCALE= 10.0
HADAPT: BLOCKS       = 3
HADAPT: OCCTHRESH    = 1000.0
```

The first five parameters control the alignment used to accumulate the statistics for the transformation calculation (or stage one from above). The next two are concerned with generating the transform set. Their values are only important when creating a transform object, as when loading a transform, the values stored on disk (in the TMF) are used.

10.4.1 A class-based example

Returning now to the example code, we show how a system developer can perform online incremental (unsupervised) adaptation using the transform object with a class-based transform. At this stage, it is assumed that the transform object has been initialised by a function similar to `exTransformObjectInit()`. Here, the results object is used to provide the transcription of the utterance for the purpose of aligning the incoming speech to the models. This example extends the frame-by-frame recognition example function `exRecognise()` shown in chapter 15.

```
int32 exRecogniseAdapt(ApplRecInfo *ari,ApplSrcInfo *asi,AppTransInfo *ati)
{
   hapiTime obsAdv;
   int32 fail=0,nFr=0,n;
   char8 buf[2048];

   fail = fail || (hapiCoderPrepare(asi->coder)<0);
   fail = fail ||
      IS_HAPI_NAN(obsAdv=hapiCoderObservationWindowAdvance(asi->coder));
   fail = fail || (hapiRecPrepare(ari->rec,asi->coder,NULL)<0);

   /* Could wait here before starting audio input */
   fail = fail || (hapiSourceStart(asi->src)<0);

   while((st=hapiRecRemaining(ari->rec))!=HAPI_REM_done) {
      if (fail = fail || (st==HAPI_REM_error)) break;
      fail = fail || ((n=hapiRecProcess(ari->rec,ari->cbi,ari->inter))<0);
      /* Process intermediate results - eg */
      hapiResTraceInfo(ari->inter,HAPI_TF_frame+HAPI_TF_word+
                       HAPI_TF_average+HAPI_TF_active,81,buf);
      printf("%s\n",buf); fflush(stdout);
      fail = fail || (hapiResClear(ari->inter)<0); nFr+=n;
   }
   fail = fail || (hapiRecComplete(ari->rec,ari->final)<0);

   /* Process final results ??? */
   hapiResTraceInfo(ari->final,HAPI_TF_frame+HAPI_TF_word+HAPI_TF_average+
                    HAPI_TF_active,2048,buf);
   printf("%s\n\n",str);  fflush(stdout);
```

```
   /* ---------  NEW  FOR ADAPTATION  ---------- */

   /* Now that final results have been processed, prepare to
      accumulate adaptation statistics */
   nFr = 0;

   /* First generate  a lattice from the results */
   fail = fail || (ati->lat=hapiResLatGenerate(ari->final,NULL);

   /* Accumulate transform adaptation statistics */
   nFr = hapiTransformAccumulate(ati->trans, ati->lat, ari->coder);
   if (nFr<0)
      fail = 1;

   fail = fail || (hapiDeleteLatObject(ati->lat));

   /* ---------------------------------------- */

   fail = fail || (hapiResClear(ari->final)<0);
   fail = fail || (hapiCoderComplete(asi->coder)<0);

   return(fail?-1:ret);
}
```

The function `exRecogniseAdapt()` checks the number of frames returned from the call to `hapiTransformAccumulate`. If this value is zero then the alignment procedure failed (i.e. alignment pruning beam widths are too small or the transcription provided in the lattice object does not adequately match the speech data), and the statistics for adaptation for this utterance are not accumulated. If the value is less than zero then the HAPI function has failed terminally. Once `exRecogniseAdapt()` has been called, the value of the integer parameter `update` can be checked and the new transform can be generated at anytime.

Note also that the results object is used to generate a results lattice which is used for the alignment process in the statistics accumulation stage. However if the transcription for the utterance is actually known (i.e. the adaptation is supervised) then a word graph lattice can be generated with simple calls to a series of lattice object functions. This lattice object can then be passed to the `hapiTransformAccumulate()` function.

The following example function is called whenever the models are to be updated using the newly collected statistics.

```
/* update the model set -- prior to calling this function, the
   apply type, (ati->applyType) must be set. */
void exUpdateModelSet(AppTransInfo *ati) {

   int32 fail=0;

   /* now calculate and apply the new transform */
   fail = fail || (hapiTransformApply(ati->trans, ati->applyType);

   return(fail?-1:ret);
}
```

The call to `hapiTransformApply()` in adaptation mode (i.e. the `hapiTransformType` at transform object creation/loading has the `HAPI_TRAN_dynamic` value set) updates the transform set based on the collected statistics to date, and applies this new transform set to the loaded HMMSet[6]. As long as some new statistics have been accumulated, a new transform can be generated at any time. If no new statistics have been accumulated, then the current transform is simply applied.

The new transform set can also be saved at any time, as can the transformed model set in an MMF file.

```
/* save with statistics and in binary format */
hapiTransformSave(ati->trans, ati->outtmf, "SB");

/* save the transformed model set */
if (ati->adapted)
   hapiTransformSave(ati->trans, ati->outmmf, "MB");
```

Note that if continued adaptation from this session is required, then the transform must be saved with statistics. Of course it is possible to continue to adapt a saved (transformed) model set. The type of transforms used and the models (TMFs or MMFs) saved will depend on the application.

10.4.2 Using a full transform

Previous sections have defined and described the global and class-based transforms and their usage. This section compares and contrasts the full transform with the class-based transform.

The full transform, in effect, transforms every parameter in the HMMSet. This transformation is dependent on the data collected for adaptation and the HAPI configuration parameter `FULLADPTSCALE`. Once this transformation has been made, and unlike the global or class-based transformations, there is no way to recover the original (pre-transformed) HMMSet. Hence, the only way to save the transformation is to save the model set itself. The full transformation is less effective than the global or class-based, when a limited amount of adaptation speech is available. However, as the amount of adaptation data increases, the full transformation tends to exceed the performance of the class-based transformation. Approximately 20 minutes of speech data is typically sufficient to estimate this type of transform. The table below compares the class-based and full transforms, and can be used as a rough guideline for the developer. It is up to the developer to decide which transform apply type is most suitable for an application.

Another important feature is that the class-based and full transformation methods can be combined (by just setting both for the transform apply type). This transformation normally results in superior performance when compared to solely using either the class-based or full transformations, no matter how much adaptation material is used. However since a full transformation is made, the developer must still save a full model set.

[6] If required to, `hapiTransformApply()` automatically transforms the HMMSet back to its original (pre-transformed) state, before applying a new transformation

Feature	Class	Full
Adaptation performance after a few minutes of data	Fast adaptation is possible, and improvement in terms of speed and accuracy is noticeable.	Adaptation likely to be ineffective.
Adaptation performance after twenty minutes of data	Improvement in terms of speed and accuracy is very noticeable.	Improvement in terms of speed and accuracy is very noticeable and comparable with class-based.
Adaptation performance after an hour or more of data	Improvement over transforms estimated from twenty minutes tends not to be especially evident. Adaptation process is likely to have saturated.	Adapted model performance continues to improve, and is likely to be better than the classed-based method.
Saving models	There are options to save a TMF (which is considerably smaller than an MMF) or an MMF. The TMF can also be stored with statistics, so that adaptation can resume from a saved point.	It is only possible to save the MMF. Adaptation can be continued from a newly saved MMF, but care must be taken when setting the `FULLADPTSCALE` configuration parameter so as not to discount previously seen data that was used in the loaded MMF.
Flexibility	For a new speaker it is easy to apply/revert a transform, and this process is fast, and can be done many times.	For a new speaker a whole new model set must be loaded, which is a little more time consuming than class-based.

One final point is that for incremental full adaptation, it is advisable to use block adaptation only – i.e. transform the model set only at the end of the session (by making the call to `hapiTransformApply()` with the apply type parameter set to `HAPI_TRANS_full`) and just before saving the new model set. At the next use of the recogniser, the new model should be used.

Chapter 11

hapiSource/CoderObject

Together the source and coder objects manage the acquisition and signal processing of the input audio data and provide the recogniser in HAPI with a source of coded vectors (observations) to process. Since the two objects can only be used together and are internally heavily linked they are described together in this chapter.

11.1 hapiSourceObject

The source object provides the application and the rest of the HAPI library with an interface to different sources of speech data. Since the underlying form of the source can vary considerably, the type of source (for example files or direct audio) alters the way in which the application interacts with the user. For example processing a series of files specified in a script can be accomplished with the minimum of interaction with the user. However, if the source is live audio the user will probably want to indicate when the input audio should be processed as well as supplying the data itself.

The types of input source supported are listed in table 11.1.

11.2 hapiCoderObject

The coder object takes the raw data from the source (which will normally just be a digitised waveform but could be partly coded) and converts it into the required type of observation. In addition to this the coder object can run a speech detector on the data in order to delimit the start and end of the data to be passed on to the recogniser.

This coding process is potentially complex.

A typical conversion will start by splitting the audio waveform into discrete windows, then each window will be filtered, analysed to find some form of short term spectrum. This spectrum will then be smoothed, decorrelated and rate of change terms added to give the final observation. Obviously with so many processes involved there are many parameters which affect the signal processing stage. These parameters are controlled by the configuration variables listed below in table 11.2.

Some of the HPARM configuration parameters are not directly related to the signal processing that converts the waveform data into the correct type of observations. These are described in the table below.

Mode	Description
HAPI_SRC_file	Read data from a single file in the format specified in the configuration (which defaults to HTK).
HAPI_SRC_script	Read data from a series of files the names of which are read from a script file. Again the file format is specified in the configuration and defaults to HTK.
HAPI_SRC_haudio	Read data directly from the workstation's built in audio hardware. The use of the speech detector is controlled by the configuration.
HAPI_SRC_haudio_no_det	Read data directly from the workstation's built in audio hardware with the audio started and stopped explicitly by the application.
HAPI_SRC_haudio_with_det	Read data directly from the workstation's built in audio hardware using the speech detector to delimit utterances.

Table 11.1: Types of input source.

11.3 Speech detector

A speech detector can be used to produce a 'press then talk' interface to the recogniser. In this mode of operation the application indicates that the source should start collecting data but recognition processing does not begin until the speech detector detects the onset of speech. Once the onset of speech is found the utterance stops when a certain period of silence is detected (or the application explicitly terminates the utterance).

The automatic speech detector uses a two level algorithm which first classifies each frame of data as either speech or silence then applies a heuristic to determine where the start and end of each utterance occurs.

The standard detector classifies each frame as speech or silence based solely on the log energy of the signal. When this exceeds a threshold the frame is marked as speech otherwise it is silence. The threshold is made up of two components both of which can be set by configuration variables. One (SILENERGY) represents the mean energy level of silence and although this can be set via the configuration would more normally be measured by one of the **Calibrate** functions described later. The other (SPEECHTHRESH) is the threshold above this silence level after which frames are classified as speech.

Once each frame has been classified as speech or silence they are grouped into windows consisting of SPCSEQCOUNT consecutive frames. When the number of frames marked as silence within each window falls below a glitch count the whole window is classed as speech. Two separate glitch counts are used, SPCGLCHCOUNT before speech onset is detected and SILGLCHCOUNT whilst searching for the end of the utterance. This allows the algorithm to take account of the tendency for the end of an utterance to be somewhat quieter than the beginning.

Module	Name	Default	Description
HPARM	TARGETKIND	ANON	Parameter kind of target
	TARGETFORMAT	HTK	Format of target
	TARGETRATE		Frame advance rate (in 100ns units)
	SOURCEKIND	ANON	Parameter kind of target
	SOURCEFORMAT	HTK	Format of source
	SOURCERATE		Source sample rate (in 100ns units)
	ZMEANSOURCE	F	Zero mean source waveform before coding
	WINDOWSIZE	256000.0	Analysis window size in 100ns units
	USEHAMMING	T	Use a Hamming window
	PREEMCOEF	0.97	Set pre-emphasis coefficient
	USEPOWER	F	Use power not magnitude in fbank analysis
	NUMCHANS	20	Number of filterbank channels
	LOFREQ		Low frequency cut-off in fbank analysis
	HIFREQ		High frequency cut-off in fbank analysis
	LPCORDER	12	Order of lpc analysis
	CEPLIFTER	22	Cepstral liftering coefficient
	NUMCEPS	12	Number of cepstral parameters
	RAWENERGY	T	Use raw energy
	ENORMALISE	T	Normalise log energy
	ESCALE	0.1	Scale for log energy
	SILFLOOR	50.0	Normalised energy silence floor in dBs
	DELTAWINDOW	2	Delta window size
	ACCWINDOW	2	Acceleration window size
	SIMPLEDIFFS	F	Use simple differences for delta calculations
	VQTABLE		Name of VQ table
	V1COMPAT	F	HTK V1 compatibility setting

Table 11.2: Parameterisation configuration parameters.

Finally the top level heuristic is used to determine the start and end of the utterance. The heuristic defines the start of speech as the beginning of the first window classified as speech. The actual start of the processed utterance is SILMARGIN frames before the detected start of speech to ensure that when the speech detector triggers slightly late the recognition accuracy is not affected. Once the start of the utterance has been found the detector searches for SILSEQCOUNT windows all classified as silence and sets the end of speech to be the end of the last window classified as speech. Once again the processed utterance is extended SILMARGIN frames to ensure that if the silence detector has triggered slightly early the whole of the speech is processed by the recogniser.

Fig.11.1 shows an example of this endpointing process. The waveform data is first classified as speech or silence at the frame and then the window level before finally the start and end of the utterance are marked. In the example audio input starts at point A (and is stopped automatically at point H). The start of speech, C, occurs when a window of SPCSEQCOUNT frames are classified as speech and the start of the utterance occurs SILMARGIN frames earlier at B. The period of silence from D to E is not marked as the end of the utterance because it is shorter than SILSEQCOUNT. However after

Module	Name	Default	Description
HWAVE	NSAMPLES		Num samples in ALIEN file input via a pipe
	HEADERSIZE		Size of header in an ALIEN file
	STEREOMODE		Select channel: RIGHT or LEFT
HPARM	SAVECOMPRESSED	F	Save the output file in compressed form
	SAVEWITHCRC	T	Attach a checksum to output parameter file
	ADDDITHER	0.0	Level of noise added to input signal. For some types of artificial data adding a small amount of noise prevents numeric overflows that would otherwise occur with some coding schemes. If this value is positive the noise signal added is the same every time (ensuring that the same file always gives exactly the same results) if negative the noise is random and the same file may produce slightly different results.

Table 11.3: Additional configuration parameters.

point F no more windows are classified as speech (although a few frames are) and so this is marked as the end of speech with the end of the utterance extended to G/

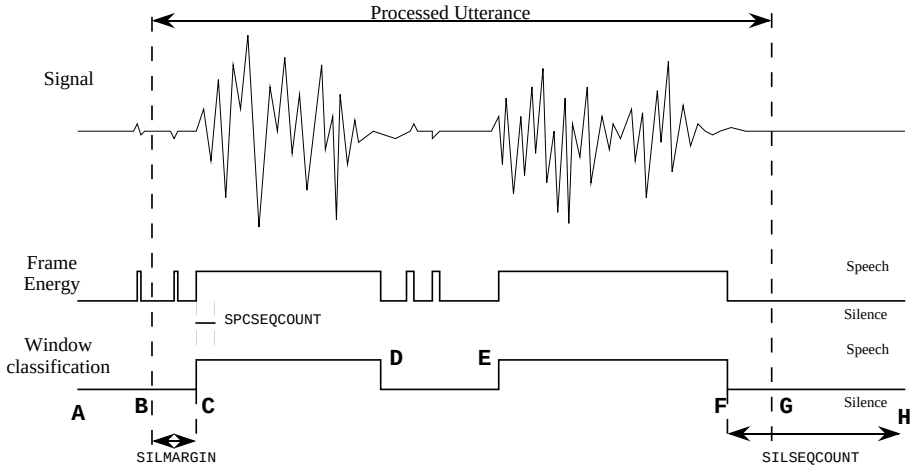

Fig. 11.1 Endpointer Parameters

Normally the threshold SILENERGY is set every session in an explicit calibration procedure in which the user is requested to supply examples of speech and silence. When this is not possible either an estimate of the silence level needs to be supplied by the configuration parameter or a different method of classifying frames as speech or silence needs to be used. (For example a Gaussian classifier could be used to class each observation as speech or silence). HAPI also provides an automatic silence calibration facility to obtain an estimate for SILENERGY. This works by using the first

SELFCALSILDET frames of the first utterance of a new session to calibrate the silence detection. (A new session can be started with a call to hapiCoderResetSession.) The developer is advised to use at least 100 frames for the SELFCALSILDET configuration setting. When calibration is not being performed automatically, two functions are provided to measure signal levels from a particular source and get an estimate for SILENERGY.

The first function hapiSourceCalibrateSpeech expects the user to speak normally for a few seconds thus allowing the measurement of mean and peak speech as well as background noise levels. Obviously it will be necessary for the application to prompt the user to ensure that they speak when required and so the function can be used initially in a query mode just to determine if the measurement is necessary. (If the speech detector is not being used or an alternative detector that does not need calibrating is used then this measurement will not be needed).

The second function hapiSourceCalibrateSilence is similar but expects a few seconds of silence and will only measure background noise levels.

Since only one of these two functions will normally be required the speech calibration is often preferred since it provides a better check of the audio input.

For example the following function first determines if a speech level measurement is required (taking it if necessary) before seeing if the silence level calibration is still needed. For the majority of sources it will not be and the second measurement will be skipped.

```
int exSourceCoderObjectCalibrate(ApplSrcInfo *asi, int wait)
{
   float32 cal[HAPI_CAL_size];
   int r,fail=0;

   fail = fail || ((r=hapiSourceCalibrateSpeech(asi->src,NULL))<0);
   if (r>0) { /* Measurement is required */
      if (wait>0) {
         printf("Press return to measure speech levels\n");
         getc(stdin);
      }
      printf("Please speak for 5 seconds\n");
      fail = fail || (hapiSourceCalibrateSpeech(asi->src,cal)<0);
      printf("   Mean speech level %.2f dB (%.1f%%), silence %.2fdB\n\n",
             cal[HAPI_CAL_speech],cal[HAPI_CAL_peak]*100.0,
             cal[HAPI_CAL_background]);
   }
   fail = fail || ((r=hapiSourceCalibrateSilence(asi->src,NULL))<0);
   if (r>0) { /* Measurement is required */
      if (wait>0) {
         printf("Press return to measure silence levels\n");
         getc(stdin);
      }
      printf("Please remain silent for 3 seconds\n");
      fail = fail || (hapiSourceCalibrateSilence(asi->src,cal)<0);
      printf("  Mean silence level %.2f dB\n\n",
             cal[HAPI_CAL_background]);
   }
   return(fail?-1:0);
}
```

The table below describes all the configuration variables associated with the speech

/ silence detector.

Module	Name	Default	Description
HParm	USESILDET	F	Use silence detector to delimit speech boundaries
	SPEECHTHRESH	9.0	Threshold for speech above silence level (in dB)
	SILENERGY		Average background noise level (in dB) - will normally be measured rather than supplied in configuration.
	SPCSEQCOUNT	10	Window over which speech/silence decision reached
	SPCGLCHCOUNT	0	Maximum number of frames marked as silence in window which is classified as speech whilst expecting start of speech.
	SILSEQCOUNT	100	Number of frames classified as silence needed to mark end of utterance.
	SILGLCHCOUNT	2	Maximum number of frames marked as silence in window which is classified as speech whilst expecting silence.
	SILMARGIN	40	Number of extra frames included before and after start and end of speech marks from the speech/silence detector.
	SELFCALSILDET	0	Number of frames at the beginning of an utterance that are used to calibrate the silence detector automatically.
	MEASURESIL	F	Measure background noise level.
	OUTSILWARN	F	Print a warning message to `stdout` before measuring audio levels

11.4 Cepstral Mean Normalisation

A common problem with speech recognition systems is that the characteristics of the channel may vary from one session to the next. One method used to minimise the effect of these differences on recogniser performance is Cepstral Mean Normalisation (CMN). Essentially CMN involves subtracting the cepstral mean, calculated across the utterance, from each frame.

Often this is applied over the whole of the utterance with the mean value of the cepstral coefficients calculated across the whole utterance then subtracted from each frame. This is not suitable for use with live recognition as no frames can be made available for processing before the whole utterance has been read in.

Consequently a simple approximation to this (referred to as dynamic CMN) has been developed for use with live audio input. This uses an IIR filter to provide a running average of the cepstral coefficients. The initial value for the average is read from a file on disk (with the file name specified by the `DEFCEPMEANVEC` configuration parameter) and the value will normally be calculated from some representative data (for example the training data). When this initial estimate is believed to be somewhat inaccurate its effect can be minimised by updating the filter before using the average. The configuration parameter `CEPMEANWAIT` specifies the number of frames to

be processed by the IIR filter at the start of each session before the average is used to normalise the observations. This increases the latency of the recogniser because no frames will be processed until the first `CEPMEANWAIT` have been captured and used to update the mean vector. The configuration variables affecting the operation of CMN are described in the table below.

Module	Name	Default	Description
HPARM	`TARGETKIND`	`ANON`	Parameter kind of target needs to include `_Z` to specify (static or dynamic) cepstral mean normalisation.
	`CEPMEANTC`	`0.0`	Filter parameter controlling calculation of running cepstral average. If set to `0.0` use static CMN (which will not work with live audio input). $\mu_{t+1} =$ `CEPMEANTC` $* \mu_t + x_t$.
	`CEPMEANWAIT`	`0`	Number of frames to average before starting running average. Larger values may give better performance when the initial cepstral mean is inaccurate but will introduce latency.
	`CEPMEANCNT`	`0`	Number of files to process before resetting cepstral mean to default. If `0` only do this when explicitly told to.
	`DEFCEPMEANVEC`		File name of initial default cepstral mean for running average CMN. File consists of number of components followed by vector.

11.5 Channels and Sessions

Despite the relatively small number of input sources the actual data can come from a variety of places. Over the telephone, from a close talking head mounted microphone or from a cell phone microphone in a car. In some cases a single application will want to recognise data from these different sources each of which needs different signal processing. To support such applications the coder implements *channels*.

Each coder can be connected through a particular channel to any type of source. Each channel can specify a different way in which the audio waveform can be converted into observations.

For example a configuration such as

```
HPARM:    TARGETKIND = MFCC_0_D_A
CHANNEL:  TARGETKIND = LPC_E_D_A
```

can be used to specify two completely different data types that will be used. The default channel will produce `MFCC` observations whilst a coder using channel `CHANNEL` will generate `LPC` ones.

The process of generating the observations from the acoustic signal can be modified to account for certain environmental effects.

For example an averaging process in the cepstral domain can be used to provide limited channel adaptation. However the calculated values should be flushed when

the channel changes. Such changes (for example when a new speaker starts using the system) should be indicated to allow the parameters to be reset to their initial values by calling

```
if (hapiCoderResetSession(asi->coder)<0) return;
```

11.6 Usage

This section includes example code illustrating how to use the source and coder objects within an application program.

The example code fragment below defines a simple structure will be used to hold information related to the source and coder objects. This structure will be used both to specify the objects as well as to pass information between functions in our example code.

```
/* --------------------  Source & coder example code -------------------- */

typedef struct applsrcinfo {
   hapiSourceType type;   /* Type of input source */
   hapiSourceObject src;  /* Reference to HAPI source */
   hapiCoderObject coder; /* Reference to HAPI coder */
   hapiTime sampPeriod;   /* Input source sample rate */
   hapiSourceObject play; /* Reference to HAPI source for playback */
   hapiObservation obs;   /* Reference to observation buffer */
   char8 *chan;           /* HParm input channel */
   char8 *file;           /* Input speech file name */
   char8 *script;         /* Script file name */
   hapiTime obsAdv;       /* Coder frame rate */
   char8 *parmKind;       /* Observation parameter kind */
}
ApplSrcInfo;
```

The example function which creates and initialises the source and coder objects is a little more complicated than the equivalent HMMSet object function (See Section 9.2). This is due to the different types of source and the fact that a single function is initialising three objects; a `hapiSourceObject` for reading audio data from the chosen source, a `hapiCoderObject` connected to that source and a second `hapiSourceObject` for playing audio to the user.

```
int exSourceCoderObjectInit(ApplSrcInfo *asi)
{
   hapiSourceObject src;
   char8 *str;
   int32 fail=0;

   if (asi->type==HAPI_SRC_none) {
      if (asi->file!=NULL) asi->type=HAPI_SRC_file;
      else if (asi->script!=NULL) asi->type=HAPI_SRC_script;
      else asi->type=HAPI_SRC_haudio_with_det;
   }
   fail = fail || ((asi->src=src=hapiCreateSourceObject(asi->type))==NULL);
   fail = fail || (hapiSourceSetAttachment(asi->src,asi)<0);
   switch(asi->type) {
```

```
    case HAPI_SRC_file:
      if (asi->file!=NULL)
         fail = fail || (hapiSourceOverrideFilename(asi->src,asi->file)==NULL);
      else {
         fail = fail || ((str=hapiSourceOverrideFilename(asi->src,NULL))==NULL);
         if (!fail) asi->file=StrAllocCopy(str);
      }
      break;
    case HAPI_SRC_script:
       if (asi->script!=NULL)
          fail = fail || (hapiSourceOverrideFilename(asi->src,asi->script)==NULL);
       else {
          fail = fail || ((str=hapiSourceOverrideFilename(asi->src,NULL))==NULL);
          if (!fail) asi->script=StrAllocCopy(str);
       }
       break;
    case HAPI_SRC_haudio:
    case HAPI_SRC_haudio_with_det:
    case HAPI_SRC_haudio_no_det:
    default:
       break;
   }
   fail = fail || (hapiSourceInitialise(asi->src)<0);

   fail = fail || ((asi->coder=hapiCreateCoderObject(asi->src))==NULL);
   fail = fail || (hapiCoderSetAttachment(asi->coder,asi)<0);
   fail = fail || (hapiCoderInitialise(asi->coder,asi->chan)<0);

   /* Want a source that can be used to play back audio */
   fail = fail || ((asi->play=hapiCreateSourceObject(HAPI_SRC_haudio))==NULL);
   fail = fail || (hapiSourceSetAttachment(asi->play,asi)<0);

   fail = fail || IS_HAPI_NAN(asi->sampPeriod=hapiSourceSamplePeriod(asi->src,NULL));
   fail = fail || (hapiSourceSamplePeriod(asi->play,&asi->sampPeriod)<0);

   fail = fail || (hapiSourceInitialise(asi->play)<0);

   fail = fail ||
      IS_HAPI_NAN(asi->obsAdv=hapiCoderObservationWindowAdvance(asi->coder));
   fail = fail ||
      ((asi->parmKind=hapiCoderObservationParameterKind(asi->coder))==NULL);

   if (fail && asi->coder!=NULL)
      hapiDeleteCoderObject(asi->coder),asi->coder=NULL;
   if (fail && asi->src!=NULL)
      hapiDeleteSourceObject(asi->src),asi->src=NULL;
   if (fail && asi->play!=NULL)
      hapiDeleteSourceObject(asi->play),asi->play=NULL;

   return(fail?-1:0);
}
```

Since the application allocates several objects the deletion function needs to delete all of them (and tries to delete all of them even if an error occurs).

```
int exSourceCoderObjectDel(ApplSrcInfo *asi)
{
   int fail=0;
   fail = (hapiDeleteCoderObject(asi->coder)<0) || fail;
   fail = (hapiDeleteSourceObject(asi->src)<0) || fail;
   fail = (hapiDeleteSourceObject(asi->play)<0) || fail;
   asi->coder=NULL;
   asi->src=NULL;
   asi->play=NULL;
   return(fail?-1:0);
}
```

11.7 Utterance processing

Each utterance is treated separately by the coder. Prior to processing each utterance the coder needs to be prepared by calling `hapiCoderPrepare` before the source is started. The observations in the utterance can then be read one at a time using **hapiCoderReadObs** until end of utterance is indicated by **hapiCoderRemaining** returning `HAPI_REM_done`.

```
if (hapiCoderPrepare(asi->coder)<0) return;
asi->obs=hapiCoderMakeObservation(asi->coder);
printf("Press return to start utterance\n");
gets(buf); nFr=0;
if (hapiSourceStart(asi->src)<0) return;
while(hapiCoderRemaining(asi->coder)!=HAPI_REM_done) {
   if (hapiCoderReadObs(asi->coder,NULL)==HAPI_REM_error) break;
   nFr++;
}
if (hapiCoderComplete(asi->coder)<0) return;
printf("\n\n Processed %d frames",nFr);
```

It is also possible for the application to stop the source directly. For example the following line could be added into the loop reading observations in the code fragment above. Assuming the function `KeyPressed()` indicates when a key has been pressed by the user this will stop audio input when a key is pressed. Note that processing of remaining frames may be slightly delayed due to the latency in the recognition process. This method of stopping is therefore not equivalent to just aborting the loop immediately although this would happen if an error code was returned.

```
if (KeyPressed() && hapiSourceStop(asi->src)<0) break;
if (!IS_HAPI_NAN(vol=hapiSourceCurVol(asi->src)) & vol>0.0)
   printf("Volume == %.1f dB\n",vol),fflush(stdout);
```

This also shows how the `hapiSourceCurVol` function can be used to provide the user with a VU meter type indication of the current input level. This is especially useful for indicating when the speech detector has detected silence at the end of the utterance and has automatically stopped audio input. The function returns a number less than zero if no estimate of the volume is currently available.

One other function that should be mentioned is `hapiSourceAvailable`. This function indicates whether more utterances are available from the source. It only really applies to input from a script file listing audio files with an utterance in each. Other source types can supply data indefinitely.

```
i=hapiSourceAvailable(asi->src);
if (i==HAPI_REM_done || i==HAPI_REM_error) return;
```

11.8 Access

Although the primary purpose of the source and coder objects is to provide observations direct to the recogniser they do also allow the application some direct access to the raw waveform data itself.

The format of an observation is hidden from the application although it can pass individual observations between parts of HAPI for rescoring and reprocessing. However waveform access is provided (with all samples stored as 16 bit linear samples) and the application can process the waveform as it wants. The most common use of waveform access is to play back portions of the data. This waveform access can also be used to provide the application with access to pre-recorded prompts. By storing these on disk and reading them in using a "dummy" coder the waveform can be made accessible for playback.

Two functions provide access to the complete utterance once the final frame has been read in. The first `hapiCoderAccessObs` provides access to the individual observations for rescoring whilst the other `hapiCoderAccessWave` provides access to the entire waveform (assuming the source is producing waveform rather than partially coded data).

For example

```
asi->obs=hapiCoderMakeObservation(asi->coder);
for (i=1;i<=nFr;i++) {
   if (hapiCoderAccessObs(asi->coder,i,asi->obs)<=0) break;
   /* Do something with observation */
.
.
}
```

will loop through all the observations read in the example above.

Probably the most common use for waveform access is audio playback and this can be accomplished in the following way.

```
n=hapiCoderAccessWave(asi->coder,1,-1,NULL);
if (n>0) {
   short16 *wave;
   printf("  Playing waveform (%d samples)\n",n);
   wave=malloc(sizeof(short16)*n);
   hapiCoderAccessWave(asi->coder,1,-1,wave);
   hapiSourcePlayWave(asi->play,n,wave);
}
else
   printf("  No access to waveform for playback\n");
```

The remaining two access functions exist to save the utterance to disk, either in the original format captured by the source object (typically waveform) or as parameterised observations.

So either

```
if (hapiCoderSaveObs(asi->coder,fn)<0)
   printf("  Failed to save observations\n");
```

or

```
if (hapiCoderSaveSource(asi->coder,fn)<0)
   printf("  Failed to save source\n");
```

can be used to save the current utterance for later processing and further experimentation.

11.9 Application defined sources

HAPI incorporates an interface through which an application can provide HAPI with audio data, either as a waveform or as a sequence of parameterised feature vectors. The application can register any number of new source drivers with HAPI which will allocate each one a new source type and return the type number to the application to allow it to create new source objects interfaced to its own data capture routines.

The interface from HAPI to the data capture routines is via a number of function pointers which are passed in a structure that defines the new source (`hapiSourceDef`).

`gInfo` The initial value for a hook that is passed to each driver function. This hook can be used to provide access to data needed across all source objects using this driver.

`sampPeriod` The sample period for both source and playback data. If this is zero then HAPI assumes that the driver can operate at any requested sampling rate as defined in the configuration file. Otherwise it is taken as the actual sample period of the data sourced from and played by the driver.

`forceDetUse` The application can also control the way in which the new driver interacts with the automatic speech detector. If this value is zero the configuration will be used to determine when to use the speech detector. Otherwise the detector is either always enabled if the flag is greater than zero or disabled if the flag is less than zero.

`srcType` Defines the type of the data generated by the source and transfered to HAPI. This will be checked at registration to ensure that the HAPI interface can correctly process the data. If it cannot the registration function will return an error value.

`playType` Defines the type of data that can be played back to the user by the source. This value will also be checked at registration to ensure that the HAPI interface can generate data of the correct form. Note that there is no need for the `srcType` and `playType` to be the same although if both capture and playback are supported this would typically be the case.

`esRegister` Function called once the source has been registered with HAPI (but before the **hapiRegister** function returns). It gives the driver a chance to perform some preliminary generic initialisation. In common with all driver routines the first parameter is the `hapiSourceInfo` structure used to return status information and to pass the global driver hook to the driver functions.

`esShutdown` Function called when the HAPI library is shutting down giving the driver an opportunity to free resources it may have allocated during the **esRegister** call.

`esCreate` Function called when a new source object of the appropriate type is created. The return value of this function is passed to every driver function associated with this object so can be used to hold object dependent driver information.

`esDelete` Function called when the source object is deleted. Should free all resources allocated in the **esCreate** call.

`esInitialise` Function called when the source object is initialised. Gives the driver an opportunity to take note of the coder object that is connected to the source.

`esSrcAvailable` Function used to determine if the source can generate further utterances. Normally this will always return `HAPI_REM_more` but sources reading from a fixed script of filenames may wish to indicate when the script is exhausted and the application should terminate.

`esPlayWave` Play waveform data to the user.

`esPrepare` During preparation of the coder the source is given an opportunity to perform and time consuming resource allocation or initialisation tasks. For some drivers this call may do nothing and the **esStart** function performs all initialisation as well as starting the source collecting data. However other drivers may wish to allocate and initialise resources although this function should not block waiting for external events. Such blocking should occur in the **esStart** function.

`esComplete` Called once utterance processing has finished. This should free resources allocated by the **esInitialise** and **esStart** functions.

`esStart` Function called when the corresponding source is started with a call to **hapiSourceStart**. The driver should begin data capture and return once data is being made available (blocking until this point when necessary).

`esStop` When the application explicitly stops the source collecting data or the speech detector has found the end of speech and ended the utterance this function is called to stop the driver collecting data. Once this function has been called the driver must show end of utterance when appropriate and should no longer block waiting for data to arrive (as none should).

`esNumSamples` HAPI polls the driver to read data. Once the utterance has been exhausted this function should return a negative number indicating end of utterance otherwise it should return the number of bytes immediately available to HAPI without the driver blocking and waiting for more data. To ensure good real time performance

of the speech detector this function must return an accurate estimate. However, whilst the utterance is in progress the driver can always return 0 as the number of samples available. In this case HAPI will only request a frame of data when it needs to be processed. This will not effect the speed of recognition but will mean that all the HAPI buffering and speech endpointing functions occur at the speed of the slowest system component (often the recogniser).

`esGetSamples` Called to obtain data for HAPI. Normally this will only be data available immediately (as determined by the **esNumSamples** call) but when HAPI is waiting for data to arrive (and has processed all the previous data) this function is the point at which the process should block.

All of these functions are passed a `hapiSrcDriverStatus` structure for returning status codes, messages and modified global hook values (when necessary).

- `hapiStatusCode status;`

  Each function is expected to return a status value as part of the `hapiSourceInfo` structure. However the value is initialised to indicate the operation was successful (`HAPI_ERR_none`) so the functions only need to alter it when this was not the case.

- `char8 statusMessage[STR_LEN]`

  When the returned status code indicates an error this buffer can be used to return a textual description of the problem (via the **hapiCurrentStatus** call).

- `void *gInfo`

  Driver hook passed to every function and initialised to the value passed in the source definition. Changes made in any driver function will be propagated when further calls are made.

11.9.1 Troubleshooting

Many of the problems that are encountered will be due to the driver not operating as HAPI expects. Here are a few tips to help with some of the more common problems.

- Driver captures data 'slowly'.

  Ensure that the **esNumSamples** function is completing quickly and is returning an underestimate of the data that can be read instantly using the **esGetSamples** call. The **esNumSamples** function is called at least once for each processed frame and if sufficient data is available this call is followed by calls to **esGetSamples**. To ensure this sequence complete quickly, **esGetSamples** must itself quickly return a value that is an underestimate of the number of samples immediately readable by **esGetSamples**.

  To test that the function is working modify the function to return `HAPI_REM_more` until the source is exhausted and then return `HAPI_REM_done`. If this improves speed of operation then the original implementation of **esNumSamples** was flawed.

- Memory corruption.

  Remember that each driver can be associated with multiple objects and so needs to keep local and global data separate (and ensure that each item of data is treated in the correct way). Ensure that each `hapiSourceObject` and associated `hapiSrcDriverObjectInfo` is treated separately and that modifications to shared data (such as the `gInfo` pointer) are meant to effect all objects using the driver.

- Driver locks up.

  It is possible to end up with the driver recursively calling itself. In general it is best to avoid calls to HAPI from within the driver. When this is not possible those calls should be restricted to the initialisation and setup portion of the driver rather than being called from within the utterance processing calls. For example if the driver uses **hapiCoderRemaining** to enquire about coder status a infinite recursive loop could be set up. This means that care must be taken if the driver calls an event handler to allow the application to do background processing during recognition.

Chapter 12

hapiDictObject

As well as providing facilities direct to the application for looking up word pronunciations and textual symbols, the `hapiDictObject` is used during the construction of recognition networks and specifies the sequence of models used to recognise each word.

The remainder of this chapter starts with a description of certain types associated with the `hapiDictObject` and follows on with descriptions of its structural definition and configuration and code examples illustrating its use and functionality.

12.1 Types

HAPI defines three overloaded integer types which are used by the dictionary object. These (`hapiWordId`, `hapiPronId` and `hapiPhoneId`) are used as indexes to reference words, different pronunciations of each words and phones defining the acoustic model sequence representing a pronunciation of each word. Word and pronunciation ids are both 32-bit signed integers whilst the phone ids are 8-bit unsigned allowing a compact representation of pronunciations. (If the dictionary contains a phone set with more than 255 phones then HAPI is unable to access and alter the pronunciations - although the dictionary can still be used). Because phone ids are actually 8-bit unsigned integers a separate type (`hapiXPhoneId`) is used as the return type for functions which also need to return a status code. This is a 32-bit signed quantity which can represent both status codes and phone ids.

Within each dictionary, words are numbered from 1 to the number of words in that dictionary. This index (or `wid`) is used within HAPI to refer to the word and, despite being an integer, it is always marked with its own type (`hapiWordId`) in function definitions. The first word (for which the `wid` is `1`) is always the special `!NULL` word used within lattices to indicate a collating node not associated with any particular word.

As will be explained later (Section 12.5) a dictionary can be modified. Words and/or pronunciations can be either added or deleted. The dictionary object is made *modifiable* prior to any edit operations and is then *finalised* after them. When a dictionary becomes modifiable the `wids` of current words do not change and new words are allocated new `wids`. However this means that not all `wids` are necessarily valid and so when the dictionary is finalised all the valid words are resorted and reallocated new `wids`. If the application caches the mapping between word names

and word ids it must flush those caches after dictionary modifications are complete. Also if initialised objects (such as lattices or networks) are using the dictionary any modifications to the dictionary will not necessarily be propagated to the objects. For example, changes to the pronunciation of a word will be completely ignored until a network is rebuilt. Similarly, although a word can be deleted even if it is actually being used by a lattice, no error is generated until the lattice is processed further.

Pronunciations are numbered from `1` and maintain this even during modifications. This means that as pronunciations are deleted their numbering can change. For example suppose a word had six pronunciations numbered from `1` to `6` and the application wished to delete the first, third and fifth. It should do this by deleting the fifth then the third and finally the first. Otherwise it would actually need to delete them as `1,2,3` because of the numbering changing as earlier pronunciations are deleted.

Phones are again indexed from `1` with a maximum of `255` phones in each dictionary. Each pronunciation is stored as a zero terminated array of `hapiPhoneIds`.

12.2 Definition

Dictionaries are stored in HTK format files off-line. These are simple text files with each line containing a pronunciation entry of the form

```
WORD [OUTSYM]  PRONPROB   a b c d
```

This indicates that `WORD` has a pronunciation `a b c d` and should produce the string `OUTSYM` when recognised. The `[OUTSYM]` is optional and when absent indicates that the word name should be produced by default. The `PRONPROB` is an optional pronunciation probability for alternative pronunciations.

In addition to these standard HTK entries a HAPI dictionary also includes a special word (`!!PHONE_SET` which defines the set of phones used in the dictionary. Normally this can (and will) be constructed automatically by scanning the dictionary and enumerating all the phone names that appear. However in small dictionaries not every phone in the phone set will necessarily be represented in the dictionary. In order for HAPI to be able to add pronunciations using rare phones it must know of their existence. Consequently dictionaries in HAPI will normally begin with a special word defining the entire phone set. This dummy word has a pronunciation consisting of every phone in the phone set and is deleted once the dictionary has been loaded.

For example a spelling recogniser could have a phone representing each letter plus a silence model and the dictionary may begin

```
!!PHONE_SET a b c d e f g h i j k l m n o p q r s t u v w x y z sil
A a
B b
C c
.
.
```

Since the dictionary consists of a single file only a single configuration parameter is required to define the location of the dictionary.

Module	Name	Default	Description
HAPI	`DICTFILE`	`"dict"`	Name of dictionary file.

This DICTFILE configuration variable would normally be set to the same value as the command line VocabFile option for the HTK decoder HVITE i.e.

```
HVite ... $DICTFILE $HMMLIST ....
```

12.3 Usage

This section uses example code to illustrate how to use the hapiDictObject within an application program. In common with the example code for the HMMSet the fragment below defines a simple structure will be used to hold information related to the dictionary object. Initially this will just contain initialisation information (which for the dictionary is just the file name from which it should be loaded) but it will be used in other code fragments to store and access both the dictionary reference and also other dictionary related information.

```
/* --------------------  Dict example code --------------------- */

typedef struct appldictinfo {
   hapiDictObject dict;   /* Reference to HAPI dictionary */
   char8 *file;           /* Input dictionary file name */
   int32 nPhones;         /* Number of phones in dictionary */
   int32 nWords;          /* Number of words in dictionary */
   hapiDictObject local;  /* Another HAPI dictionary */
   hapiPhoneId phid;      /* A phone id */
   hapiWordId wid;        /* A word id */
   hapiPronId pid;        /* A pronunciation number */
}
ApplDictInfo;
```

Allocating and initialising the dictionary is very easy since it is loaded from a single file that can either be set via the configuration parameter, or by overriding this before initialising the object. The initialisation code below checks the value of adi->file before either overriding the configuration filename or querying it.

```
int exDictObjectInit(ApplDictInfo *adi)
{
   hapiDictObject dict;
   char8 *str;
   int32 fail=0;

   /* Create dictionary and attach application info to hapiObject */
   fail = fail || ((adi->dict=dict=hapiCreateDictObject())==NULL);
   fail = fail || (hapiDictSetAttachment(dict,adi)<0);

   if (adi->file!=NULL)
      /* Need to set dictionary file name */
      fail = fail || (hapiDictOverrideFileName(adi->dict,adi->file)==NULL);
   else {
      /* Need to query dictionary file name */
      fail = fail || ((str=hapiDictOverrideFileName(dict,NULL))==NULL);
      if (!fail) adi->file=StrAllocCopy(str);
   }
```

```
   fail = fail || (hapiDictInitialise(dict)<0);

   adi->nPhones=hapiDictAllPhoneNames(dict,NULL);
   adi->nWords=hapiDictAllWordNames(dict,NULL);

   if (fail && adi->dict!=NULL) {
      hapiDeleteDictObject(adi->dict);
      adi->dict=NULL;
   }

   return(fail?-1:0);
}
```

Note how the function tests the return code of most functions and deletes the dictionary object if an error occurs.

Once the application has finished using the dictionary (both directly and via results, networks and lattices based on the dictionary) it should delete the object.

```
int exDictObjectDel(ApplDictInfo *adi)
{
   int32 fail=0;
   fail = fail || (hapiDeleteDictObject(adi->dict)<0);
   adi->dict=NULL;
   return(fail?-1:0);
}
```

There is also a function to save a dictionary to a file that could be used to update the dictionary on disk to reflect changes made during the execution of the application.

```
   if (hapiDictSave(adi->dict,fn)<0)
      printf(" Failed to save dictionary\n");
   else
      printf(" Dictionary Saved in File %s\n",fn);
```

12.4 Access Functions

HAPI provides a number of functions which allow read access to various parameters in the dictionary object. These access functions can be grouped into three broad categories.

- Functions to access the phone table to determine the mapping between phone names and ids.
- Functions to find what words are present in the dictionary and map between their names and ids.
- Functions to access pronunciations of words and their associated output symbols.

There are three functions concerned with access to the phone table. The first `hapiDictAllPhoneNames` can be used to query the number of distinct phones in the dictionary (and optionally to obtain an array containing their names indexed by their ids), the second `hapiDictPhoneName` provides a mapping from the phone id to a

textual name and the final one `hapiDictPhoneId` provides the reverse mapping from a textual name to the phone id.

The following code fragment, which prints the phone table for the dictionary, uses all three functions.

```
char8 buf[STR_LEN];

if ((adi->nPhones=hapiDictAllPhoneNames(adi->dict,NULL))<=0)
   printf(" Dictionary invalid\n\n");
else
   printf(" Dictionary has %d phones:\n",adi->nPhones);
for (i=1;i<=adi->nPhones;i++) {
   if (hapiDictPhoneName(adi->dict,i,buf)>=0) {
      printf(" %2d=%-3s",hapiDictPhoneId(adi->dict,buf),buf);
      if ((i%10)==0) printf("\n");
   }
}
printf("\n");
```

The first call finds the number of phones in the phone table (but does not obtain the array because the second argument is `NULL`). Then for each valid phone id, the loop finds the name of the phone and prints that together with its index (verified by actually looking it up).

It is also possible to obtain this mapping between phone ids and names in a single call in the form of an array indexed by the phone id. This method of access is illustrated by the following code fragment.

```
char8 **ptrs;

if ((adi->nPhones=hapiDictAllPhoneNames(adi->dict,NULL))<=0)
   printf(" Dictionary invalid\n\n");
else {
   printf(" Dictionary has %d phones:\n",adi->nPhones);
   ptrs=malloc(sizeof(char*)*(adi->nPhones+1));
   adi->nPhones=hapiDictAllPhoneNames(adi->dict,ptrs);
   for (i=1;i<=adi->nPhones;i++) {
      if (ptrs[i]!=NULL) {
         printf(" %2d=%-3s",hapiDictPhoneId(adi->dict,ptrs[i]),ptrs[i]);
         if ((i%10)==0) printf("\n");
      }
   }
}
free(ptrs);
printf("\n");
```

Note that the entries in this table will remain allocated until the dictionary is deleted but will only remain valid until the dictionary is next finalised (because words may change their ids at this time).

This method of access is quite common in HAPI with an initial call to a particular function being used to determine the size of an array to hold results from the next call to the function. Also note that because the phones are indexed from 1 the initial element of the array is not used. Consequently it would be perfectly valid to allocate `ptrs` as

```
ptrs=malloc(sizeof(char*)*adi->nPhones); ptrs--;
```

Three similar functions are used to access the word table mapping word names to ids. For example the following code can be used to print all the words in the dictionary.

```
char8 buf[STR_LEN];

if ((adi->nWords=hapiDictAllWordNames(adi->dict,NULL))<=0)
   printf(" Dictionary invalid\n\n");
else
   printf(" Dictionary has %d words:\n",adi->nWords);
for (i=1;i<=adi->nWords;i++) {
   if (hapiDictWordName(adi->dict,i,buf)<=0)
      printf("   [DEL]       ");
   else printf(" %3d=%-10s",hapiDictWordId(adi->dict,buf),buf);
   if ((i%5)==0) printf("\n");
}
printf("\n");
```

Note that the `hapiDictAllWordNames` function can be used to create a table for the mapping from word ids to textual names. Again the entries in this table will remain valid until the dictionary is next finalised or deleted.

```
char8 **ptrs;

if ((adi->nWords=hapiDictAllWordNames(adi->dict,NULL))<=0)
   printf(" Dictionary invalid\n\n");
else {
   printf(" Dictionary has %d words:\n",adi->nWords);
   ptrs=malloc(sizeof(char*)*(adi->nWords+1));
   hapiDictAllWordNames(adi->dict,ptrs);
   for (i=1;i<=adi->nWords;i++) {
      if (ptrs[i]==NULL) printf("   [DEL]       ");
      else printf(" %3d=%-10s",hapiDictWordId(adi->dict,ptrs[i]),ptrs[i]);
      if ((i%5)==0) printf("\n");
   }
   free(ptrs);
   printf("\n");
}
```

The final set of access functions are for accessing output symbols and pronunciations of words.

```
hapiPhoneId *pids;

n=hapiDictWordAllProns(adi->dict,adi->wid,NULL);
if (hapiDictWordName(adi->dict,adi->wid,buf)<0 || n<=0)
   printf(" Word %d is not valid\n",adi->wid);
else {
   printf(" Word %d (%s) :\n",adi->wid,buf);
   for (i=1;i<=n;i++) {
      if ((q=hapiDictWordPron(adi->dict,adi->wid,i,NULL))<=0 ||
          (pids=malloc(sizeof(hapiPhoneId)*q))==NULL ||
           hapiDictWordPron(adi->dict,adi->wid,i,pids)!=q)
```

```
            continue;
         ptr=hapiDictPronOutSym(adi->dict,adi->wid,i,NULL);
         if (ptr==NULL) printf(" %2d  NULL - ",i);
         else printf(" %2d  [%s] -",i,ptr);
         for (j=0;j<q;j++) {
            if (hapiDictPhoneName(adi->dict,pids[j],buf)<0)
               printf(" %d==???",pids[j]);
            else
               printf(" %d==%s",pids[j],buf);
         }
         printf("\n");
         free(pids);
      }
   }
```

Since there is a pre-defined maximum number of phones in any pronunciation it is possible to avoid the need to allocate the array to hold each pronunciation individually and instead use a pre-allocated array of the maximum size. In the example above the code calling `hapiDictWordPron` and allocating `pids` can be replaced with

```
hapiPhoneId pids[HAPI_MAX_PRON_LEN];
if ((q=hapiDictWordPron(adi->dict,adi->wid,i,pids))<=0) continue;
```

Note that the function `hapiDictPronOutSym` can be used to alter the output symbol associated with a particular word or pronunciation without needing to make the dictionary modifiable. Changes made in this way will be propagated immediately to all networks and lattices using the dictionary.

```
oldSym=hapiDictPronOutSym(adi->dict,adi->wid,i,newSym);
```

In a similar manner to the `hapiDictAllPhones` and `hapiDictAllWordNames` functions, `hapiDictWordAllProns` can be used to query the number of pronunciations of a chosen word (and optionally to load an array with the textual representation of these pronunciations).

```
hapiPhoneId *pptrs;

n=hapiDictWordAllProns(adi->dict,adi->wid,NULL);
if (hapiDictWordName(adi->dict,adi->wid,buf)<0 || n<=0)
   printf(" Word %d is not valid\n",adi->wid);
else {
   pptrs=malloc(sizeof(hapiPhoneId*)*n); pptrs--;
   hapiDictWordAllProns(adi->dict,adi->wid,pptrs);
   printf(" Word %d (%s) :\n",adi->wid,buf);
   for (i=1;i<=n;i++) {
      ptr=hapiDictPronOutSym(adi->dict,adi->wid,i,NULL);
      if (ptr==NULL) printf("  NULL - ");
      else printf("  [%s] -",ptr);
      for (j=0;pptrs[i][j]!=0;j++) {
         if (hapiDictPhoneName(adi->dict,pids[j],buf)<0)
            printf(" %d==???",pids[j]);
         else
            printf(" %d==%s",pids[j],buf);
      }
```

```
      printf("\n");
   }
   free(pptrs);
}
```

12.5 Modification Functions

Facilities also exist to modify existing dictionaries, by adding and/or deleting new words and/or pronunciations. Such edit operations must be wrapped with calls to make the dictionary modifiable and then finalise the changes prior to using the dictionary again e.g. to generate a recognition network.

```
   fail = fail || (hapiDictModify(adi->dict)<0);
.
.
   /* Make changes */
.
.
   fail = fail || (hapiDictFinalise(adi->dict)<0);
```

Note that it is not feasible to simply create a new dictionary from scratch within the application. There must be some way to define the phone set that it will use. This can either be read in from a file or from an empty dictionary sharing a phone set with an existing dictionary.

```
   adi->local=hapiEmptyDictObject(adi->dict);
.
.
   /* Add words */
.
.
   fail = fail || (hapiDictFinalise(adi->local)<0);
```

When a dictionary is created in this manner it is returned in a modifiable state ready for pronunciations to be copied directly from the template dictionary. Hence the call to `hapiDictFinalise` without a corresponding call to `hapiDictModify`. The following code fragment shows how words are copied from the template dictionary into the newly created dictionary.

```
hapiPhoneId pron[HAPI_MAX_PRON_LEN];
char8 buf[STR_LEN],*out;
int32 p;

p=hapiDictWordPron(adi->dict,adi->wid,adi->pid,pron))<0);
if (hapiDictWordName(adi->dict,adi->wid,buf)<0) p=0;
out=hapiDictPronOutSym(adi->dict,adi->wid,adi->pid,NULL);
if (out==NULL) out=buf+strlen(buf); /* Empty string */
if (p>0) hapiDictAddItem(adi->local,buf,pron,out)<0);
```

Deleting items is equally simple

```
if (adi->pid<=0) /* Delete word and all prons */
   hapiDictDeleteItem(adi->local,adi->wid,0);
else /* Delete a single pron */
   hapiDictDeleteItem(adi->local,adi->wid,adi->pid);
```

However, as explained previously, deleting pronunciations results in the remaining ones being renumbered and so when multiple pronunciations are to be deleted they should be done in decreasing number order. For example.

```
int del[]={ 5, 3, 1, 0},*i; /* In correct order */

for (i=del;*i!=0;i++)
   hapiDictDeleteItem(adi->local,adi->wid,*i);
```

or

```
int del[]={ 5, 3, 1, 0},*i;

for (i=del;*i!=0;i++);  /* Sort into correct order */
qsort(del,i-del,sizeof(int),int_cmp);
for (i=del;*i!=0;i++)
   hapiDictDeleteItem(adi->local,adi->wid,*i);
```

Chapter 13

hapiLatObject

A `hapiLatObject` represents a word level lattice. These lattices can serve two purposes. Firstly they are used to specify the recognition grammar of the task at hand and secondly they can be used to store the results of the recognition process. The lattice containing the recognition results is, in effect, a subset of the grammar definition lattice containing only the most likely portion of the original lattice with respect to the utterance recognised. Results lattices can be used to store and provide the NBest hypotheses i.e. the most likely answer to the Nth most likely answer. Figure 13.1 shows an example of a word level lattice for use in a simple phone dialer task.

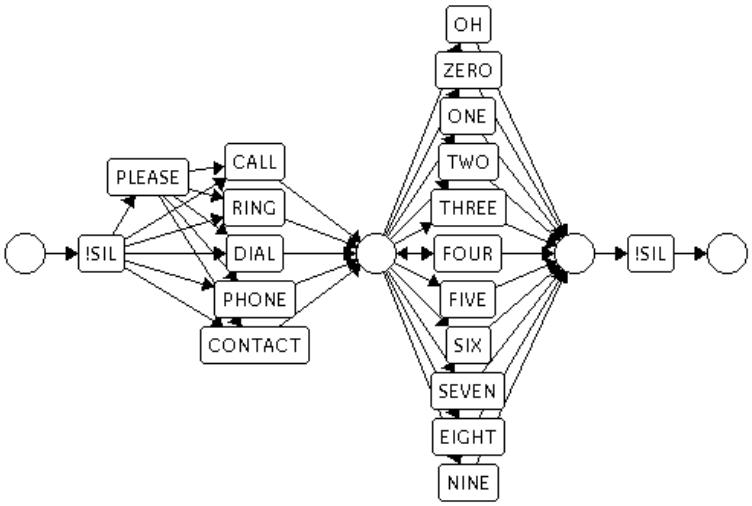

Fig. 13.1 A word level lattice

The round circles represent `!NULL` words which are effectively just collating points used to reduce the number of links required. It should also be noted that the first and last words in every path through the lattice are `!SIL`. This is a word representing the silence that will normally occur at the beginning and end of each utterance.

13.1 Definition

Lattices are used for specifying finite state syntaxes and storing results. Both types of lattice can be stored off-line in a file on disk. The only difference between the two forms of lattice is that results lattices include likelihoods in the definition. Internally there is a more compact format for storing syntax networks which does not include likelihood information. This is used when networks are loaded directly from disks and can be used with lattices loaded from disk when the configuration indicates this is required. This format is never used with results derived lattices.

The format for lattices provides for the inclusion of sub-lattices within the main lattice. Sub-lattices have essentially the same form as lattices and are used as a convenient method of encapsulating portions of a lattice which are repeated many times throughout the main lattice. An example of a sub-lattice might be a loop of digit models.

Some of these lattice attributes are configurable using the configuration variables described in the following table.

Module	Name	Default	Description
HAPI	`LATFILE`	`"lat"`	Default name of lattice file to load.
	`TIMEWIND`	`10.0`	Time window around requested start and end times for determining whether nodes are potential start and end of alternative paths specified for **hapiLatTimeAlternatives**.
	`LATWITHLIKES`	`TRUE`	Include likelihoods in lattices read from disk. This allows the scores to be accessed and used for determining likelihoods of paths (thus allowing the generation of alternatives) but uses more memory. If lattices read from disk are just used as word syntaxes this can be set to `FALSE` (results lattices are always generated with likelihoods).
	`SUBOBJECTS`	`TRUE`	A `hapiLatObject` is created for each sub lattice read in as part of multi level HTK lattice. This allows an application to freely manipulate any lattice contained within a multi level lattice file but again requires more memory. If the application does not perform any lattice manipulation this can be safely set to `FALSE`.

13.2 Usage

Lattices can be created in three ways.

They can be loaded from disk, created and built by the application or generated from a results object. This final method is described in the **hapiResObject** section.

```
/* --------------------  Lattice example code --------------------- */
```

```
typedef struct applatinfo {
   hapiLatObject lat;     /* Reference to lattice object */
   hapiDictObject dict;   /* Reference to dictionary used by lattice */
   char8 *file;           /* Input lattice file name */
   char8 name[STR_LEN];   /* Assigned lattice ID name */
   int32 nNodes;          /* Number of nodes in lattice */
   int32 nArcs;           /* Number of arcs in lattice */
   int32 n;               /* Number of arcs in best path (0 if not def) */
   hapiLatNodeId nid;     /* A node id */
   hapiLatArcId lid;      /* An arc id */
   hapiLatArcId *route;   /* A route through the lattice */
   hapiLatObject sub;     /* Reference to a sub lattice object */
}
ApplLatInfo;
```

The lattice object can either be created and loaded in a single function call (illustrated below) or can be created, have configuration parameters overridden and then loaded from disk as part of the initialisation call.

```
int32 exLatObjectInit(ApplLatInfo *ali,ApplDictInfo *adi)
{
   hapiLatObject lat;
   int32 fail=0;

   /* Create lattice and attach application info to hapiObject */
   if (ali->file!=NULL)
      fail = fail || ((lat=hapiLoadLatObject(adi->dict,ali->file,NULL))==NULL);
   else {
      fail = fail ||
 ((ali->file=StrAllocCopy(hapiOverrideConfStr("LATFILE",NULL)))==NULL);
      fail = fail || ((lat=hapiLoadLatObject(adi->dict,NULL,NULL))==NULL);
   }
   ali->lat=lat;

   fail = fail || (hapiLatSetAttachment(ali->lat,ali)<0);
   fail = fail || (hapiLatName(ali->lat,ali->name)<0);
   fail = fail || (hapiLatSize(ali->lat,&ali->nNodes,&ali->nArcs,&ali->n)<0);
   fail = fail || ((ali->dict=hapiLatDictObject(ali->lat))==NULL);

   if (fail && ali->lat!=NULL)
      hapiDeleteLatObject(ali->lat),ali->lat=NULL;

   return(fail?-1:0);
}
```

The numbers obtained from **hapiLatSize** can be used to determine the range of node and arc ids present in each lattice and also to decide if the lattice is derived from results or is just a syntax network. The final number that can be obtained from this function (`ali->n` above) indicates the number of arcs in the most likely path through the lattice. For syntaxes or other lattices without the required likelihood information the number returned will be zero.

When no longer needed the lattice object should be deleted.

```
int32 exLatObjectDel(ApplLatInfo *ali)
{
   int32 fail=0;
   fail = (hapiDeleteLatObject(ali->lat)<0) || fail;
   ali->lat=NULL;
   return(fail?-1:0);
}
```

HAPI also includes a function that writes out a lattice to a file on disk.

```
hapiLatSave(ali->lat,fn,"tvalIB");
```

The second argument to this function is a string specifying the output format, in particular the level information included in the output lattice.

Option	Description
`t`	Include node times (field `t`) in lattice
`v`	Pronunciation numbers
`a`	Acoustic likelihoods
`l`	Language model likelihoods
`r`	Pronunciation likelihoods
`d`	Phone align sequence
`n`	Phone durations
`m`	Phone likelihoods
`S`	Sort nodes and arcs before output
`A`	Write word labels on arcs rather than nodes
`B`	Write numeric fields in binary
`I`	Write single level lattice

The default format (used if the function parameter is `NULL`) is `"tvaldnm"`.

13.3 Access

The majority of functions are used to access the information in the lattice. These can be broadly split into two categories. Functions that return information about nodes and functions that return information about arcs. However probably the most important functions are those that specify which arcs are connected to which nodes.

There are functions that return the id of the node at the start and at the end of a particular arc.

```
hapiLatNodeId st,en;

st=hapiLatArcStart(ali->lat,ali->lid);
en=hapiLatArcEnd(ali->lat,ali->lid);
printf(" Arc %d from node %d to node %d\n",ali->lid,st,en);
```

There are functions that return the number of arcs (and optionally their ids) either following or preceding a particular node. When this number is zero the node is either the last or first node in the lattice.

Finding the follower arcs.

```
fail = fail || ((n=hapiLatNodeFollArc(ali->lat,ali->nid,links))<0);
fail = fail || ((links=malloc(sizeof(hapiLatArcId)*n))==NULL);
fail = fail || (hapiLatNodeFollArc(ali->lat,ali->nid,links)!=n);
if (n==0)
   printf("  Node %d - end node",ali->nid);
else
   printf("  Node %d - followed by",ali->nid);
for (j=0;j<n;j++)
   printf(" %d",links[j]);
printf("\n"); free(links);
```

Equivalent function for finding the preceeding arcs.

```
fail = fail || ((n=hapiLatNodePredArc(ali->lat,ali->nid,links))<0);
```

13.3.1 Nodes

Although HAPI treats lattices as though the arcs represent the words (with the nodes representing points at which transitions between words occur) the actual word labels are held on the nodes. This saves memory and is possible because of the way in which lattices are structured with all arcs ending at a particular node representing only one word. Consequently functions are provided not just to access the time of each node but also to query the identity of the word label associated with the node.

```
hapiWordId wid;
hapiPronId pnum;
hapiTime time;
char8 *ptr,buf[STR_LEN];

wid=hapiLatNodeWordId(ali->lat,ali->nid,NULL);
pnum=hapiLatNodePronId(ali->lat,ali->nid,NULL);
time=hapiLatNodeTime(ali->lat,ali->nid);
if (wid>0) {
   ptr=(hapiDictWordName(dict,wid,buf)<0)?"<Unknown>":buf;
   printf(" Node %d @%.2f sec represents WORD - %s %d/%d\n",
          ali->nid,time/1000.0,ptr,wid,pnum);
}
```

Note that the functions that access the word and pronunciation numbers can also be used to set them. Since this operation does not change the structure of the lattice it can be done at any time without the need to make the lattice modifiable first (See Section 13.6). To make use of this ability all that is necessary is to pass a pointer to the new value as the third function argument.

For example the following

```
hapiWordId wid,newid=1;

wid=hapiLatNodeWordId(ali->lat,ali->nid,&newid);
```

changes node `nid` to represent the `!NULL` word.

In common with the HAPI objects themselves each lattice node supplies the application with an attachment that it may use for any purpose.

```
void *ptr,newptr=NULL;

ptr=hapiLatNodeGetAttachment(ali->lat,ali->nid);
printf(" Node @ is attached to 0x%08x\n",ali->nid,ptr);
hapiLatNodeSetAttachment(ali->lat,ali->nid,newptr);
```

The separate functions which read and set the node attachment pointers operate in the same way as the attachment functions for each of the main objects.

13.3.2 Arcs

As explained above HAPI views each lattice arc as representing a word. Consequently the majority of access functions obtain information about a particular arc.

```
float32 like,aclk,lmlk;
hapiWordId wid;
hapiPronId pnum;
hapiTime stTime,enTime;
char8 *ptr,*out,buf1[STR_LEN],buf2[STR_LEN];

ptr=(hapiLatArcWordName(ali->lat,ali->lid,buf1)>0)?buf1:"<Unknown>";
out=(hapiLatArcOutSym(ali->lat,ali->lid,buf2)>0)?buf2:" - ";
wid=hapiLatArcWordId(ali->lat,ali->lid,NULL);
pnum=hapiLatArcPronId(ali->lat,ali->lid,NULL);
if (hapiLatArcTimes(ali->lat,ali->lid,&stTime,&enTime)<0)
   stTime=enTime=-999.999;
like=hapiLatArcScore(ali->lat,ali->lid,HAPI_SCORE_total,NULL);
aclk=hapiLatArcScore(ali->lat,ali->lid,HAPI_SCORE_acoustic,NULL);
lmlk=hapiLatArcScore(ali->lat,ali->lid,HAPI_SCORE_lm,NULL);
printf("  Arc %d:  Word %s [%s] (%d/%d) %.2f->%.2fs\n",
       ali->lid,ptr,out,wid,pnum,stTime/1000.0,enTime/1000.0,
printf("   Overall likelihood %.2f [Ac %.2f, LM %.2f])\n",like,aclk,lmlk);
```

Note that again the functions that return word ids, pronunciation numbers and scores can be used to set the values as well as read them and since this does not alter the structure of the lattice it does not have to be made modifiable prior to these calls (See Section 13.6). This is particularly useful in the case of **hapiLatArcScore** which can be used to set the language model scores for individual arcs. By setting the scores to very negative values (for example `LZERO` or `-1E10`) parts of the lattice down to the level of individual words can be selectively disabled without needing to modify and finalise the lattice. At a later stage these parts can be enabled by setting the language model score to `0`. However it should be noted that this enabling/disabling will only work if the recogniser language model scaling factor is set to a positive value. If it is zero the language model scores in the network (and therefore the lattice) will be ignored.

13.3.3 Phone alignment

If the lattice is derived from results produced by a recogniser that stores phone level alignment then the following functions will provide the application with score and timing information at the phone level.

```
if ((n=hapiLatArcAllPhoneIds(ali->lat,ali->lid,pron))>0) {
   for (i=0;i<n;i++) {
      id=hapiLatArcPhoneId(ali->lat,ali->lid,i);
      ptr=(hapiDictPhoneName(ali->dict,id,buf)>0?buf:"<unk>");
      if (hapiLatArcPhoneTimes(ali->lat,ali->lid,i,&stTime,&enTime)<0)
         stTime=enTime=-999.999;
      like=hapiLatArcPhoneScore(ali->lat,ali->lid,i,HAPI_SCORE_total);
      printf("  Phone %d == %d/%d (%s) %.2f-%.2f - %.2f\n",
             i,pron[i],id,ptr,stTime/1000.0,enTime/1000.0,like);
   }
}
```

13.4 Routes

Routes through lattices are stored as zero terminated arrays of lattice arc ids. Although there is no requirement for the arcs to form an unbroken sequence through the lattice they will typically be used for this type of route.

Functions exist for accessing paths through lattices in a single call. These are especially useful when processing the results from functions which return alternative hypotheses as these are in the form of routes through the lattice.

```
n=hapiLatRouteWordIds(ali->lat,NULL,ali->route);

if ((wids=malloc(sizeof(hapiWordId)*n))==NULL ||
    (names=malloc(sizeof(char8*)*n))==NULL ||
    (syms=malloc(sizeof(char8*)*n))==NULL)
   return;
if (hapiLatRouteWordIds(ali->lat,wids,ali->route)!=n ||
    hapiLatRouteWordNames(ali->lat,names,ali->route)!=n ||
    hapiLatRouteOutSyms(ali->lat,syms,ali->route)!=n)
   return;
for (i=0;i<n;i++)
   printf("  %3d : Wid %3d - %s [%s]\n",i,wids[i],
   (names[i]==NULL?"Unknown":names[i]),
   (syms[i]==NULL?" - ":syms[i]));
free(syms);free(names);free(wids);
```

13.5 Sub lattices

Both HAPI and HTK support nesting of lattice definitions i.e. a node in a main lattice referring to a sub lattice rather than just a word.

Normally when a multi-level lattice is loaded into HAPI each lattice is associated with a separate lattice object. These are somewhat special in that they are effectively owned by the top level lattice and the memory they use will only be freed when the top level object is deleted. It is an error to delete the top level if some of its child lattices are being used elsewhere.

```
hapiWordId wid;
char8 *sub,*ptr,buf[STRLEN];

wid=hapiLatNodeWordId(ali->lat,ali->nid,NULL);
```

```
sub=hapiLatNodeSublatId(ali->lat,ali->nid);
if (sub!=NULL) {
   ali->sub=hapiLatNodeSublatObject(ali->lat,ali->nid,NULL);
   if (ali->sub==NULL || hapiLatSize(ali->sub,&nn,&na,NULL)<0)
   printf(" Node %d is !SUBLAT - %s %d/%d)\n",ali->nid,sub,nn,na);
}
else {
   ptr=(hapiDictWordName(dict,wid,buf)<0)?"<Unknown>":buf;
   printf(" Node %d is WORD - %s %d\n",ali->nid,ptr,wid);
}
```

There is also a function that adds a node representing a sub lattice which is described below.

13.6 Modification

In a similar way to the modification of the `hapiDictObject` any modifications to the structure of a lattice must also be surrounded by calls to make the lattice modifiable and then to finalise the changes made to the structure. A newly created lattice can be made modifiable without reading it from disk when the application wants to construct the lattice from scratch.

Once the lattice has been marked as modifiable it cannot be used as a template to create a network. Although lattices that use the current lattice as a sub-lattice can be used to create a network any changes to the current lattice will not be propagated into the network until the lattice is finalised.

```
   fail = fail || (hapiLatModify(ali->lat)<0);
.
.
   /* Make changes */
.
.
   fail = fail || (hapiLatFinalise(ali->lat,sn,en)<0);
```

The finalise function takes two optional arguments specifying the expected start and end nodes of the lattice. If either of these is zero the start or end nodes are found automatically.

Before the lattice is actually finalised a check is performed to ensure that the lattice has a single start and a single end node, that all arcs and nodes are accessible from both start and end and that, when specified, the start and end nodes match those expected.

If all these checks are okay the lattice is rebuilt in an unmodifiable state, the nodes and arcs are reordered to ensure their ids are contiguous and if this lattice is used as a sub-lattice in any other lattices any changes are propagated and visible from the parent.

Once the lattice is modifiable both nodes and arcs can be added and deleted. There are two functions to add nodes; one to add a node representing a word (or `!NULL`)

```
dict=hapiLatDictObject(ali->lat);
ali->nid=hapiLatAddNode(ali->lat,wid,0);
ptr = (hapiDictWordName(dict,wid,buf)<=0)?"<Unknown>":buf;
printf(" Added word %s node %d\n",ptr,ali->nid);
```

and one to add a node representing a sub lattice

```
ali->nid=hapiLatAddSublatNode(ali->lat,ali->sub);
if ((ptr=hapiLatNodeSublatId(ali->lat,ali->nid))==NULL) ptr="<Failed>";
printf(" Added sublat %s node %d\n",ptr,ali->nid);
```

Since nodes are labelled during creation only a single function is required to add an arc.

```
ali->lid=hapiLatAddArc(ali->lat,st,en);
printf(" Added arc %d from node %d to %d\n",ali->lid,st,en);
```

Deleting nodes and arcs is also simple but it should be noted that deleting a node also removes any arcs connected to that node. This prevents the creation of dangling arcs.

For example

```
links=malloc(sizeof(hapiLatArcId)*ali->nArcs);
printf("  Deleting node %d will also delete arcs ",ali->nid);

n=hapiLatNodePredArc(ali->lat,ali->nid,links);
for (j=0;j<n;j++)
   printf(" %d",links[j]);
n=hapiLatNodeFollArc(ali->lat,ali->nid,links);
for (j=0;j<n;j++)
   printf(" %d",links[j]);
printf("\n"); free(links);

if (hapiLatDeleteNode(ali->lat,ali->nid)>=0)
   printf(" Deleted node %d\n",ali->nid);
```

or

```
if (hapiLatDeleteArc(ali->lat,ali->lid)>=0)
   printf(" Delected arc %d\n",ali->lid);
```

13.7 NBest

One of the principle uses of lattices is to allow the generation of multiple hypotheses for a single recognised utterance. Knowing these alternatives can make correction of results much easier and faster and also allows further post-processing of results to correct mistakes without needing user intervention.

Several functions are provided which return alternatives. All of them return the alternatives sorted into most likely to least likely order as arrays of routes. They differ in how the portion of the lattice to be scanned for alternatives is specified.

Three of the functions take a route as an argument together with two integers that specify the portion of the route for which alternatives are required. The route may be `NULL` implying that alternatives to a portion of the most likely hypothesis are required. Also the integers may specify either an offset from the start of the route when they are positive or from the end of the route when negative. So, for example, calling the function with route equal to `NULL` and `st=0, en=-1` requests alternatives for the whole utterance.

The functions differ in the way in which they consider routes to differ; one considers routes which represent the same word sequence to be equivalent, one considers routes with the same output symbols equivalent and one requires identical pronunciation sequences to consider the routes equivalent.

The returned answers will represent different routes (the level at which they differ is function specific) through the selected portion of the lattice with the remainder of the utterance unchanged.

For example consider an utterance and lattice for which the most likely hypothesis is `THE BLACK CAT SAT ON THE BROWN MAT`. Calling **hapiLatWordAlternatives** passing a `NULL` pointer for the route (implying that alternatives to the most likely hypothesis are required) and `st=1, en=2` requests alternatives for sentences of the form `THE ??? SAT ON THE BROWN MAT`. The array of routes will just cover the `???` portion of the utterance and will start with `BLACK CAT` as this is the most likely hypothesis.

```
#define NALT 10
   hapiLatArcId *links,*alt[NALT];
   char8 buf[STR_LEN];
   int32 n;

   switch(type) {
    case 'o':
       n=hapiLatOutSymAlternatives(ali->act,st,en,ali->route,NALT,alt);
       break;
    case 'p':
       n=hapiLatPronAlternatives(ali->act,st,en,ali->route,NALT,alt);
       break;
    case w':
    default:
       n=hapiLatWordAlternatives(ali->act,st,en,ali->route,NALT,alt);
       break;
   }
   if (n>0) {
      for (i=0;i<n;i++) {
         for (links=alt[i];*links;links++) {
            if (hapiLatArcWordName(ali->act,*links,buf)>0)
                printf(" %s",buf);
         }
      printf("  [");
      for (links=alt[i];*links;links++) {
         if (hapiLatArcOutSym(ali->act,*links,buf)>0)
            printf(" %s",buf);
      }
      printf(" ]\n");
   }
```

The remaining functions are less complex and do not perform any path equivalence tests. Alternatives returned will represent different paths through the lattice but may not represent different word sequences or different output symbol sequences.

One of the functions returns paths spanning a particular (time) period of the utterance and one of them returns paths between two nodes. Extreme care should be taken when using these functions though because the way in which periods of the utterance are specified is not as flexible as the previous functions. The **hapiLatTimeAlternatives** function requires that the word boundary times match those specified by

the function parameters to within a particular accuracy (which is configurable using the `TIMEWIND` configuration variable). Alternatives which span the period requested but do not actually start and end within the window period are not considered or returned. The **hapiLatNodeAlternatives** function finds alternative routes between the two specified nodes.

As an example the `switch` statement above could be expanded to include the following cases.

```
case 'n':
   n=hapiLatNodeAlternatives(ali->act,st,en,NALT,alt);
   break;
case 't':
   n=hapiLatTimeAlternatives(ali->act,stTime,enTime,NALT,alt);
   break;
```

Chapter 14

hapiNetObject

The network object provides the recogniser with a syntax defining what can be said (or at least what can be recognised). Since different recognisers may operate in different ways (because they have been optimised for different tasks) the network object is potentially recogniser specific.

14.1 Definition

Network objects are created in a variety of ways. The standard decoder creates them from a dictionary plus a grammar specification held as a word lattice. However a dictation type decoder will create a network from a dictionary plus a statistical language model rather than a finite state grammar held as a lattice.

Module	Name	Default	Description
HAPI	NETFILE	"net"	Filename of network to be loaded.
HNET	ALLOWCXTEXP	T	Allow context expansion to get model names
	ALLOWXWRDEXP	F	Allow context expansion across words
	FACTORLM	F	Factor language model likelihoods throughout words rather than applying all at transition into word. Can increase accuracy when pruning is tight and language model likelihoods are relatively high.
	CFWORDBOUNDARY	T	For word internal triphone systems, treat context free phones within words as marking word boundaries.
	FORCECXTEXP	F	Force triphone context expansion to get model names (is overridden by ALLOWCXTEXP)
	FORCELEFTBI	F	Force left biphone context expansion to get model names (ie. don't try triphone names)
	FORCERIGHTBI	F	Force right biphone context expansion to get model names (ie. don't try triphone names)

The standard decoder uses the above configuration parameters to control the construction of the network.

The word-level lattices used by the standard decoder are the same as those used by HVITE with the exception that HAPI can directly use multi-level lattices (i.e. those containing sub-lattices). These are only usable with the HTK decoder HVITE once they have been expanded into single level lattices using the HTK network building tool HBUILD.

When only a single network is used to recognise a series of utterances the config variable specifying the network is used in the same was as the HVITE command line option.

```
HVite ... -w $NETFILE ...
```

14.2 Usage

This section uses example code to illustrate how to use the network object within an application program.

The example code fragment below defines a simple structure will be used to hold information related to the network. This structure will be used to pass information between functions in our example code.

```
/* --------------------  Net example code -------------------- */

typedef struct applnetinfo {
   hapiNetObject net;
   hapiDictObject dict;
   hapiHMMSetObject hmms;
   hapiRecType type;      /* Recogniser type */
   ApplLatInfo *ali;      /* Lattice to convert to network */
   char8 *file;            /* Net file name */
}
ApplNetInfo;
```

There are three ways of creating a network. The standard method, which works with any network type, is to create the object then override any configuration parameters and finally initialise the network loading any required files from disk. The other two methods, which work for the standard recogniser but not necessarily for any other type, are to explicitly create and load the network from a file in one call or to produce a network from a word level lattice. The following initialisation function illustrates all three (the choice of which is controlled by the parameters passed to the function).

```
int exNetObjectInit(ApplNetInfo *ani,ApplHMMInfo *ahi,ApplDictInfo *adi)
{
   hapiNetObject net;
   char8 *str;
   int fail=0;

   if (ani->ali!=NULL)
      fail = fail ||
         ((ani->net=hapiBuildNetObject(ani->type,ahi->hmms,
                                       adi->dict,ani->ali->lat))==NULL);
```

```
   else
      if (ani->file!=NULL)
         fail = fail ||
            ((ani->net=hapiLoadNetObject(ani->type,ahi->hmms,
                                         adi->dict,ani->file))==NULL);
      else {
         /* Create Net and attach application info to hapiObject */
         fail = fail || ((ani->net=hapiCreateNetObject(ani->type,ahi->hmms,
                                                       adi->dict))==NULL);

         /* Need to query Net list file */
         fail = fail || ((str=hapiNetOverrideFileName(ani->net,NULL))==NULL);
         if (str!=NULL) ani->file=StrAllocCopy(str);

         fail = fail || (hapiNetInitialise(ani->net)<0);
      }
   fail = fail || (hapiNetSetAttachment(ani->net,ani)<0);
   fail = fail || ((ani->hmms=hapiNetHMMSetObject(ani->net))==NULL);
   fail = fail || ((ani->dict=hapiNetDictObject(ani->net))==NULL);

   /* We have one function not used yet so we need to demonstrate it */
   fail = fail || (ani!=hapiNetGetAttachment(ani->net));

   if (fail && ani->net!=NULL) {
      hapiDeleteNetObject(ani->net);
      ani->net=NULL;
   }

   return(fail?-1:0);
}
```

In common with all objects there is a delete function that frees all the memory allocated to the object.

```
int exNetObjectDel(ApplNetInfo *ani)
{
   int fail=0;
   fail = (hapiDeleteNetObject(ani->net)<0) || fail;
   ani->net=NULL;
   return(fail?-1:0);
}
```

Chapter 15

hapiRecObject

The recogniser is the object that brings all of the other resources together to process an utterance and hypothesise some results.

It requires a source of observations (either direct from the coder or via the application) and a network to direct the recognition. The network is generated from a dictionary and an HMMSet which are needed during recognition to calculate likelihoods and provide results information.

15.1 Types of Recogniser

HAPI contains support for multiple underlying decoders accessed by creating recognisers of different types.

These may be variants of the baseline decoder which operate in the same general fashion or they may be special purpose decoders optimised for a particular kind of recognition task.

For example the large vocabulary dictation decoder requires a special network together with a statistical language model and an extensively modified HMMSet of a particular type. Although extra configuration variables are used to specify the extra files and decoder parameters the actual HAPI interface remains the same. The network is created using the standard **hapiCreateNetObject** and **hapiNetInitialise** calls and is passed to the recogniser which is created, initialised and used in the same way. And although the results may contain slightly different information they are accessed via the same object interface as those produced by the standard decoder. Consequently this chapter will concentrate on describing the interface to the standard HAPI decoder.

The recogniser object is configured using the configurations variables shown in the table below.

Module	Name	Default	Description
HAPI	NTOKS	1	Number of tokens propagated into each state. Storing more than one token in each state allows the generation of multiple alternatives.
	LMSCALE	1.0	Language model scale factor. Alters balance between acoustic likelihoods calculated by HMMs and the prior probabilities in the network. Optimal values improve recognition accuracy.
	WORDPEN	0.0	Word insertion penalty. Also alters balance of HMMs and network.
	GENBEAM	250.0	General beam width controlling which hypotheses are pursued. Larger values can prevent certain types of errors at the expense of greater computation.
	WORDBEAM	75.0	Word-end pruning beam width. Controls extension of hypotheses at the end of words. Again larger values can increase accuracy but will require more computation.
	TMBEAM	10.0	Another pruning parameter but only used with "tied-mixture" HMMs. Value has little effect on accuracy and default is normally okay.
	MAXBEAM	2500	Maximum number of active nodes in the recognition network. Controls worst case speed of operation with smaller values resulting in faster recognition (at the expense of more errors).
HRec	FORCEOUT	F	Normally when no token reaches the end of the network by the last frame of the utterance no results are generated (indicated by number of words equal to zero). If this parameter is set to T then results are based on the most likely token anywhere in the network.

A typical set of entries in the configuration file might be.

```
# Recogniser parameters
 HAPI:  NTOKS = 1
 HAPI:  NBEST = 1
 HAPI:  LMSCALE = 1.0
 HAPI:  WORDPEN = 0.0
 HAPI:  GENBEAM = 250.0
 HAPI:  WORDBEAM = 75.0
 HAPI:  TMBEAM = 10.0
 HAPI:  MAXBEAM = 2500
```

Most of the configuration parameters set scaling factors and beam widths that

have direct correspondence to the command line options for the HTK recogniser tool HVITE.

```
HVite ... -n $NTOKS $NBEST  -s $LMSCALE  -p $WORDPEN \
      -t $GENBEAM  -v $WORDBEAM  -c $TMBEAM  -u $MAXBEAM ...
```

However, the recogniser components (which for HVITE are passed as further command line arguments) are not specified by the configuration but are explicitly passed as arguments to the relevant functions.

15.2 Usage

This section uses example code to illustrate the way that the recogniser object is used within an application program.

Unlike most other objects the recogniser does not really possess any resources of its own. It uses the other HAPI objects to provide it with the facilities it requires. These are explicitly passed as function parameters rather than being specified in configuration files and being directly loaded from disk. Of course, the objects may themselves have been created in just this way.

The example code fragment below defines a simple structure which will be used to hold information related to the recogniser.

```
/* -------------------  Rec/Res example code ------------------- */

typedef struct applrecinfo {
   hapiRecType type;      /* Recogniser type */
   hapiRecMode mode;      /* Recogniser mode */
   hapiResInfo info;      /* Results info required */
   hapiRecObject rec;     /* Reference to recogniser object */
   hapiResObject inter;   /* Reference to results object */
   hapiResObject final;   /* Reference to final results object */
   hapiLatObject lat;     /* Reference to results lattice object */
   hapiHMMSetObject hmms; /* Reference to HMMSet object */
   hapiNetObject net;     /* Reference to network object */
   volatile int32 ready;  /* Set non-zero when recognition complete */
   int32 cbi;             /* Call back interval */
   hapiObservation obs;   /* Useful observation */
   /*                        Configuration parameters */
   int32 ntoks;           /* Number of tokens */
   int32 nbest;           /* Number of alternatives written to label file */
   float32 scale;         /* Language model scale factor */
   float32 pen;           /* Word insertion penalty */
   float32 genBeam;       /* General beam width */
   float32 wordBeam;      /* Word end beam width */
   float32 tmBeam;        /* Tied mixture beam width */
   int32 max;             /* Maximum number of active network nodes */
   int32 (*finalFunc)(hapiResObject res); /* Final results */
   int32 (*interFunc)(hapiResObject res); /* Intermediate results */
}
ApplRecInfo;
```

Note that the structure contains two results objects **inter** and **final**. The recogniser object can supply intermediate results part way through an utterance as well as

the final results at the end. Since an application may want to process both intermediate and final results (but in different ways) it makes sense to create two separate results objects; one to hold the intermediate results (inter) and the other to hold the final results (final). This makes it possible to request different information and in call back mode makes it easy for the application to determine whether the call back function is being called to process the final or the intermediate results.

```
int32 exRecObjectInit(ApplRecInfo *ari,ApplHMMInfo *ahi,ApplNetInfo *ani)
{
   int32 fail=0;

   /* Create recogniser and attach application info to hapiObject */
   fail = fail || ((ari->rec=hapiCreateRecObject(ari->type,ari->mode,
                                                 ahi->hmms,ani->net))==NULL);
   fail = fail || ((ari->inter=hapiCreateResObject(ari->rec,
                                                   ari->info))==NULL);
   fail = fail || ((ari->final=hapiCreateResObject(ari->rec,
                                                   ari->info))==NULL);
   fail = fail || (hapiRecSetAttachment(ari->rec,ari)<0);
   fail = fail || (hapiResSetAttachment(ari->inter,ari)<0);
   fail = fail || (hapiResSetAttachment(ari->final,ari)<0);

   ari->hmms=ahi->hmms;ari->net=ani->net;

   fail = fail || (hapiRecInitialise(ari->rec)<0);
   fail = fail || ((ari->obs=hapiRecMakeObservation(ari->rec))==NULL);

   /* We have one function not used yet so we need to demonstrate it */
   fail = fail || (ari!=hapiRecGetAttachment(ari->rec));

   if (fail && ari->rec!=NULL)
      hapiDeleteRecObject(ari->rec),ari->rec=NULL;
   if (fail && ari->inter!=NULL)
      hapiDeleteResObject(ari->inter),ari->inter=NULL;
   if (fail && ari->final!=NULL)
      hapiDeleteResObject(ari->final),ari->final=NULL;

   return(fail?-1:0);
}
```

Unlike the majority of configuration parameters which only need to be set once, the recogniser pruning and scaling factors can be changed between recognitions (and in the case of the pruning parameters during recognition as well). So the following code could be used just before initialisation of the recogniser object (as is typical with other objects) or before the recognition of each utterance or even during the recognition process itself.

```
   hapiRecObject rec=ari->rec;

   if (ari->ntoks>=0)
      fail = fail || (hapiRecOverrideNToks(rec,&ari->ntoks)<0);
   else
      fail = fail || ((ari->ntoks=hapiRecOverrideNToks(rec,NULL))<0);
   if (ari->nbest>=0)
```

```
      fail = fail || (hapiRecOverrideNBest(rec,&ari->nbest)<0);
   else
      fail = fail || ((ari->nbest=hapiRecOverrideNBest(rec,NULL))<0);

   if (ari->scale>=0)
      fail = fail || (hapiRecOverrideLMScale(rec,&ari->scale)<0);
   else
      fail = fail || IS_HAPI_NAN(ari->scale=hapiRecOverrideLMScale(rec,NULL));
   if (ari->pen>=-1E10)
      fail = fail || (hapiRecOverrideInsPenalty(rec,&ari->pen)<0);
   else
      fail = fail || IS_HAPI_NAN(ari->pen=hapiRecOverrideInsPenalty(rec,NULL));

   if (ari->genBeam>=0)
      fail = fail || (hapiRecOverrideGenBeam(rec,&ari->genBeam)<0);
   else
      fail = fail || IS_HAPI_NAN(ari->genBeam=hapiRecOverrideGenBeam(rec,NULL));
   if (ari->wordBeam>=0)
      fail = fail || (hapiRecOverrideWordBeam(rec,&ari->wordBeam)<0);
   else
      fail = fail || IS_HAPI_NAN(ari->wordBeam=hapiRecOverrideWordBeam(rec,NULL));
   if (ari->tmBeam>=0)
      fail = fail || (hapiRecOverrideTiedMixBeam(rec,&ari->tmBeam)<0);
   else
      fail = fail || IS_HAPI_NAN(ari->tmBeam=hapiRecOverrideTiedMixBeam(rec,NULL));
   if (ari->max>=0)
      fail = fail || (hapiRecOverrideMaxActive(rec,&ari->max)<0);
   else
      fail = fail || ((ari->max=hapiRecOverrideMaxActive(rec,NULL))<0);
```

As with the combined source/coder example the `AppRecInfo` structure contains references to several HAPI objects and these need to be freed in the correct order (and even if an error occurs this function tries to free all the objects).

```
int32 exRecObjectDel(ApplRecInfo *ari)
{
   int32 fail=0;
   fail = (hapiDeleteResObject(ari->inter)<0) || fail;
   fail = (hapiDeleteResObject(ari->final)<0) || fail;
   fail = (hapiDeleteRecObject(ari->rec)<0) || fail;
   ari->rec=NULL; ari->inter=NULL; ari->final=NULL;
   return(fail?-1:0);
}
```

15.3 Frame by frame operation

The recogniser can operate in frame by frame mode whereby a fixed number of frames of data are continually processed until the utterance is over. Frame by frame operation of the recogniser is split into three phases

- Preparation.
- Until recognition complete

  - Process fixed number of observations.

- Completion.

Preparation consists of a single function call initialising the recogniser for use with a particular source/coder and network. Of course it is also necessary to prepare the coder for the next utterance as well as start the source at the appropriate time but these tasks are required regardless of whether the recogniser is running in frame by frame mode or in call back mode (See Section 15.4).

The loop processes a fixed observations and then returns control to the application which can display status information and do any background tasks prior to the next recognition call.

Finally once all the observations have been processed the recognition is completed by generating the final results object and disposing of the memory used to process the utterance.

```
int32 exRecognise(ApplRecInfo *ari,ApplSrcInfo *asi)
{
   hapiTime obsAdv;
   int32 fail=0,nFr=0,n;

   fail = fail || (hapiCoderPrepare(asi->coder)<0);
   fail = fail ||
      IS_HAPI_NAN(obsAdv=hapiCoderObservationWindowAdvance(asi->coder));
   fail = fail || (hapiRecPrepare(ari->rec,asi->coder,NULL)<0);

   /* Could wait here before starting audio input */
   fail = fail || (hapiSourceStart(asi->src)<0);

   while((st=hapiRecRemaining(ari->rec))!=HAPI_REM_done) {
      if (fail = fail || (st==HAPI_REM_error)) break;
      fail = fail || ((n=hapiRecProcess(ari->rec,ari->cbi,ari->inter))<0);
      /* Process intermediate results - eg
         hapiResTraceInfo(ari->inter,HAPI_TF_frame+HAPI_TF_word+
                          HAPI_TF_average+HAPI_TF_active,81,buf);
         printf("%s\n",buf); fflush(stdout); */
      fail = fail || (hapiResClear(ari->inter)<0); nFr+=n;
   }
   fail = fail || (hapiRecComplete(ari->rec,ari->final)<0);
   /* Process final results ??? */
   fail = fail || (hapiResClear(ari->final)<0);

   fail = fail || (hapiCoderComplete(asi->coder)<0);

   return(fail?-1:ret);
}
```

Frame by frame operation is also used for rescoring observations. This can be applied to portions of an utterance or (as the example below shows) to the whole of the utterance. Of course if this code were appended to the above function the same results would be obtained both times. However a rescoring operation could make use of different acoustic models or even a different recognition network.

```
   fail = fail || ((obs=hapiRecMakeObservation(ari->rec))==NULL);
```

```
   fail = fail || (hapiRecPrepare(ari->rec,asi->coder,NULL)<0);
   for (n=1;n<=nFr;n++) {
      fail = fail || (hapiCoderAccessObs(asi->coder,n,asi->obs)<0);
      if (fail) break;
      fail = fail || (hapiRecProcessObs(ari->rec,obs,obsAdv,NULL))<0);
   }
   fail = fail || (hapiRecComplete(ari->rec,ari->final)<0);
   /* Process final results ??? */
   fail = fail || (hapiResClear(ari->final)<0);
```

15.4 Callback operation

Callback operation encapsulates the three stages of frame by frame recognition into a single function call. Results processing is accomplished in an application function (known as the callback function) passed as a parameter to the **hapiRecRecognise** call. This call back function is called at the end of the utterance to process the final results and optionally after a certain number of frames to process intermediate results.

```
int32 exRecognise(ApplRecInfo *ari,ApplSrcInfo *asi)
{
   hapiTime obsAdv;
   int32 fail=0,nFr=0,n;

   fail = fail || (hapiCoderPrepare(asi->coder)<0);

   /* Could wait here before starting audio input */
   fail = fail || (hapiSourceStart(asi->src)<0);

   hapiRecRecognise(ari->rec,ari->net,asi->coder,ari->final,
                    ari->cbi,ari->inter,exCallbackFunc);
   while(!ari->ready) sleep(1);

   fail = fail || (hapiCoderComplete(asi->coder)<0);

   return(fail?-1:ret);
}
```

Since an application may want to process both intermediate and final results (but in different) ways it makes sense to create two separate results objects; one to hold the intermediate results and the other to hold the final results. This makes it easy for the application to determine whether the callBack function is being called to process the final or the intermediate results.

The following example callback function shows one way in which that could be done.

```
int32 exCallbackFunction(hapiResObject res)
{
   ApplRecInfo *ari;
   int32 ret=0,fail=0;

   fail=fail || ((ari=hapiResGetAttachment(res))==NULL);
   if (ari->final==res) {
      if (ari->finalFunc!=NULL) ret=ari->finalFunc(res);
```

```
      ari->ready=1; /* Recognition now complete */
   }
   else {
      if (ari->interFunc!=NULL) ret=ari->interFunc(res);
   }
   fail = (hapiResClear(res)<0) || fail;
   /* Recognition always complete after error */
   if (fail || ret<0) ahi->ready=1;
   return(fail?-1:ret);
}
```

Chapter 16

hapiResObject

The results object provides the application with access to the single best hypothesis for each utterance. When alternative answers are required a lattice may be generated from the results.

Results fall into two categories.

- Intermediate results.

  These results are available during the processing of the utterance. The most likely hypothesis can change as recognition proceeds and so these are mainly useful for indicating progress to the user. Normally intermediate results will not contain multiple alternatives and only the single currently most likely hypothesis will be accessible.

- Final results.

  Once all the observations in an utterance have been processed the final results are available. In simple applications access to the string of words recognised will be all that is required. However, more complex applications may need to do further processing of the results.

Both categories of results are held in the same type of object and support the same range of access functions with the exception that only final results can be used to generate a lattice of possibilities.

16.1 Usage

Results objects are created in much the same way as all other objects:

```
   res=hapiCreateResObject(ari->rec,HAPI_RI_htk);
   if (res==NULL || hapiResSetAttachment(res,ari)<0 ||
       hapiResGetAttachment(res)!=ari) return();
.
.
.
   hapiDeleteResObject(res);
```

The second argument to this function specifies what type of results should be stored in the results object. When the recogniser and the application are running in a single

process all types of results supported by the current decoder will be accessible from the results object. However when the recogniser is running remotely from the application this information has to be transfered in a potentially costly manner. In this case only the information explicitly requested by the application will be transfered, other information (which may be available to the recogniser) will not be copied.

If the requested information is not available from the chosen recogniser `NULL` will be returned and no results object created.

Different types of results information are requested by bit flags. These flags, along with a description of the type of results information for each, are listed in the table below.

Flag	Description
`HAPI_RI_inter`	Intermediate results.
`HAPI_RI_final`	Utterance final results.
`HAPI_RI_lat`	NBest results as lattice.
`HAPI_RI_trace`	Trace information.
`HAPI_RI_lev_word`	Word level results.
`HAPI_RI_lev_phone`	Phone level results.
`HAPI_RI_inc_times`	Boundary times.
`HAPI_RI_inc_like`	Total likelihoods.
`HAPI_RI_inc_acoustic`	Explicit access to acoustic likelihoods
`HAPI_RI_inc_lm`	to language model likelihoods.
`HAPI_RI_inc_pron`	to pronunciation likelihoods.
`HAPI_RI_inc_background`	to background model likelihoods.
`HAPI_RI_inc_confidence`	to confidence scores
`HAPI_RI_def_htk`	Results supported by default recogniser.
`HAPI_RI_def_htk_phone`	Results supported by default recogniser which include phone level alignment.

Each available recogniser has a predefined value for this flag indicating the information available by default. This will normally be used when creating results objects in single process applications. For remote servers or multi-process applications the flags should indicate the information required.

For both categories of results *trace* information is available. This takes the form of a text string containing selected information about the hypothesis. This can include number of observations processed, sequence of words hypothesised, likelihood of those words and average search complexity.

For both intermediate and final results the trace information is accessed by the following function.

```
char8 buf[81];
hapiResTraceInfo(res,HAPI_TF_frame+HAPI_TF_word+HAPI_TF_average,81,buf);
```

The second argument to this function is formed by a bitwise combination of the following flags.

Flag	Format	Meaning
HAPI_TF_frame	@%d	Current frame number
HAPI_TF_word	%s	Word
HAPI_TF_pron	%s#%d	Word and pronunciation
HAPI_TF_like	(%.2f)	Word likelihood
HAPI_TF_acoustic	(.. a=%.2f ..)	Word acoustic likelihood
HAPI_TF_lm	(.. l=%.2f ..)	Word lm likelihood
HAPI_TF_dict	(.. p=%.3f ..)	Word pron likelihood
HAPI_TF_time	-%d-	Word boundary times
HAPI_TF_average	[..%.2f..]	Average frame likelihood
HAPI_TF_total	A=%.2f L=%.2f	Total likelihoods
HAPI_TF_active	[..N=%.1f]	Average number of active nodes during recognition

16.2 Word level results access

HAPI provides functions which allow access to results information on a word by word basis. The code fragment below shows examples of functions returning the likelihood, segmentation times, output symbols etc for individual words.

```
n=hapiResAllWordIds(res,NULL);time=0.0;like=0.0;
for(i=1;i<=n;i++) {
   p=hapiResWordId(res,i);
   wrd=(p>0 && hapiResWordName(res,i,buf1)>0)?buf1:"<unk>";
   sym=(hapiResOutSym(res,i,buf2)>0)?buf2:"[NULL]";
   q=hapiResWordPronId(res,i);
   if (hapiResWordTimes(res,i,&st,&en)<0) st=en=-999.999;
   time+=en-st;
   aclk=hapiResWordScore(res,i,HAPI_SCORE_acoustic);
   lmlk=hapiResWordScore(res,i,HAPI_SCORE_lm);
   like+=hapiResWordScore(res,i,HAPI_SCORE_total);
   printf(" %.2f-%.2f %s (%d %d %s) Ac=%.2f LM=%.2f\n",
          st/1000.0,en/1000.0,wrd,p,q,sym,aclk,lmlk);
}
printf(" Total %.2fs = %.2f : Av = %.3f\n",time/1000.0,like,like/time*10.0);
```

There are also functions that provide access to the whole utterance results with a single call.

```
if ((n=hapiResAllWordIds(res,NULL))<=0) return; /* No results */
if ((words=malloc(sizeof(hapiWordId)*n))==NULL ||
    (names=malloc(sizeof(char8*)*n))==NULL ||
    (syms=malloc(sizeof(char8*)*n))==NULL)
   exit(1); /* No memory is a fatal error */
words--; names--; syms--;
if (n==hapiResAllWordIds(res,words) &&
    n==hapiResAllWordNames(res,names) &&
    n==hapiResAllOutSyms(res,syms))
   for (i=1;i<=n;i++)
      printf(" %3d - %-19s (%3d %s)\n",i,names[i],words[i],
             (syms[i]==NULL)?"[NULL]":syms[i]);
/* Free these now although they remain valid until dictionary deleted */
free(words+1); free(names+1); free(syms+1);
```

16.3 Phone level results access

When the recogniser stores information at the phone level the results object provides the application with access to results at the phone level.

The following code fragment could be used to print phonetic alignment and score information if it were available.

```
if ((m=hapiResAllPhoneIds(res,i,pron))>0) {
   for (j=0;j<m;j++) {
      id=hapiResPhoneId(res,i,j);
      ptr=(id>0 && hapiDictPhoneName(dict,id,buf)>0?buf:"<unk>");
      if (hapiResPhoneTimes(res,i,j,&st,&en)<0) st=en=-999.999;
      aclk=hapiResPhoneScore(res,i,j,HAPI_SCORE_total);
      printf("  Phone %d == %d/%d (%s) %.2f-%.2f - %.2f\n",
             i,pron[i],id,ptr,st/1000.0,en/1000.0,aclk);
   }
}
```

Part III

Application Issues

Chapter 17

System Design

The next two chapters describe other issues relevant to incorporating speech recognition into applications.

Although the whole of the HAPI library has been described there are many issues involved in producing a real application that have not been covered.

Some of these concern design of the recognition system itself to ensure that it is robust to recognition errors and can tolerate incorrect use by naive users. Also, it must be able to work correctly in adverse environments and when errors occur. Other points of concern relate to the mode of operation, with all the example code assuming the user sitting in front of a computer with input and output not just via the audio channel but with access to both visual feedback and keyboard input. This chapter provides discussion of these issues and explains in a general way how the tutorial dialer application can be extended to make it operate solely over an audio channel and make it more robust in use.

The other important issue that has not been covered is the "understanding" of recognition results and how to undertake a dialog with the user where the application tries to ensure that its actions correspond to what the user desires. The final chapter is a discussion of semantic processing and outlines some simple techniques used in another demonstration system, a voice controlled email system.

Although these chapters outline some techniques that can be used in producing real applications they are by no means complete and are intended to get the programmer thinking about the problems rather than present complete solutions.

17.1 Hands-free operation with HAPI

To be useful the dialer really has to work in a hands free manner. If the user has a keyboard around this will probably provide a quicker and easier method of selecting numbers. However if the user is using a hands-free phone (or just has access to a telephone handset) then dialing using both numbers and peoples aliases using voice commands would be useful.

For example the dialer could be modified to work in the following manner

Prompt *Please state the name or number you wish to contact.*

Reply Call Entropic.

Prompt *Dialing 01223 302651.*

Reply Stop.

Prompt *Dialing aborted. Please state the name or number you wish to contact.*

Reply Call Julian Odell.

Prompt *Dialing 01223 370720.*

HAPI includes basic facilities to support dialogs with the user. These include playing audio prompts to the user (which can be read from files or synthesised by the application) and switching syntax networks from utterance to utterance. Although the available playback facilities are simple the application does have the ability to perform complex offline generation of the waveform[1].

During the development of the application both the networks and the pre-recorded prompts need to be designed and produced before the application code. Once these are available (together with a top level flowchart for the dialog) the programmer can begin to structure the application code to reflect the course of the dialog. Figure 17.1 shows how a simple dialing dialog may be structured.

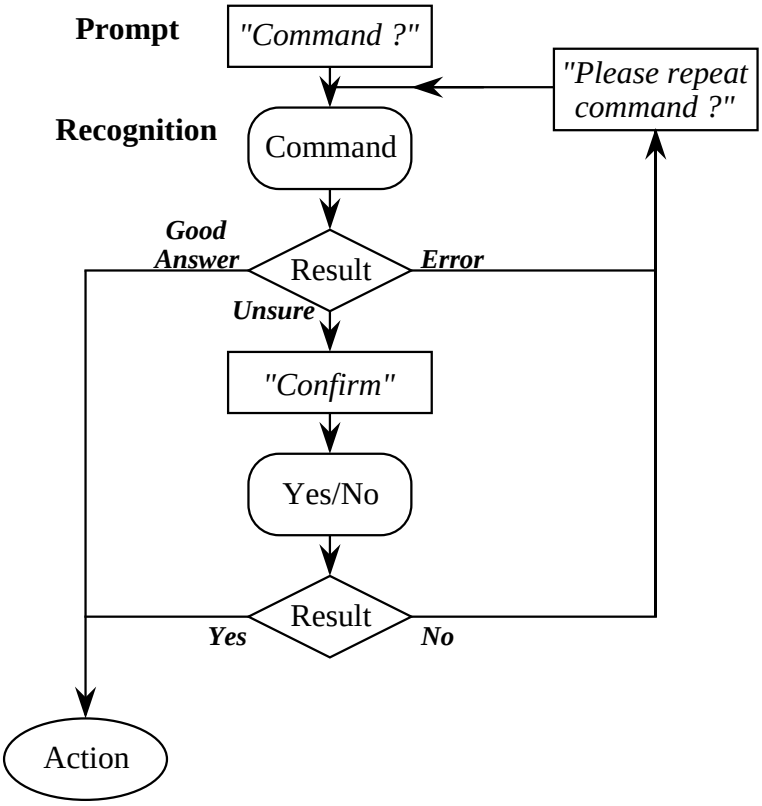

Fig. 17.1 An example dialog

[1]Currently HAPI itself does not contain support for speech synthesis direct from text although it can concatenate waveform fragments to perform simple synthesis. However, Entropic does offer the TrueTalk speech synthesis package to application developers.

As the application processes the results of each utterance, it chooses the appropriate action based on the recognised response. This may take the form of some final action (such as dialing a number) or switching the recogniser to use the network appropriate for the next stage of the dialog whilst playing the user a connecting prompt.

17.2 Managing dialogs

The most important aspect in the design of a dialog is ensuring that the user can convey the required information to the system as quickly and accurately as possible. Ensuring that this is the case will often require several iterations around the design process. As testing phases become more extensive and more representative, they will highlight different modes of user interaction that did not occur in earlier stages necessitating further dialog refinement. This is particularly true when mistakes are made or recognition is poor. Since the applications designer will probably be aiming for high success rates, problems will only occur in a small number of cases until extensive field testing (with thousands of trials) is possible.

Initial design can normally be carried out direct from the task specification with initial "expert" knowledge of what will be said to each prompt decided by the system designer. They can produce the first dialogs which despite a simple fixed structure will cope with the majority of user interactions and will be suitable for initial trials and accuracy measurements. Careful wording of the questions asked at each stage can often help to standardise the response from users. This in turn will help to ensure that the recogniser performs accurately and minimises problems due to the user saying unexpected phrases. However, a good out-of-grammar rejection model or consistent use of confidence scores to reject unexpected utterances will help the reliability and usability of the system.

Once this initial design and the simple standard networks have been created and the viability of the concept tested, the system designer needs to start thinking about how to cope with mistakes and errors.

17.3 Robustness

In a perfect world the speech recogniser would never make mistakes. However with current automatic speech recognition some errors are inevitable, as even people are fallible in this way. Fortunately people are very good at interacting to clarify answers and correct any errors that have been made. This is unfortunate for the designer wishing to incorporate speech recognition into their application as it means that users expect automatic systems to appear similarly clever. Adding these abilities to an application is an important stage in making the product usable by both experienced and naive users when the underlying speech recognition is working well or poorly.

Recognition errors occur when the spoken words do not match the equivalent acoustic models as closely as some other sequence. When this occurs the hypothesis for the utterance will not match the spoken words and the application could take the wrong action as a result. These errors can have many causes, some short term which will probably not occur if the utterance is repeated and some long term which will not be helped by simple repetition. This means the application can initially ignore a particular utterance and ask the user to repeat the command. However, it must

be able to go further and simplify the recognition task by asking different questions which have less variable answers.

The simplest strategy for ensuring correct operation is to ensure that every action is explicitly confirmed by the user. However this can become tedious if every interaction with the user requires a separate confirmation step, especially when the user is familiar with the system. Consequently at each stage of the dialog the developer needs to carefully weigh up the chance of errors occurring with the effect of those errors and the need for a responsive system in order to decide upon the level of confirmation required from the user. Ideally this will be adaptive so that new users are given more opportunity to correct mistakes whilst expert users and those for which the system is known to work well are given the speedy treatment.

It is also necessary to allow the user to correct recognition mistakes and interrupt actions that may have been taken in error. The way in which these verification stages are presented to the user may be changed depending upon the position in the dialog. For our dialer errors will be more common for strings of digits than for names selected from the phone book consequently the dialog could be different depending upon what was recognised (and maybe dependent even upon the recognisers confidence that the correct words have been recognised).

For instance in the example given in section 17.1 the system started dialing the number immediately and the user had to interrupt and stop the system (in this case because they wanted to contact a particular person rather than Entropic as opposed to because of a recognition error). An alternative is to use an element of confirmation to ensure that the action performed by the application is that desired by the user.

Prompt *Please state the name or number you wish to contact.*

Reply Call Entropic.

Prompt *Dialing 01223 302651.*

To make the system more robust when the user says actual phone numbers the next stage of the dialog could be an explicit confirmation step

Prompt *Please state the name or number you wish to contact.*

Reply Call 01223 302651.

Prompt *Dial 01223 302651 ?*

Reply Yes

Prompt *Dialing*

With actual interruptions (such as the **Stop** command in the first example) the dialog will have moved onto another stage (in that example the dialog has finished and the phone number is being dialed). Consequently extra possibilities will have to be added to networks to allow for these later commands.

17.3.1 Out of grammar rejection

One important cause of recognition errors is the inability of the syntax used to parse the spoken sentence. In other words the spoken sentence cannot be generated by any path through the recognition syntax. In these cases the recogniser will inevitably

make mistakes no matter how many times the user says the same sentence or how careful they are to pronounce each word correctly.

In such cases either the recognition syntax needs to be modified to accommodate the new valid utterance or the utterance should be rejected if it is genuinely malformed. This issue, rejection of out-of-grammar utterances, has already been touched upon in section 1.1. When the user says something unexpected it is necessary to detect that the hypothesised sentence is a poor match and ensure that it is either rejected immediately or at least confirmation is required before actions are taken.

For example, if a dialer responded in the following way to an obviously out of domain sentence the user could feel genuinely agrieved.

Prompt *Please state the name or number you wish to contact.*

Reply Martha, can you make me a cup of tea ?

Prompt *Dialing 958 3423.*

By including extra paths in the network which contain generic models of speech (or by checking *confidence* scores from the results) such out-of-grammar utterances will be rejected and ignored.

17.3.2 Barge-in

Another issue that is important for unimodal audio interfaces and especially important if errors are made is known as barge-in. As users become more confident and familiar with systems they often interrupt the prompts before they have finished. Unless the system has been designed to cope with this eventuality errors may occur because:

- The beginning of the users response is missed because the system does not start listening before the prompt finishes.
- Recognition errors are made because of the increased levels of background noise due to the spoken prompt being picked up by the microphone.

The first cause of errors is relatively simple to avoid by ensuring that the system starts listenning for a response as soon as the prompt has been spoken. However, the second cause is more difficult to prevent. The simplest method is to stop the prompt as soon as the user begins to respond. This is not ideal but does limit the interference from the prompt to just the initial sound or word spoken by the user. A better solution is to monitor the prompt signal and remove the echo from the input signal as well as stopping the prompt as soon as the user has started speaking. This ensures that the whole of the user's reply can be processed uncorrupted by prompt echo and gives most accurate results.

17.3.3 Simplicity

Probably the best way to ensure that speech recognition can be used successfully in an application is to ensure that it is used as simply as possible. By ensuring that the acoustic distinctions that need to be made are clear and the choices limited the reliability of most recognition systems can be dramatically improved.

Simplifying the task is also the last resort method of ensuring that the application continues to function in adverse conditions or other circumstances when recognition is poor. Often this will necessitate a gradual fallback to simpler and simpler networks eventually reaching dialog stages with only a single word answer required. For example

Prompt *Please state yes or no. Is number 302651 correct ?*

Reply Yes

In this case the reply is a single word chosen from a pair (see figure 17.2) that are not easily confused acoustically. This ensures that the task for the recogniser is as simple as possible. Ideally the two choices would be slightly longer to increase the acoustic information used to differentiate between them. However, the desire for accurate recognition must be tempered by the need to preserve system usability (it is unlikely that the user would be happy if asked to say either *Yes, please* or *No, thank you* specifically).

Fig. 17.2 A simple choice

Notice how the instructions for what to say are given before the actual question. This ensures that the user listens to the whole of the prompt and does not try and barge in and confuse the system. For example,

Is 302651 correct ? Please state yes or no.
Yes yes yes no no.

In adverse conditions the more complex (and thus less robust) features of the system will be disabled and typically once this level of operation has been reached the ability to barge-in over the prompt will be disabled. However if the user does not realise this they may just feel that the recogniser is performing poorly as a result by the third time they have replied to the question they may just vary their response to see what happens. Consequently the reply the system actually listens to may not be the response the user wanted to give.

17.4 The dialer

Having discussed some of the general issues to do with dialogs and hands-free operation the remainder of this chapter describes how these relate to the dialer application in particular.

17.4.1 Playback of audio prompts

The use of pre-recorded prompts has been mentioned in general terms above and in the earlier parts of the book. However, a specific example of opening a file for use solely as an audio prompt may be needed.

```
fail = fail || ((src=hapiCreateSourceObject(HAPI_SRC_file))==NULL);
fail = fail || ((coder=hapiCreateCoderObject(src))==NULL);
fail = fail || (hapiSourceInitialise(src)<0);
fail = fail || (hapiCoderInitialise(coder,NULL)<0);

fail = fail || ((play=hapiCreateSourceObject(HAPI_SRC_haudio))==NULL);
fail = fail || (hapiSourceInitialise(play)<0);

if (hapiCoderPrepare(coder)<0 ||
    (nSamps=hapiCoderAccessWave(asi->coder,1,-1,NULL))) return;

fail = fail || ((waveData=malloc(sizeof(short16)*n))==NULL);
fail = fail || (hapiCoderAccessWave(asi->coder,1,-1,wave)!=nSamps);
fail = fail || IS_HAPI_NAN(sampPer=hapiSourceSamplePeriod(src,NULL));

fail = fail || (hapiSourceSamplePeriod(play,&sampPer)<0);
fail = fail || (hapiSourcePlayWave(play,nSamps,waveData)<0);
```

The code is lifted straight from the examples given in chapter 11 but a couple of additional points need to be made.

Since the application can guarantee that the data is being read from a file (and thus the source/coder combination stops as soon as it is prepared) there is no need to step through the observations before accessing the waveform.

Secondly each file could have a different sample period so the application should be careful to either playback each file at the correct rate or to check that all files have the same sample period.

17.4.2 Verification of results

Once the dialing command has been recognised the application can either act on the command immediately or can confirm that it was recognised correctly with the user. The extended text based example offered the user a confirmation stage but this was added mainly to demonstrate the use of NBest results rather than to address the issue of verification.

So after the sequence

Prompt *Please state the name or number you wish to contact.*

Reply Call `number`.

The application decides whether it is confident of having the correct number (based on whether the number is taken from the phone book or has been dictated by the user). If it is then it can begin dialing immediately after confirming the number to the user,

Prompt *Dialing* `number`.

Otherwise it can prompt for explicit confirmation with a single word.

Prompt *Dial* `number`.

Reply `Yes` or `no`.

Prompt *Dialing.*

17.4.3 Phone book addition

The use of a hands free, voice only interface use becomes much more complex when one considers the process of adding entries to the user's phone book.

Pronunciations for the words to be added can be generated in a number of ways.

- Phonetic recognition.

  The simplest way to get the sequence of phones for a new word is use a phonetic network to recognise the new name. Unfortunately the accuracy of such unconstrained phonetic recognisers is poor (rarely better then 75% accuracy and sometimes as low as 50%).

- Large vocabulary recognition.

  A network of names may give better performance, however there will normally be many confusable names and the size of the network may make this computationally undesirable.

- Spelt letter recognition.

  Having the user spell each new name will normally give best recognition accuracy. In conjunction with a phonetic dictionary that gives the pronunciation for each name, this method will give a relatively quick, simple and natural method for adding a new name.

In practice a combination approach may be required. The spelt letter recogniser is initially used to provide a spelling that is looked up in a database of names. Once a match is found pronunciations for that word (or set of words if several names match closely) can be used to build a phonetic network that is used to recognise with the spoken name. The results of this recognition can be used to rank the different possible pronunciations.

Prompt *Please state the name or number you wish to add.*

Reply Add Roger Rabbit

Prompt *Please spell the name.*

Reply R O G E R R A B B I T

Prompt *Please give number for Roger Rabbit.*

Reply 01223 987456

Prompt *Add entry for Roger Rabbit as 01223 987456 ?*

Reply Yes

This assumes that the recognition is reasonably accurate

17.4.4 Fallback simplification in adverse conditions

The above examples all assume that the recognition is reasonably accurate and that, by and large, the recognised word string matches closely with the spoken words. When this is not the case the system should degrade gracefully and be able to continue to operate even if it loses some of its usability.

For example as the user has more attempts to input a number the dialog becomes simpler and simpler.

Prompt *Please state the name or number you wish to contact.*

Reply Call 01223 324560

Prompt *Dial 01483 344760 ?*

Reply No

Prompt *Dial 01483 344560 ?*

Reply No

Prompt *Please repeat the number that you wish to call.*

The first stage in the graceful degradation is to ask for the number only. This simplifies the network by not allowing filler words (such as `Call`) and not enabling the phone book. However even this may not restrict the choice enough.

Reply 01223 324560

Prompt *Dial 01283 344560 ?*

Reply No

Prompt *Please say the number one digit at a time ending with* `done`*. If a mistake is made correct it by saying* `no` *followed by the correct digit or* `void` *to restart the number otherwise continue with the next digit after each digit is confirmed.*

Reply 0

Prompt *9*

Reply 1

Prompt *4*

Reply void

Prompt *Please start again*

This final fallback position restricts the choice at each prompt to a choice of ten digits and two command words. This is as simple as the dialer's syntax can get and so represents the final fallback position that hopefully will work well enough in all circumstances. If it doesn't the only solution is to go back to the operator (or the telephone keypad if available).

Reply 0

Prompt *0*

Reply 1

Prompt *4*

Reply no 1

Prompt *01*

Note that as the same part of the number is repeated multiple times the application could try to combine results from all recognition attempts to improve accuracy. This would be complex (since it would need scores for each of the alternatives for each attempt) but could improve recognition accuracy enough to be worthwhile in cases where the performance is very poor.

Also the way in which the digits are read back to the user is only one of a number of possible ways and is not a "recommendation" (single digit by single digit may well be a better place to start).

Reply 2

Prompt *2*

Reply 2

Prompt *2*

Reply 3

Prompt *01223*

Reply 3

Prompt *3*

Reply 2

Prompt *2*

Reply 4

Prompt *324*

Reply 5

Prompt *5*

Reply 6

Prompt *6*

Reply 0

Prompt *560*

Reply Done

Prompt *Dial 01223 324560 ?*

Reply Yes

Finally, the user has the number they require. Although this may appear to be a very long winded approach it will hopefully only be required in a very small proportion of cases and so will not, therefore, annoy the majority of users as they will never experience this mode of interaction.

Chapter 18

Extended Results Processing

This chapter describes some of the issues that arise in real applications when non-trivial processing of results is required. A voice controlled email application is used as a simple example in which the application requires some 'understanding' of, as opposed to merely recognition of, the results.

18.1 Voice controlled Email

The discussion of results processing will focus upon the email application included with HAPI. This is a simple program that implements part of a 'real' voice controlled email application. It can capture commands from the user, process these commands and to a limited extent respond to them. However, it does not actually process the user's system mailbox, nor does it send or receive email. It is simply a demonstration of a speech recogniser and some semantic processing.

The example application uses a visual interface to display selected information to the user. Speech is used to control the interface, select messages and dictate replies. The application response is reflected in changes to the display. A single, modeless input syntax is used with no option for simpler dialogs in adverse conditions. The previous chapter has covered many of the issues that arise in dialog design and in converting an application to operate using a purely audio interface. This chapter however concerns the processing of results and so concentrates on the more complex sentences that occur with modeless networks. In this example the application must be able to take appropriate action upon receiving recognised output such as.

```
Forward message number four to Bill with copies to Tony and John.
Save next message to folder personal.
Deleted message dated the tenth of November.
```

These commands are more difficult to process than the results of dialogs such as

```
prompt: What command?
user:   Send a message
prompt: To who?
user:   To John
prompt: Copies to anyone?
user:   Bill and Tony
```

When dialogs take this form each command is explicitly broken down and the resulting dialog is easier to process as it is done piece-by-piece. Of course such a dialog is much less desirable from the user's point of view.

There are many different ways in which an email application can operate. This examples assumes the use of **folders** to hold messages that initially arrive in the user's **inbox**. Once the user has read and reviewed their mail, the messages would normally be saved to a particular folder (either the **default** folder or a user defined one, such as **personal**) or deleted.

Figure 18.1 shows a typical screen with the contents of the **inbox** shown. The user has four numbered messages with the date of each message, the sender and the subject shown.

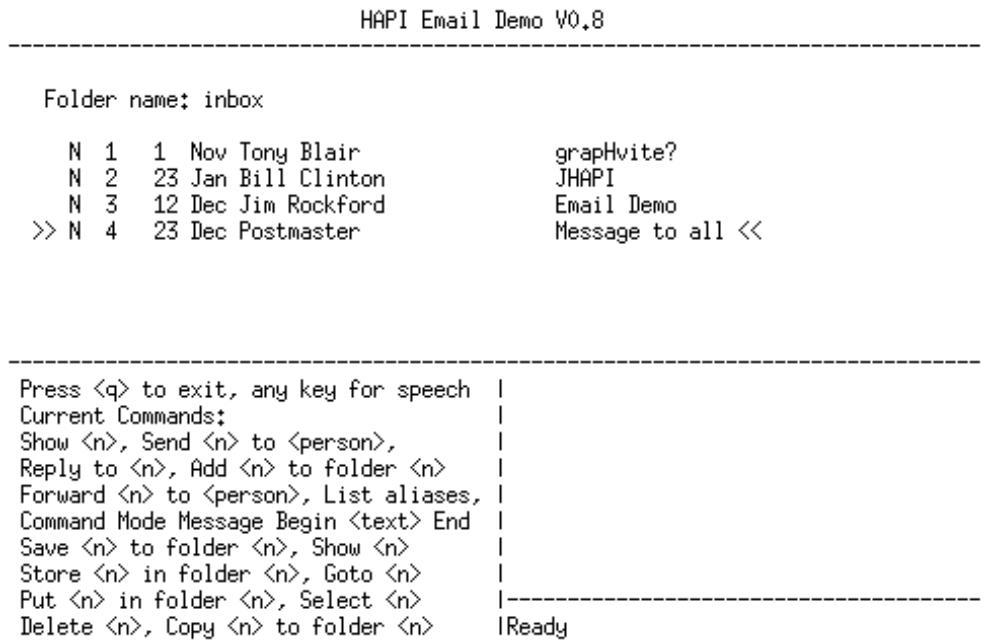

Fig. 18.1 Opening Screen of the Email Demo

In common with the tutorial dialer application the emailer implements an address book which allows people to be identified by an alias rather than by their email address. For example, the spoken word "John" could be used to indicate the email address `John.Major@westminster.co.uk`. This limits the number of people that the user can send email to but makes the system practical as otherwise an unlimited vocabulary recogniser would be needed to recognise recipients names.

The voice commands allow the user to select which messages are displayed, send, reply to or forward email to people in the user's address book[1].

A phonetic loop is used to transcribe the content of the message into a series of phones punctuated by silence gaps. The application could locate this part of the

[1] Note again that no mail is actually sent or received

message and send it as an audio file attachment. In a real system the application may use a large vocabulary dictation recogniser to actually transcribe the message into text.

Operating in this fashion limits the scope of the recognition syntax to include commands, methods of identifying messages, folders and people.

Several ways of identifying messages have been implemented.

By number The easiest way for the application to identify an individual message is have the user refer to it by its index number in the current folder.

By sender For messages sent by people appearing in the user's address book, the alias of the message sender can be used to narrow down the message selection.

By date Similarly the date on which the message was sent can be used to further narrow the selection. This does not have to be a complete date and parts of a date specification can be used to narrow down the search to a particular month or day of the month.

By Offset Phrases can be used to refer to messages relative to the currently selected message in the currently selected folder.

This gives the user several ways to refer to any message. For example the third message in figure 18.1 can be referred to as any of the following

```
MESSAGE NUMBER THREE
THE MESSAGE FROM JIM
THE MESSAGE DATED DECEMBER TWELFTH
THE PREVIOUS MESSAGE
```

When specifying a message the selection criteria is first applied to the current folder and if no matching message is found, the inbox is checked.

Figure 18.2 shows the form each of these specifications take in the recognition network. Sub-lattices (indicated by oval boxes) are used to specify complex parts of the network (such as the message `number`, `date` or `alias` for the person).

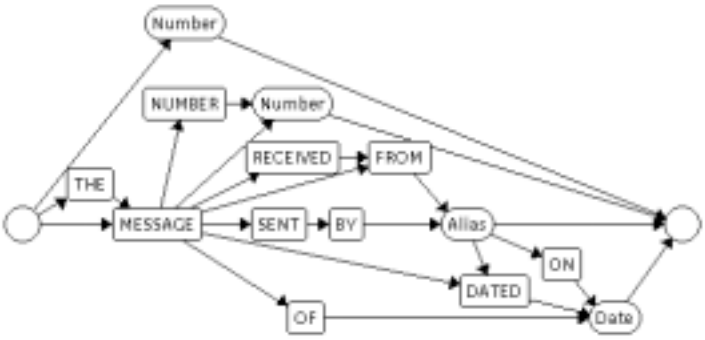

Fig. 18.2 The Messages Network

For completeness the command syntax of the email application is outlined below.

18.2 Command Syntax

This section details the form of each of the commands necessary to facilitate the major operations required by an emailer.

- `SAVE <message> TO FOLDER <folder name>`

  This command should make a copy of the selected message, and store it in the specified folder. It should mark the original message for deletion. Figure 18.3 shows the network for this command.

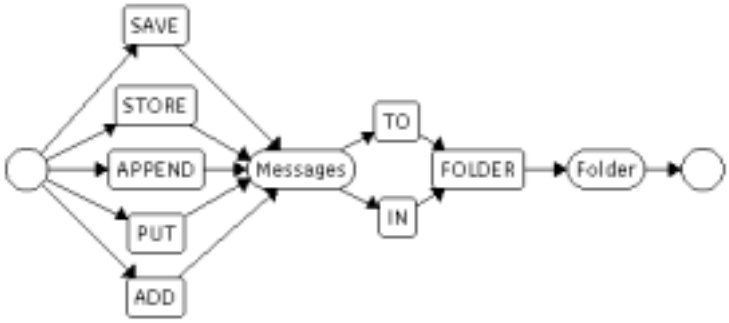

Fig. 18.3 The Save Command

- `SEND MESSAGE TO <person>`

  This should display the message to be sent and allow the next command to be the dictation of a message which is to be included in the message. Figure 18.4 shows the network for this command.

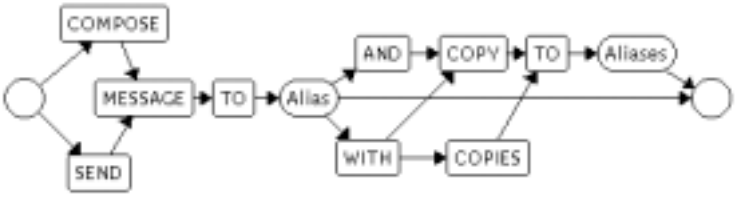

Fig. 18.4 The Compose Command

- `DELETE <message>`

  This command should mark the specified message for deletion. The network is included in figure 18.5.

- `UNDELETE <message>`

  A message marked for deletion should revert to its original state (i.e. answered, new or read). The network is included in figure 18.5.

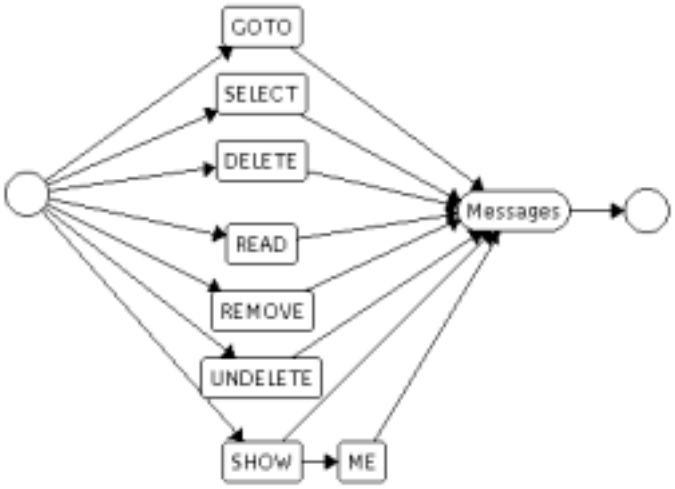

Fig. 18.5 The Simple Command Network

- `DISPLAY <message>`

  The specified message should be displayed on the screen. The network is displayed in figure 18.5.

- `SELECT <message>`

  The select message marker should be moved to the specified message. The network is shown in figure 18.5.

- `DISPLAY FOLDER <folder name>`

  The specified folder should be displayed on the screen showing all the messages saved there. The messages should have a status value indicating whether they have been read, answered or are marked for deletion. When a new folder is selected, all messages in the current folder which have been marked for deletion should be removed. The network is displayed in figure 18.6.

Fig. 18.6 The Show Command Network

- `DISPLAY FOLDERS`

  A list of the folders should be displayed, including the number of messages in each. The network is shown in figure 18.6.

- `LIST ALIASES`

  A list of the current people should be displayed. The network is shown in figure 18.6.

- `REPLY TO <message>`

  A message should be composed and sent to the person who sent the specified message. The next command can be the dictation of a message. The network is displayed in figure 18.7.

Fig. 18.7 The Reply Command Network

- `FORWARD <message> TO <person>`

  A message specified by the `<message>` should be sent to the person specified by `<person>`. The next command could be the dictation of a message if additional information is required along with the forwarded message. The network is shown in figure 18.8.

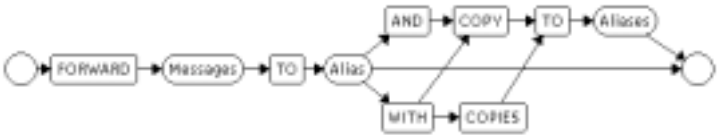

Fig. 18.8 The Forward Command Network

- `COMMAND MODE MESSAGE BEGIN <message body> END`

  This command is the dictation of a message. It can only follow a command to send, reply or forward a message. Once the dictation has finished the message is saved to the saved messages folder. The network is shown in figure 18.9

Fig. 18.9 The Command Mode Network

This syntax is by no means complete and does not cover all the commands that a real emailer needs to process. However, it is sufficient to illustrate some of the techniques applicable to more complex results processing.

18.3 Command Processing

Given the network definitions above the command processor must convert the output from the recogniser into the appropriate action.

The most general parser would just take the string of words pass them through a natural language processor which "understands" the English language and outputs some kind of command sequence to the application. This would work with any network defining a valid English sentence and could even work from typed in commands. As yet however, nobody has produced such a natural language processor and real applications will have to resort to simpler methods.

The simplest techniques will have a complex set of commands closely linked to the form of the network. Specific sequences of words or parts of the network will be used to trigger specific actions. However, this type of technique is very network and word order dependent. Small changes to the input syntax can completely break the command parser.

A more complex technique known as *form-filling*, processes the sentence with little regard for the order in which command elements are encountered. It copes better with optional parts of and small changes to the input syntax. This is the technique used in the email application.

The input sentence is parsed in word order. As they are encountered individual words (and sometimes phrases) are used to progressively enter information into a *form* that will be used to specify the final command to the application.

For this particular application and command syntax the following fields make up the *form* for each command.

Command This is a numerical value that indicates the basic command spoken. For example commands indicating that the application should display a particular message (such as `SHOW NEXT MESSAGE`) map to command code `01`.

Message number This specifies the number of the message referred to in the command. It is an index into the relevant folder. For example the command `DISPLAY MESSAGE NUMBER TWO` will map to a message number of `2`.

Sender This is a reference to the person sending the message. In the current implementation people are identified by their index into the user's address book.

Date This specifies the date on which the message was sent and is composed of two separate parts, month and day of the month[2]. Both of these are numerical quantities (months run from 1 `JAN` to 12 `DEC`).

Folder This specifies the folder mentioned in the command. This may be either the folder to display or the location to which a message should be saved.

Destination When a message is sent to a particular person this field holds the person's id (index into address book).

Copies This is a list of people (address book ids) to whom copies of the outbound message should be sent.

Message This is a place holder for the body of an email message dictated by the user. This could be the actual text itself (if a recogniser supporting dictation of email messages were used) or could be a section of audio waveform (if the application sent a voice mail message).

Obviously not all fields are needed for every single command and reserved values are used to indicate fields that did not appear in a particular command.

For example the command `DISPLAY MESSAGE NUMBER 3` would produce a form

Command `DISPLAY` (code `04`).

Message number `3`.

with all other fields set to the reserved *undefined* value.

Similarly the command `REPLY TO MESSAGE FROM BILL DATED JANUARY TWENTY THIRD` would produce

Command `REPLY` (code `05`).

Sender `BILL` (say index `01`).

Date-month `JAN` (number `01`).

Date-day `23`.

[2]For the sake of simplicity the year is ignored from the date specification

18.4 Ambiguity

Duplications of parts of the syntax within this form causes ambiguities when deciding which field in a form should be altered. This ambiguity can occur at a low level where the individual word appears in several networks and should affect different fields. For example the word `TWENTY` in the following two sentences is used in different ways.

```
REPLY TO MESSAGE NUMBER TWENTY THREE
REPLY TO THE MESSAGE DATED THE TWENTY THIRD OF JANUARY
```

In the first case the word is referring to the message number and should be used to calculate the **Message Number** field. In the second case it is part of a date and should be used to calculate the **Date-day** field.

Ambiguities can occur at a higher level as well when a complete network is used in several different places. For example, when a person's alias occurs in a command this could be because their name was used to specify the sender of a message, they were the intended recipient of a new message or should receive a copy of the message.

Ambiguities of both types can be resolved in several ways. Probably the simplest manner is to include extra information in the results which is used to modify the way in which words are processed.

This mode shifting can be implemented in two ways

- Context marks in the network

  Additional null words (which have no pronunciation and so do not effect recognition) can be included in the network and used to wrap parts of the output to indicate different processing modes. For example the date network could be expanded (as shown in figure 18.10) to include the `!DATE` and `!ENDOFDATE` markers.

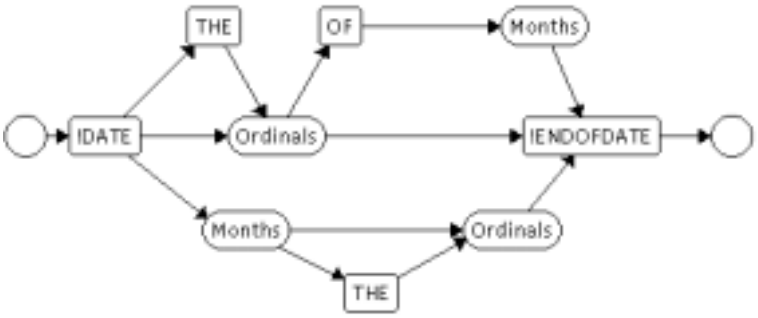

Fig. 18.10 The Date Network

If the numbers network was wrapped in the same way the results processing extended to spot these extra words, it becomes easy to differentiate different contexts. The word `TWENTY` in `MESSAGE NUMBER !NUMBER TWENTY THREE !ENDOFNUMBER` alters the message number whilst in `MESSAGE DATED !DATE THE TWENTY THIRD OF JANUARY !ENDOFDATE` it alters the date field.

- Context marks from words

  In this particular example it is easy to accomplish the mode switching from words already present in the network. For example the word `DATED` can be used to switch the parser into a mode in which numbers are associated with a date rather than with a message number. When this is possible, it helps to keep the network and dictionary uncluttered with extra null words.

Both solutions can be used to deal with the ambiguity in the meaning of aliases. Parsing mode can be switched by words such as `FROM` (in phrases such as `DISPLAY MESSAGE FROM BILL`) and `COPY` (in phrases such as `REPLY TO THIS MESSAGE WITH COPY TO JOHN`). Alternatively extra null words could be added to the network if this simpler method were not feasible (because there were no unique real words associated with either field).

18.5 Implementation

The above form filling method is implemented by adding information to the output symbols for each word.

For each word recognised the recogniser can output two items of information, the `Word` and the output symbol (`OutSym`). In practice it is up to the application writer to decide how to use this output. Usually, the `Word` is a word in a language, and the `OutSym` is used internally inside a program for ease of programming. Below are a few entries from the Email dictionary.

```
SIL               [_XXX_00]  sil
DECEMBER  [DECEMBER_MTH_12]  d iy s eh m b er sp
DELETE      [DELETE_CMD_08]  d ih l iy t sp
```

Each output symbol is split into three fields (`WORD_TYPE_VALUE`) separated by `_` characters.

`WORD` The textual representation of the word used when displaying recognition results. For example the word representing silence has an empty `WORD` field and no output is generated in the printed results when this word is recognised.

`TYPE` The type of the final field. This is used in conjunction with the current parser state to determine how the value is used.

`VALUE` The final field is the actual value for this word.

The interpretation of the three examples is as follows

`_XXX_00` Do not produce any output or modify the form when this word is recognised.

`DECEMBER_MTH_12` This word generates output of `"DECEMBER"` when recognised and sets the **Date-month** field in the form to `12`.

`DELETE_CMD_08` This word generates output of `"DELETE"` when recognised and sets the **Command** field in the form to `08`.

This makes it easy to add new ways of saying commands since the application itself does not need to be altered, only the network and dictionary. For example, a new command `REMOVE` could be added with the same functionality as `DELETE` just by adding a new word to the dictionary

```
REMOVE [REMOVE_CMD_08] r ih m uw v sp
```

and then adding this new word into the command syntax at the appropriate point. The application will take the same action because the command will affect the command form in the same way.

This makes it easy to add *filler* words to the network. These do not affect the semantics of each command but are recognised and output in the hypothesised sentence.

```
PLEASE [PLEASE_XXX_00] p l iy z sp
```

18.6 An Example Command

Consider the following recognition hypothesis

```
!SIL REPLY TO THE MESSAGE FROM TONY WITH COPIES TO JOHN AND BILL !SIL
```

This corresponds to the following sequence of output symbols

```
_XXX_00
REPLY_CMD_05
TO_XXX_00
THE_XXX_00
MESSAGE_XXX_00
FROM_MOD_02
TONY_WHO_02
WITH_XXX_00
COPIES_MOD_03
TO_XXX_00
JOHN_WHO_06
AND_XXX_00
BILL_WHO_01
_XXX_00
```

After recognition the parser checks for the presence of the end of sentence marker before processing the command. (Partial sentences can be generated by the recogniser when recognition fails and the `HREC:FORCEOUT` configuration parameter is set `TRUE`). Assuming the end of sentence marker is found the parser reads each symbol in turn, and stores the appropriate values.

`_XXX_00`	Does not affect form or word string
`REPLY_CMD_05`	Set **command** field to 5
`TO_XXX_00`	Does not affect form.
`THE_XXX_00`	Does not affect form.
`MESSAGE_XXX_00`	Does not affect form.

`FROM_MOD_02` Set the parser mode to `02` (**Sender**).

`TONY_WHO_02` Store value `02` (**Tony**) in the **Sender** field. The sender field is altered because of the current parser mode (**Sender**).

`WITH_XXX_00` Does not affect form.

`COPIES_MOD_03` Set the parser mode to `03` (**Copies**).

`TO_XXX_00` Does not affect form.

`JOHN_WHO_06` Add value `06` (**John**) to the **Copies** field. Again choice of field to alter is due to parser mode.

`AND_XXX_00` Does not affect form.

`BILL_WHO_01` Add value `01` (**Bill**) to the **Copies** field as parser still in **Sender** mode.

The above parsing constructs the following form.

Command `05`

Message number *Undefined*

Sender `02`

Date *Undefined*

Folder *Undefined*

Destination *Undefined*

Copies `06 01`

Message *Undefined*

The recognition hypothesis `REPLY TO THE MESSAGE FROM TONY WITH COPIES TO JOHN AND BILL` is the same as the recognised word sequence except that the beginning and end of utterance silence words have been quietly deleted.

Once the form has been completed the command is processed in two stages.

The first stage is to resolve any message identities. The **Sender** and **Date** fields are combined together and used to set or verify the **Message number** field. If there is ambiguity command processing is terminated and the user must provide additional clarification. If this process was done for the mailbox shown in figure 18.1 for the above command the **Message number** field would be set to `01`.

The second stage is command specific and involves ensuring that the form contains enough information to successfully perform the command and, assuming that it does, the appropriate action is taken. For the example, a reply to `Tony Blair` would be prepared and once the reply was complete copies would be sent to `Bill Clinton` and `John Major`.

Part IV

Decoders

Chapter 19

Decoder Variations

The HAPI interface can provide access to a variety of speech recognition decoders. Different decoders will have particular strengths that make them more suitable for specific tasks. The base decoder type is selected via the `hapiRecType` parameter when a `hapiRecObject` is created. At present HAPI can incorporate two main decoder types - the standard (HTK) decoder and the large vocabulary decoder (LVX). These are selected by specifying `HAPI_RT_htk` and `HAPI_RT_lvr` respectively when instances of `hapiRecObject` are created. Selecting variants of the basic decoder types allows more information to become available to the `hapiRecObject` at the results processing stage.

In addition to the two basic decoder types, some versions of HAPI incorporate further optimisations and refinements to the HTK decoder. This type of "optimised" HTK decoder will be referred to as the MVX decoder. Generally, in order to take advantage of the optimisations available in the MVX and LVX decoders, the developer must use special MVX and LVX-compliant[1] acoustic model sets. In addition, certain configuration parameters must also be set to explicitly enable the MVX/LVX optimisations. These configuration parameters will be described in the following chapters. Existing HTK toolkit customers will not be able to immediately realise the benefits of these optimisations by simply plugging their own acoustic model sets into the MVX and LVX decoders.

The table below highlights the HAPI library versions that are shipped with various Entropic products, together with the decoders that are available within each library.

Product	Library Name	Decoders available
HTK	HAPI_htk.a	HTK only
grapHvite	HAPI_mvx.a	MVX (i.e. optimised HTK)
TRANSCRIBER	HAPI_lvx.a	LVX and MVX

19.1 The HTK decoder

The standard core HTK decoder provides a versatile development tool that can be used for most tasks. However, this basic decoder may provide sub-optimal performance and for later testing and deployment,

[1] The ***grapHvite*** and TRANSCRIBER packages include such model sets.

The HTK decoder also has the ability to record the model and state level alignment information of the most likely token in each state. The alignment records contain boundary times and individual segment likelihoods. However, it is not possible to record this information in the alternative tokens and consequently alignment and NBest are mutually exclusive. Alignment is enabled by selecting one of the HTK decoder variants (`HAPI_RT_htk_phone` or `HAPI_RT_htk_state`) when the `hapiRecObject` is created.

The HTK recogniser also features a confidence scoring mechanism. The function `hapiResWordScore` can be made to return a confidence score between 0.0 and 1.0 for a word in the results object when the `hapiScoreType` is set to `HAPI_SCORE_confidence`. The confidence score can then be used for rejection if it falls below a threshold setting. The confidence score can be more reliably calculated if a background model is used. The background model is enabled by selecting the optional `hapiRecType` of `HAPI_RT_htk_bkgrd` when the `hapiRecObject` is created.

19.2 The MVX decoder

The MVX decoder offers all the features of the standard HTK decoder.At the same time, the MVX decoder features further optimisations which may offer improved recognition speed. As mentioned above, to take advantage of the MVX refinements, the user must explicitly set certain configuration parameters and make use of MVX-compliant acoustic models.

19.3 The LVX decoder

The LVX decoder is a task specific decoder that is several orders of magnitude faster than the standard HTK decoder for dictation applications. The LVX decoder has been optimised for medium to large vocabulary decoding of a fully connected network using an ngram language model to direct the search. The one variant available is `HAPI_RT_lvr_sil` which aligns the speech and silence.

19.4 Decoder development

In general, runtime-optimised decoders will provide a sub-set of the functionality of the core decoder. Although this means that the system designer must select a decoder that supports the functionality required by the particular application, once this has been identified it can be incorporated into the application with minimal changes and will immediately provide a speed increase.

As the Entropic Group improves the underlying decoder technology and associated resources (i.e. acoustic and language models) these can be incorporated into HAPI applications without needing changes to the application source code.

Chapter 20

The Core HTK Decoder

The core HAPI decoder is derived from the decoder used by the HTKtool HVITE. This was designed to be as versatile as possible and consequently does not include any recognition system or network specific computational optimisations. Often initially applications will not require maximum performance and this decoder will be suitable for preliminary testing. Later when the application is to be deployed and the computational resources required become more important, an optimised decoder can be used to reduce the computational load during recognition.

The remainder of this chapter outlines the type of recognition system that can be used with the core HAPI decoder. It explains how the HMMs, dictionary and grammar specification interact to produce a network and the way in which the decoder will then use those resources during recognition. Much of this information will be duplicated in the HTK book which contains a detailed description of how recognition systems for HTK can be constructed. This information is relevant to the recognition system designer who must decide on the type of acoustic models, dictionary and networks are needed for the particular task. Normally the application programmer will not need to be concerned with the make-up or source of the required acoustic models, dictionary or grammar specification.

Further chapters will concentrate on explaining how each decoder differs from this standard one. Consequently it is important that the user is familiar with the standard decoder before investigating the specialised ones.

20.1 Systems

Since the HAPI decoder is essentially the same as the one supplied with HTK any recognition system that can be used with HVITE should work with HAPI applications.

The core resource used by the decoder is the network (and the HMMSet to which it is linked). The decoder uses the network as a template upon which it overlays a set of active instances through which it conducts the search.

Each network is created from a syntax specification, dictionary and HMMSet. As the network is created the syntax specification (which is a word level lattice) is expanded into the model based representation required by the decoder. The way in which this expansion is performed depends upon the type of system being used. Normally the system type is determined automatically but configuration variables can be used to modify the default behaviour.

Network construction proceeds in three stages.

- Scan HMMSet and determine contexts and phone types.
- Scan lattice, dictionary and HMMSet to determine level of context dependency.
- Expand each lattice node into corresponding network nodes.

20.1.1 HMMs

The first step of the expansion process is scanning the HMMSet models list to determine its level of context dependency and the type and identity of all the phones that appear in the HMMSet.

At the end of this scan each phone will have been classified as either

- Context independent

  The phone (`x`) only occurs in a context independent form. This means that no context dependent versions (`*-x`, `x+*` or `*-x+*`) occur in the model list.

- Context free

  When a phone (`x`) does not occur as the context part of any model name (`x-*` or `*+x`) the phone is classed as context free. Context free phones are by default classed as word boundaries (so for a word internal system they block context expansion) but this can be changed by setting the configuration variable `CFWORDBOUNDARY` to `FALSE`. In this case (and for cross word systems) context expansion will skip over the context free phone

  Note that HAPI and HTK expect context free phones to be context independent (and produce errors if context dependent versions of the phone exist).

- Context dependent

  If a phone is not context independent (or by implication context free) it is classed as context dependent. When the network is expanded the logical model name for each HMM will be of the context dependent form `l-x+r`.

Taking all this together and assuming that `sil` is context independent, `sp` is context free and all other phones are triphone context dependent,

```
sil   w   er   d   sp   ih   n   t   er   ....
```

will be expanded into the model sequence

```
sil sil-w+er w-er+d er-d+ih sp d-ih+n ih-n+t n-t+er t-er+n ....
```

Assuming that this corresponded to the word sequence `SIL WORD INTERNAL` with pronunciations

```
SIL      sil
WORD     w er d
INTERNAL ih n t er n ax l
```

then the actual network would also include word end nodes signifying the word ends

```
sil SIL sil-w+er w-er+d er-d+ih sp WORD d-ih+n ih-n+t n-t+er t-er+n ....
```

20.1.2 Dictionaries

There is no expansion or manipulation of the dictionary when the network is constructed. Each word instance is expanded into each of its pronunciations which are then expanded into models. Inter-word silence must be either modelled within the network or more often within the dictionary. Typically this is done by appending an `sp` model (which is optional as it contains a transition from its entry to its exit state) at the end of each pronunciation.

20.1.3 Networks

As explained above network construction proceeds in three stages with the second stage determining the type of expansion required in the final one. There are three basic types of network.

- Context independent

  The pronunciations appearing in the dictionary specify model names directly. This can either be because the system is context independent or because the dictionary itself contains context dependent model names (which is only possible for word-internal context dependency).

- Word-internal context dependent

  The pronunciations in the dictionary are expanded to account for the surrounding models. However, at word boundaries expansion stops. For example the word

  ```
  INTERNAL ih n t er n ax l
  ```

  could be expanded into the set of triphones

  ```
  ih+n ih-n+t n-t+er t-er+n er-n+ax n-ax+l ax-l
  ```

  or with a set of right biphones

  ```
  ih+n n+t t+er er+n n+ax ax+l l
  ```

  This type of expansion can be performed upon the dictionary with the network expansion being treated as if the system were context independent.

- Cross-word context expansion

  Word internal context dependent models can be used relatively efficiently because they do not intrinsically expand the size of the recognition network. However, they obviously do not account for the effects of co-articulation between words.

  The major drawback with modelling contextual effects across word boundaries is the need to add additional nodes into the network.

  For instance the example above may need to be expanded to

```
ng-ih+n                                          ax-l+n
dh-ih+n                                          ax-l+f
 d-ih+n ih-n+t n-t+er t-er+n er-n+ax n-ax+l ax-l+k
 r-ih+n                                          ax-l+s
                                                 ax-l+t
```

The actual number of different word initial and word final models depends upon the number of different contexts occurring at that point in the network.

Single phone words are especially costly since a complete cross-bar of contexts may be needed. In the above example there are four preceding contexts (`ng`, `dh`, `d` and `r`) and five following (`n`, `f`, `k`, `s` and `t`). Expansion of a single phone word in the same position in the network would require not the single model needed in a context independent system, nor the extra seven models needed by `INTERNAL` but twenty separate models.

Often the decision about context dependency can be made automatically but if the default action needs to be changed the following configuration parameters can be used to control the behaviour of the network expansion functions.

Module	Name	Default	Description
HNet	`ALLOWCXTEXP`	`T`	Allow context expansion to get model names
	`ALLOWXWRDEXP`	`F`	Allow context expansion across words
	`FORCECXTEXP`	`F`	Force triphone context expansion to get model names (is overridden by `ALLOWCXTEXP`)
	`FORCELEFTBI`	`F`	Force left biphone context expansion to get model names (ie. don't try triphone names)
	`FORCERIGHTBI`	`F`	Force right biphone context expansion to get model names (ie. don't try triphone names)
	`CFWORDBOUNDARY`	`T`	For word internal triphone systems, treat context free phones within words as marking word boundaries.

The default configuration means that the simplest possible network which does not include cross-word context dependency will be generated. If the dictionary is closed, with every phone name appearing in the model list, no context expansion will be performed and the phone names will map directly to model names. Otherwise word internal context expansion will be used to produce HMM names.

If `ALLOWXWRDEXP` is set to `TRUE` full cross word context expansion will be performed if the dictionary is not closed after word internal expansion.

In summary the default behaviour is to perform no context expansion if the HMMSet includes all model names appearing in the dictionary. Otherwise, check to see if the HMMSet includes all word internal triphones, word initial right biphones and word final left biphones, if it does perform word internal expansion otherwise full cross word context expansion.

20.2 Decoder

The decoder uses a *token passing* paradigm to implement a modified Viterbi algorithm. The decoder moves tokens representing partial hypotheses around a network updating the token likelihoods with transition, observation and network likelihoods as well as the history information required to determine the route the token took through the network.

The decoder can store more than one token in each state thus allowing the resulting recognition results to incorporate information not just about the most likely hypothesis but also about alternatives.

Once the network has been constructed the decoder uses the network as a template for recognition. Altering the dictionary pronunciations, lattice structure and likelihoods will no longer effect recognition results (although they may alter the way in which the results are interpreted by the application).

The configuration parameters shown in table 20.1 control the various beam widths and options needed for recognition. The pruning beams can be altered with immediate effect at any time using the specific HAPI override functions and all HAPI parameters can be overridden using the generic functions before the recognition object is created.

Further configuration parameters shown in table 20.2 provide optimisation controls for the MVX decoder (and the LVX decoder). These parameters are not available with the standard HTK decoder.

20.2.1 Features

The decoder includes the ability to scale the pronunciation weights from the dictionary as well as the language model likelihoods from the network. This language model allows the application to selectively disable parts of the network by setting lattice language model likelihoods and pronunciation weights to low values (probabilities to `0.0` and log likelihoods to `log(0.0)`). If the network is then rebuilt (using **hapiBuildNetObject**) the recogniser will no longer consider the disabled parts of the network.

20.2.2 Pruning

Search complexity is controlled by pruning. Without pruning the size of the search is defined by the size of the network. Introducing pruning of the less likely paths substantially increases speed but can introduce *search errors*. A search error occurs when the most likely path through the network is not found. These errors are in addition to *modelling errors* which occur when the most likely sequence of models does not match the spoken words and network errors which occur when the grammar specification does not include the spoken word string.

The values of the various beam widths described below should be chosen carefully to ensure that the recogniser operates efficiently without introducing too many search errors.

- General Beam

  Every frame the most likely token is found and its likelihood used to set the top of a beam. Any token that falls outside of the beam is not propagated further.

- Word End Beam

Module	Name	Default	Description
HAPI	NTOKS	1	Number of tokens propagated into each state. Storing more than one token in each state allows the generation of multiple alternatives.
	LMSCALE	1.0	Language model scale factor. Alters balance between acoustic likelihoods calculated by HMMs and the prior probabilities in the network. Optimal values improve recognition accuracy.
	WORDPEN	0.0	Word insertion penalty. Also alters balance of HMMs and network.
	GENBEAM	250.0	General beam width controlling which hypotheses are pursued. Larger values can prevent certain types of errors at the expense of greater computation.
	WORDBEAM	75.0	Word-end pruning beam width. Controls extension of hypotheses at the end of words. Again larger values can increase accuracy but will require more computation.
	TMBEAM	10.0	Another pruning parameter but only used with "tied-mixture" HMMs. Value has little effect on accuracy and default is normally okay.
	MAXBEAM	2500	Maximum number of active nodes in the recognition network. Controls worst case speed of operation with smaller values resulting in faster recognition (at the expense of more errors).
HRec	FORCEOUT	F	Normally when no token reaches the end of the network by the last frame of the utterance no results are generated (indicated by number of words equal to zero). If this parameter is set to T then results are based on the most likely token anywhere in the network.

Table 20.1: Core HAPI decoder configuration parameters.

The most likely token in a word end node is found each frame and used to set the top of a beam. Only tokens that fall within that beam propagate from word-end nodes, the rest are blocked. Often the word end beam can be narrower than the general beam, decreasing the computation required for recognition without introducing significant numbers of additional search errors.

- OutP Pruning

  Individual state output likelihoods can be pruned. This beam width controls

Module	Name	Default	Description
HNET	PHNTREESTRUCT	F	If set to true, sharing is enabled, making the network more compact.
HMODEL	GAUSSOPT	F	If set to true, an optimised Gaussian calculation is performed during recognition.
	GMPOPT	F	If set to true, OutP calculations are optimised.
HAPI	GOPTBEAM	50.0	Beam width for outP pruning.

Table 20.2: Further decoder configuration parameters for decoder optimisation.

how many state likelihoods are floored. Note that unlike the other types of pruning this can result in all tokens being pruned and no recognition output generated. Typically values around 10%-25% of the main beam are suitable for most systems.

20.2.3 Considerations

As mentioned above alignment and NBest are mutually exclusive. Alignment information can only be kept for the most likely token in a state, the remaining tokens do not hold the required alignment information.

Cross word context dependent networks (which include those generated for biphone or triphone phone decoders) are both large and inefficient and may only be suitable for the simplest of tasks.

Chapter 21

The LVX Decoder

The LVX decoder is a task specific decoder that is several orders of magnitude faster than the standard HTK decoder for dictation applications. The LVX decoder has been optimised for medium to large vocabulary decoding of a fully connected network using an ngram language model to direct the search.

The decoder is in no way general purpose, it works with only a single type of HMMSet and constructs a network in a specific fashion. There is no flexibility in either the form of the network or the type of HMMs that can be used.

21.1 Systems

The major features of an LVX compliant system are

- Cross-word state-tied triphone HMMs.
- Two silence models; one optional and context free (`sp`), one longer duration and context independent (`sil`). These are automatically added and do not appear in the dictionary.
- Fully connected "word-loop" network - cannot specify grammar with word lattice.
- Back-off bigram or trigram language model.

This type of system will typically be used for dictation rather than command and control as the free-form grammar allows any word sequence to be recognised.

21.1.1 HMMs

The LVX decoder is specifically designed to use a cross word triphone model set. No other type of HMMSet is compatible with the LVX decoder. The decoder incorporates specific optimisations for continuous density tied-state diagonal covariance HMMs and models of this type should be used for greatest performance.

21.1.2 Dictionaries

During network building optional silence is added to the network and consequently the pronunciations in a dictionary for use with the LVX decoder should not include a word final silence model.

21.1.3 Networks

LVX networks are created from a word list (dictionary) and an ngram language model. The actual network is a modified word loop in which each pronunciation if followed by either the `sp` and `sil` models before being connected to the start of all words in the list. The utterance initial and utterance final words are then added to appropriate points in the network (possibly with a transition from final to initial points to allow multiple sentence utterances).

Note that this network construction assumes that a complete triphone system is being used. Context independent models cannot be used at any point in the network (apart from the special handled `sil` model). However, it is possible to clone a model which is actually context independent by adding multiple triphone entries to the HMM list all of which are tied to the context independent model. The only exception to this rule are context free models (such as `sp`) which can appear within a word but not as one of the first two or last two phones.

The names of the two silence models and the utterance initial and final words can be set by configuration parameters (and can be overridden any time before the network is created).

Module	Name	Default	Description
HAPI	`STARTWORD`	`!SENT_START`	Utterance initial word
	`ENDWORD`	`!SENT_END`	Utterance final word
	`SILNAME`	`sil`	Name of silence model
	`SPNAME`	`sp`	Name of short pause model

Table 21.1: LVX network configuration parameters.

Since the network cannot be constructed from a word lattice the **hapiBuildNetObject** function is not supported for networks with a `hapiRecType` of `HAPI_RT_lvr`.

21.2 Decoder

The LVX decoder operates in a broadly similar fashion to the standard HAPI decoder. However, it does incorporate some additional pruning as well as reordering of some operations for improved computational efficiency. For example, calculation of output likelihoods is performed in blocks the size of which can be controlled by the configuration parameter `NFWD`. The table below lists the additional configuration parameters used by the LVX decoder that effect recognition. The LVX decoder also uses the optimisation configuration parameters shown in table 20.2.

Module	Name	Default	Description
HAPI	NFWD	2	Block size for outP computations
	OUTBEAM	40.0	Beam width for outP pruning
	ZABEAM	75.0	Word end ZA beam width
HRec	OUTPRUNE	F*	Enable outP pruning
	USELOOKAHEAD	F*	Enable lookahead calculation of outP values
HNet	SENTLOOPPROB	0.0	Probability of transition from end of sentence to start of sentence word mid utterance. Set to 0.0 to disable feature.

Table 21.2: LVX decoder configuration parameters.

21.2.1 Features

Although the decoder does not support full phonetic alignment it does preserve boundary times between the silence and speech portions of the network. It also allows the language model to be used in "paragraph" mode rather than sentence by sentence by adding a transition from sentence end to sentence start.

If this feature is enabled (by setting the configuration parameter SENTLOOPPROB to a value other than 0.0) the decoder can recognise utterances of the form

```
<s> THE FIRST SENTENCE </s> <s> THE SECOND SENTENCE </s>
```

whereas normally the sentence end, sentence start transition is not possible and the resulting compound

```
<s> THE FIRST SENTENCE THE SECOND SENTENCE </s>
```

may not be recognised due to the poor language model likelihood for the sequence SENTENCE THE SECOND.

21.2.2 Pruning

The LVX decoder incorporates several types of pruning. These include the standard pruning methods used in the core HTK decoder as well as additional LVX specific ones.

- General Beam

  This operates in exactly the same way as the standard decoder. Every frame, the most likely token is found and its likelihood used to set the top of a beam. Any token that falls outside of the beam is not propagated further.

- Word End Pruning

  The LVX decoder incorporates two forms of word end pruning, one called word-end and one ZA. These operate in the same way as the word end pruning in the standard decoder with tokens blocked at word ends if they do not fall within the word end beam. However, the two types of pruning occur in slightly different places in the network. In general standard word-end and LVX specific ZA beams should be of similar size with a value around half that of the general beam.

21.2.3 Considerations

Some of the speed-up features are off by default. These are marked with asterisks in table 21.2. For most systems (including all specifically for use with the LVX decoder), the following three lines should be included in the configuration file

```
HREC: OUTPRUNE = T
HREC: USELOOKAHEAD = T
```

If these settings cause problems (possibly due to different types of models being used or other system changes) then they should be switched over one by one (starting with `OUTPRUNE = F`) until the problem disappears. However this will reduce system speed.

The accuracy of the results will be greatly reduced if the number of tokens is limited to one or two. To preserve accuracy the `NTOKS` configuration parameter should be set between `4` and `8`.

Part V

Appendices

Appendix A

HAPI Reference

This appendix contains a complete HAPI reference. It describes all types defined and used by HAPI, lists all configuration parameters that effect HAPI programs and describes every function in the HAPI library.

A.1 Data Types

All quantities (except pointers which will comply with machine default) are of a particular size. Even enumerated types are passed as 32 bit integers (`int32`). The basic types are shown below.

`char8`	– 8 bit character
`byte8`	– 8 bit signed integer
`byte8u`	– 8 bit unsigned integer
`short16`	– 16 bit signed integers
`short16u`	– 16 bit unsigned integers
`int32`	– 32 bit signed integers
`int32u`	– 32 bit unsigned integers
`bool32`	– 32 bit signed boolean value
`float32`	– 32 bit IEEE floating point values
`double64`	– 64 bit IEEE floating point values

In order to make it explicitly clear what parameters represent HAPI defines many types with simple underlying form. For example `hapiWordId`, `hapiNodeId` and `hapiStatusCode` are all actually of type `int32`.

```
typedef int32 hapiStatusCode;
```

All HAPI functions return a status code. For most functions this code is explicitly included in the integer actually returned by the function. However when the function returns a pointer or a floating point quantity (or an integer for which both positive and negative values are meaningful) the return value just indicates if an error has occurred and **hapiCurrentStatus** *must be called to find out what the error was.*

`HAPI_ERR_generic`	A generic non-fatal error occurred.

`HAPI_ERR_status` Status check of HAPI library failed.

`HAPI_ERR_fatal` Error codes less than this number are considered fatal to HAPI and no further HAPI functions should be called if such an error is returned.

`HAPI_ERR_magic` Magic number of object incorrect.

`HAPI_ERR_object_fatal` Error codes less than this number are considered fatal to the object for which they apply. The object in question should be considered corrupt and the reference to it discarded and not passed to further HAPI functions.

`Comment` Functions that return pointers use `NULL` to indicate error conditions while functions returning floating point quantities return a number `HAPI_NAN` in case of error (this should be checked with the macro `IS_HAPI_NAN` since the underlying value of `HAPI_NAN` is platform dependent).

One notable exception to having each function that returns an integer indicate an error with numbers less than zero is for functions that indicate progress made through an operation. These functions return the following type for which only a single number is reserved to indicate an error condition.

```
typedef int32 hapiRemainingStatus;
```

Many operations can provide information about the amount of processing remaining. For example the `hapiCoderObject` *can provide an estimate of the number of frames left to read in the current utterance.*

`HAPI_REM_error` The most negative integer is reserved for indicating that the function failed and an error occurred.

`HAPI_REM_more` All other values less than zero indicate that the total number of operations left if unknown but the number pending will be at least the positive of the particular value minus one. For example if the `hapiRemainingStatus` for the coder is `-101` there are 100 observations pending and an unknown number beyond that.

`HAPI_REM_done` The operation is complete.

`HAPI_REM_never` Positive values indicate the actual number of operations left before completion. Larger values are not necessarily exact but may be underestimates. The most positive integer is reserved for indicating operations that will never finish. For example if the `hapiRemainingStatus` for the coder is `100` there are at least 100 observations pending and no more than a few beyond that.

`Comment` This behaviour will mean that `hapiRemainingStatus` changes in the following manner as processing occurs. Initially -1 will be returned whilst waiting for the source of the data to start generating it. Then as data becomes available (assuming that this occurs faster

than the data is processed) the number returned will become more negative until the source of the data is finished (say it reaches -51). At this point the number will become positive because the end of operation is known (in this case starting at 50) and begin to decrease towards 0. When `HAPI_REM_done` is returned the operation is complete.

Another special case for return values is

```
typedef int32 hapiScoreType;
```

The application has access to various different types of recognition score. These are normally supplied by a single function which is passed a `hapiScoreType` *parameter to indicate which score the application requires. The following values can be used for this parameter.*

`HAPI_SCORE_lm`	Scaled language model likelihood
`HAPI_SCORE_pron`	Scaled pronunciation likelihood
`HAPI_SCORE_acoustic`	Acoustic likelihood
`HAPI_SCORE_confidence`	Confidence score between 0.0 and 1.0
`HAPI_SCORE_total`	Total likelihood

`Comment` The return value of `HAPI_NAN` is reserved for indicating fatal errors. If the score is not set a default value is returned (`0.0` for likelihoods and confidence scores).

The other major use for floating point values is times.

```
typedef double64 hapiTime;
```

All times in HAPI are measured in milliseconds and returned as double precision floating point quantities. Note that observation and sample indexes which are integers can be converted to times but are not considered to be absolute times.

`Comment` Although all times in HAPI are measured in milliseconds this is not true of HTK. HTK configuration parameters retain their own units (often 100 ns) and care must be exercised when modifying times in configuration files.

Many recognisers will make use of specialised decoders which require special networks and/or modifications to HMMSets. Such specialisations are requested by indicating recognisers of different types.

```
typedef int32 hapiRecType;
```

Each HAPI recogniser uses a particular decoder. Each decoder can have its own strengths and weaknesses. For example optimisations that improve the performance of a large vocabulary dictation system may actual slow down a small digit recogniser. Chapters 19 - 21 supply further information on the two main types of decoder available – the standard HTK and large vocbulary LVX decoders.

`HAPI_RT_htk`	The standard HTK decoder.
`HAPI_RT_lvr`	The large vocabulary dictation (LVX) decoder.

A.2 Configuration Variables

The table below describes each of the configuration variables which are specific to HAPI and can be set in a configuration file. Part of the description of each object is a listing of all configuration parameters (including HTK as well as HAPI ones) that affect the behaviour of the object in question.

Name	Default	Description
ABORTONERR	F	Causes HError to abort rather than exit.
WARNONERR	T	Display a warning every time an error condition is returned by a HAPI function.
ENDONERR	F	Display a warning and exit application every time an error condition is returned by a HAPI function.
SHOWCONFIG	F	Print the current configuration to stdout.
HMMLIST	"list"	List of names of acoustic HMMs.
HMMDIR	"."	Directory in which to find acoustic HMMs.
HMMEXT	""	Extension for HMM files.
MMF	"MODELS"	Name of Master Model File.
DICTFILE	"dict"	Name of dictionary file.
SOURCE	""	Name of data file read by source.
SCRIPT	"script"	Name of script file used by source.
DATAFORMAT	HTK	Format of data files
LATFILE	"lat"	Name of lattice file.
TIMEWIND	10.0	Accuracy to which times for NBest output must match.
LATWITHLIKES	TRUE	Include likelihoods in lattices read from disk.
SUBOBJECTS	TRUE	A `hapiLatObject` is created for each sub lattice read in as part of multi level HTK lattice.
NETFILE	"net"	Name of network file.
NTOKS	1	Number of tokens propagated into each state.
NBEST	1	Maximum number of alternative sentence hypotheses for output label file.
MLFFILE		Save label files to master label file rather than to individual files.
LMSCALE	1.0	Language model scale factor.
PRSCALE	1.0	Scale factor for pronunciation likelihoods.
WORDPEN	0.0	Word insertion penalty.
GENBEAM	250.0	General beam width controlling which hypotheses are pursued.
WORDBEAM	75.0	Word-end pruning beam width.
TMBEAM	10.0	Pruning beam width controlling the amount of Gaussian calculation performed in "tied-mixture" HMMs.
MAXBEAM	2500	Maximum number of active nodes in the recognition network.

A.3 Objects

The list of functions together with a detailed description of the configuration variables will be divided according to the object with which they associated. This starts with generic HAPI functions which are not associated with any particular object.

```
hapiStatusCode hapiInitHAPI(char8 *config, void (*error)(int32));
```

Initialise HAPI and HTK by reading specified configuration file `config` *returning a status value indicating whether HAPI initialised correctly.*

`config`	String containing name of configuration file. If `config` equals `NULL` or an empty string this defaults to `"config"`.
`error`	Either `NULL` or a pointer to a function to be called in place of exit when an error occurs within HTK/HAPI. Optionally redirects stdout/stderr to a file.
`Return`	`hapiStatusCode` indicating success or otherwise of initialisation with values less than zero indicating an error occurred. In this case no further HAPI calls should be made. It is also possible that this call will result in an internal HAPI/HTK error in which case the error function will be called and this call will never return.

```
hapiStatusCode hapiCurrentStatus(char8 *buf);
```

Return the status code of the last function call to HAPI. If this is negative (indicating an error) copy a short error message into the buffer `buf` *(if not* `NULL`*).*

`buf`	Buffer for textual error message. May be `NULL` if message not required.
`Return`	Status code from previous function call.

```
char8 *hapiOverrideConfStr(char8 *conf, char8 *s);
int32 *hapiOverrideConfInt(char8 *conf, int32 *i);
float32 *hapiOverrideConfFlt(char8 *conf, float32 *f);
int32 *hapiOverrideConfBool(char8 *conf, int32 *b);
```

Query and override generic configuration parameters.

`conf`	Configuration parameter name.
`s,i,f,b`	Pointer to new value for parameter. If `NULL` then do not set parameter just return current value.
`Return`	If value is currently set and of correct type (numeric types get converted automatically) return it, otherwise return `NULL`.
`Comment`	These functions must be called before the object related to the configuration variable is created. Although they can be used to override all configuration parameters they will normally only be used where specific override functions are not available.

A.4 hapiSourceObject

The source object represents the actual source of audio data. It is of little use by itself as the only access to the data is via a coder object connected to the source. However the coder acts as a buffer and the majority of its functions operate synchronously with the recogniser. Calls to source object functions operate synchronously with the audio, effectively in real time.

Module	Name	Default	Description
HAPI	SOURCE	""	When SOURCE is HAPI_SRC_file specify the file to load speech from.
	SCRIPT	"script"	When SOURCE is HAPI_SRC_script list of files is read from SCRIPT.
HAudio HWave HParm	SOURCERATE	0	Sample rate of source in 100ns units
	SOURCEFORMAT	HTK	File format of source
HParm	SOURCEKIND	ANON	Parameter kind of source
	AUDIOSIG	0	Specify audio signal number for delimiting utterances
	USESILDET	F	Use silence detector to delimit utterances

```
hapiSourceObject hapiCreateSourceObject(hapiSourceType type);
```

Create a Source object connected to the correct `type` *of input, initialise parameters from configuration and return a reference to object created.*

`type` Type of source to be created.

`Return` Reference to Source object or `NULL` in case of error.

`Comment` The source type can be chosen from the following values

Mode	Description
HAPI_SRC_file	Read data from a single file in the format specified in the configuration (which defaults to HTK).
HAPI_SRC_script	Read data from a series of files the names of which are read from a script file. Again the file format is specified in the configuration and defaults to HTK.
HAPI_SRC_haudio	Read data directly from the workstation's built in audio hardware. The use of the speech detector is controlled by the configuration.
HAPI_SRC_haudio_no_det	Read data directly from the workstation's built in audio hardware with the audio started and stopped explicitly by the application.
HAPI_SRC_haudio_with_det	Read data directly from the workstation's built in audio hardware using the speech detector to delimit utterances.

```
hapiStatusCode hapiDeleteSourceObject(hapiSourceObject source);
```

Free all memory associated with a Source object.

`source`	Reference to Source object.
`Return`	Status code less than zero in case of error.
`Comment`	In common with all deletion calls the reference is no longer a valid object after the call.

```
hapiStatusCode hapiSourceSetAttachment(hapiSourceObject source, Ptr ptr);
```

Set Source object general purpose attachment to `ptr`.

`source`	Reference to Source object.
`ptr`	New attachment
`Return`	Status code less than zero in case of error.

```
Ptr hapiSourceGetAttachment(hapiSourceObject source);
```

Return Source object general purpose attachment.

`source`	Reference to Source object.
`Return`	Attachment to Source object or `NULL` in case of error.

```
char8 *hapiSourceOverrideFilename(hapiSourceObject source, char8 *fn);
```

Return and optionally override location of file specifying source. Depending upon the type of the source this filename can refer to a script file containing the names of files to read, a data file containing either a waveform or parameterised speech or be completely ignored (in the case of direct audio).

`source`	Reference to Source object.
`fn`	Either `NULL` or new name for source file.
`Return`	Current value of source filename or `NULL` if not set or an error occurs.
`Comment`	A single buffer is used to hold returned strings and so the returned value will be valid until the next call to this function or the object is destroyed.

```
hapiStatusCode hapiSourceInitialise(hapiSourceObject source);
```

Connect the object to the data source and prepare for the first utterance.

`source`	Reference to Source object.
`Return`	Status code less than zero in case of error.

```
hapiRemainingStatus hapiSourceAvailable(hapiSourceObject source);
```

Return availability of source for next utterance.

`source`	Reference to Source object.
`Return`	Returns the status of the current recognition.
`Comment`	This function is only relevant for sources of type HAPI_SRC_script as this is the only source which can run out of utterances.

```
hapiStatusCode hapiSourcePlayWave(hapiSourceObject source, int32 n,
                                  short16 *data);
```

Some sources are bidirectional and can actually play waveform data (notably the direct audio source). This function plays `n` *samples from the data buffer* `data` *to the output device.*

`source`	Reference to Source object.
`n`	Number of samples to play.
`data`	The audio data in the form of signed 16 bit linear samples.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiSourceStart(hapiSourceObject source);
```

Start the source sampling. In interactive applications this event will normally be triggered by the user indicating that they wish to start speaking and that processing of the next utterance should begin.

`source`	Reference to Source object.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiSourceStop(hapiSourceObject source);
```

Stop the source sampling. In interactive applications this event will normally be triggered by the user indicating that the utterance has finished.

`source`	Reference to Source object.
`Return`	Status code less than zero in case of error.

```
hapiTime hapiSourceSamplePeriod(hapiSourceObject source,
                                hapiTime *sampPeriod);
```

Return and optionally set the sample period of the source.

`source`	Reference to Source object.
`sampPeriod`	Either `NULL` or a pointer to a `hapiTime` structure to contain the returned sampling period.
`Return`	Current sampling period or `HAPI_NAN` in case of error.

```
float32 hapiSourceCurVol(hapiSourceObject source);
```

Return estimate (in dB between 0.0 and 100.0) of current input level. The number returned is only meaningful for audio sources which are currently sampling.

`source`	Reference to Source object.
`Return`	Current volume level if measurable, a negative number if not or `HAPI_NAN` in case of error.

A.5 hapiSourceObject Drivers

The standard source object can only read from files (waveform or parameterised) or direct audio. However there is a facility to add new drivers to HAPI which allow data capture from any source using a driver written by the application programmer.

A single HAPI function is used to add the new source driver into the library. It returns the type of the new source (or a negative status code in case of error) and this type value can be used when creating new source objects that will read data from the application defined driver.

```
hapiSourceType hapiRegisterSrcDriver(hapiSrcDriverDef *def);
```

Return new source type that will utilise the application defined driver to capture data under control of HAPI.

`def`	Driver definition described below.
`Return`	New source type identifier or a status code less than zero in case of error.

HAPI has the ability to process a variety of types of data returned by the driver. The type is encoded into a single 32 bit value with each byte having a particular meaning and importance.

Byte	Mask	Information
High 0	`HAPI_SDT_base`	Basic sample type such as `HAPI_SDT_mfcc`, `HAPI_SDT_lpc`, `HAPI_SDT_fbank` and `HAPI_SDT_wave`.
1	`HAPI_SDT_variant`	Information about particular type variant (for example linear or mu-law for waveform, with deltas or normalised for MFCCs).
2	`HAPI_SDT_chans`	Number of channels in each sample. Will be 1 for single channel waveform or number of dimensions for parameterised data.
Low 3	`HAPI_SDT_bytes`	Size of data for each channel.

This allows the total size of a sample to be found using

```
size = ((type & HAPI_SDT_chans) >> 8) * (type & HAPI_SDT_bytes)
```

Some typical base types, waveform variants and complete types for common waveform variants are show in the table below.

```
typedef int32 hapiSrcDriverDataType;
```

Source drivers can supply data to HAPI in a variety of formats. The type of the data is encoded into a 32 bit integer that contains holds information about the size of the samples as well as their basic type and specific variation. Initially only the three formats described below are supported. The driver will have to convert multiple channel waveform data into a single channel and parameterised data will have to be written to a HTK format file before being processed by HAPI.

`void *gInfo;`	Initial value for a global driver hook that is passed to each driver function.
`hapiTime sampPeriod;`	Sample period of the data sourced from the driver. This can be zero indicating that the driver can operate at the rate requested when the driver is prepared or a fixed value if the driver has a fixed sampling rate.
`int32 forceDetUse;`	Value controls the use of the speech detector with data from this driver. Zero means the driver can be used with the speech detector with the channel configuration used to determine whether it is. Greater than zero indicates the driver must be used with the speech detector whilst a negative value indicates that the detector cannot be used.
`hapiSrcDriverDataType srcType;`	Defines the type of data sourced from the driver. This must be one of supported values or the **hapiRegisterSrcDriver** function will fail.
`hapiSrcDriverDataType playType;`	Defines the type of data that can be played by the driver. In the case above the driver does not support playback.

Table. A.1 The Extended Source Definition (a)

`HAPI_SDT_lin16`	Samples are 16 bit linear.
`HAPI_SDT_mu8`	Samples are 8 bit mu-law.
`HAPI_SDT_alaw8`	Samples are 8 bit a-law.

As explained above the source definition is held in a structure which holds driver parameters and functions. The first few members of the structure define the capabilities of the source driver. An explanation of the individual fields and the values in the example driver are shown in table A.1.

The remaining members of the definition are function pointers that are called by the HAPI library when it needs information from the driver, wishes to communicate information to the driver or even just wishes to give the driver an opportunity to do some processing. During processing of an utterance every frame the driver is given at least one opportunity to do some processing. In general this will occur many times each second.

The first group of functions (see table A.2) provide registration, creation and ancillary functions that are used to set up the new source driver before it is used for data capture and to shut it down once data capture has finished.

The remainder of the functions (shown in table A.3) are used to process each utterance. HAPI uses an opportunistic read method of taking data from the driver. Each time the coder object is accessed during data capture (usually twice per frame - once to determine the coder status and once to actually read the observation) the

`void (*esRegister) (hss,hsd);`
When the source driver has been successfully registered this function is called to allow the driver to allocate and initialise any global data and resources.

`void (*esShutdown) (hss);`
Called when the HAPI is (and therefore the driver should be) shutdown. This gives the driver an opportunity to free any resources allocated in the **esRegister** call.

`hapiSrcDriverObjectInfo *(*esCreate) (hss);`
Function called when a new `hapiSourceObject` of the driver's type is created. This function should allocate any additional resources needed for each source object. The return value (a pointer) is passed to each driver function associated with the source object that has just been created and can be used to pass the information specific to the particular source.

`void (*esDelete) (hss, oInfo);`
When the source object is deleted this function is called to allow the driver to free resources allocated in the **esCreate** call.

`void (*esInitialise) (hss, oInfo, coder);`
This function is called when the source object is initialised. It provides the driver with an opportunity to take note of the coder (should it be necessary to query it later) but no specific action is required.

`hapiRemainingStatus (*esSrcAvailable) (hss, oInfo);`
Return a value indicating how many more utterances are available for processing. Most drivers are always available and either always return `HAPI_REM_more` or don't implement this function. However for batch mode drivers (like the built-in script facility) can use this function to indicate to the application that the current task is finished.

`void (*esPlayWave)(hss, oInfo, n, data);`
Some source also have the ability to play an audio message to the user. When they can this function is used to play back a waveform buffer.

`hapiSrcDriverStatus *hss;`
`hapiSrcDriverDef *hsd;`
`hapiSrcDriverObjectInfo oInfo;`
`hapiCoderObject coder;`
`int n;`
`void *data;`

Argument type definitions for above functions.

Table. A.2 The Extended Source Definition (b)

driver is queried to determine how much data is available. When data is immediately available it is read and buffered within the coder straight away.

In general each utterance will be processed with the following sequence of function calls.

- `esPrepare` Prepare for start of utterance
- `esStart` Start data collection
  - `esNumSamples` Query number of sample available
  - `esGetSamples` Read those samples
  - `[esStop]` May be called to stop collection
- `esComplete` Utterance processing complete

The core of the driver (which must be implemented by every new source extension) are the final two calls **esNumSamples** and **esGetSamples**.

```
void esRegister(hapiSrcDriverStatus *hss, hapiSrcDriverDef *hsd);
```

Once the source definition passed to the **hapiRegisterSrcDriver** *function has been checked and the new driver has been registered by HAPI this function is called to allow the driver to allocate and initialise any global data and resources required.*

`hss`	Structure holding driver status information (including the global driver hook). This is initialised to indicate that no error occurred but should be altered if a driver error occurs.
`hsd`	Copy of the source definition. This is the only driver function passed the definition structure so if any of the information this holds is needed by the driver such information must be noted in this call.
`Comment`	The `hss` structure is used to return status codes and textual descriptions together with altered values of the global driver hook.

```
void esShutdown(hapiSrcDriverStatus *hss);
```

This call should free all resources allocated in the **esRegister** *call. Currently this call is never made because there is no facility to completely shutdown HAPI. However, when such a facility is added this function will be called for each registered driver*

`hss`	Driver status information.
`Comment`	The `hss` structure is used to return driver status information to HAPI.

`void (*esPrepare)(hss, oInfo, name, sampPeriod);`
When the coder is prepared for the next utterance this function is called to allow the source to perform any computationally expensive initialisation. However this call should not block (blocking should occur in the **esStart** function).
`void (*esComplete) (hss, oInfo);`
This function is called when the current utterance has been processed and the associated coder is being completed. It gives the driver an opportunity to release any utterance specific resources.
`void (*esStart) (hss, oInfo);`
Called when the application wants data capture to begin. The driver should initiate data capture and block until this has successfully occurred.
`void (*esStop) (hss, oInfo);`
Called when the application wants data capture to stop. The driver should stop data capture (but still hold the data currently buffered and waiting to be read by HAPI).
`hapiRemainingStatus (*esNumSamples) (hss, oInfo);`
This function is used by HAPI to determine the input status of the driver. Whilst data capture is in progress the returned value is a negative number one less than the number of samples available without the need to block. When capture finishes the number returned is just the number of samples left before the drivers buffer is exhausted. This function will be called several times for each frame processed and so must complete quickly.
`int32 (*esGetSamples) (hss, oInfo, n, data);`
Called when HAPI wants to read some data. This function returns the number of samples correctly read. If the data is not available when this routine is called the function should block and wait for more data since return values less than the requested number of samples are taken to mean that the end of the input has been reached.

`hapiSrcDriverStatus *hss;`
`hapiSrcDriverObjectInfo oInfo;`
`hapiTime sampPeriod;`
`int n;`
`void *data;`

Argument type definitions for above functions.

Table. A.3 The Extended Source Definition (c)

`hapiStatusCode status;`
Status code for operation. This is initialised to `HAPI_ERR_none` before each driver function call and so only needs to be updated in case of error.
`char8 statusMessage[STR_LEN];`
Status message for operation. This is initialised to an empty string but may be used to pass a textual description of any error to the user since this buffer is appended to a generic description and is made available via the **hapiCurrentStatus** function.
`void *gInfo;`
Value of global driver hook passed to each driver function.

Table. A.4 The `hapiSrcDriverStatus` structure.

```
hapiSrcDriverObjectInfo *(*esCreate)(hapiSrcDriverStatus *hss);
```

When a source object that will use this driver is created (by calling **hapiCreateSourceObject** *with the type returned by* **hapiRegisterSrcDriver** *when this driver was registered) this function is called. This provides the driver with an opportunity to allocate any object specific resources and return the object specific hook that is passed to each object dependent driver call.*

`hss`	Driver status information.
`Comment`	The `hss` structure is used to return driver status information to HAPI.

```
void (*esDelete)(hapiSrcDriverStatus *hss, hapiSrcDriverObjectInfo oInfo);
```

When a source object utilising an external driver is deleted this function is called to allow the driver to release resources allocated during driver operation and by the **esCreate** *call.*

`hss`	Driver status information.
`oInfo`	The object specific hook returned by the **esCreate** function.
`Comment`	The `hss` structure is used to return driver status information to HAPI.

```
void (*esInitialise)(hapiSrcDriverStatus *hss, hapiSrcDriverObjectInfo oInfo,
                     hapiCoderObject coder);
```

This function is called when the source object is initialised. At this time the source will have been connected to a coder and so the identity of the coder using this source is passed to allow the driver to make note of its controller.

`hss`	Driver status information.

`oInfo`	The object specific hook returned by the **esCreate** function.
`coder`	The coder connected to this source object.
`Comment`	If the driver does not need to know to which `coder` it is connected and does not perform any initialisation at this point there is no need to implement this function. As usual the `hss` structure is used to return driver status information to HAPI.

```
hapiRemainingStatus (*esSrcAvailable)(hapiSrcDriverStatus *hss,
                                      hapiSrcDriverObjectInfo oInfo);
```

This function should return a value indicating the number of utterances still to be processed. Typically this will always be `HAPI_REM_more` *indicating that the driver can generate utterances on demand. However, for batch processing drivers (such as the built-in script file facility) this call can be used to inform the application that the current task has been completed (by returning* `HAPI_REM_done`*).*

`hss`	Driver status information.
`oInfo`	The object specific hook returned by the **esCreate** function.
`Return`	An indication of the number of utterances left to process. Note that status codes are returned in the `hss` structure.
`Comment`	If this function is not implemented the driver is assumed to return `HAPI_REM_more`.

```
void (*esPlayWave)(hapiSrcDriverStatus *hss, hapiSrcDriverObjectInfo oInfo,
                   int n, void *data);
```

Some drivers can also play recorded audio to the user. When they can (and the `playType` *in the definition is not* `HAPI_SDT_none`*) this function is called when the application requests the source driver play a waveform to the user.*

`hss`	Driver status information.
`oInfo`	The object specific hook returned by the **esCreate** function.
`n`	Number of samples to play from the `data` buffer.
`data`	Audio data (in the format that the `hapiSrcDriverDef` indicates is supported.
`Comment`	The `hss` structure is used to return driver status information to HAPI.

```
void (*esPrepare)(hapiSrcDriverStatus *hss, hapiSrcDriverObjectInfo oInfo,
                  char8 *name, hapiTime sampPeriod);
```

When the coder is prepared for the next utterance the source is given an opportunity to perform any computationally expensive initialisation. However, this function should not block waiting for an external event - such blocking should occur in the **esStart** *function.*

`hss`	Driver status information.
`oInfo`	The object specific hook returned by the **esCreate** function.
`name`	File name associated with this utterance.
`sampPeriod`	Expected sample period (from configuration).
`Comment`	The `hss` structure is used to return driver status information to HAPI.

```
void (*esComplete)(hapiSrcDriverStatus *hss, hapiSrcDriverObjectInfo oInfo);
```

Once processing of an utterance is complete this function is called to allow the driver to clean up and free any resources allocated for processing this utterance.

`hss`	Driver status information.
`oInfo`	The object specific hook returned by the **esCreate** function.
`Comment`	The `hss` structure is used to return driver status information to HAPI.

```
void (*esStart)(hapiSrcDriverStatus *hss, hapiSrcDriverObjectInfo oInfo);
```

This function is called when the application requests that the source begin collecting data. This call should start the driver capturing data and block until capture is initiated.

`hss`	Driver status information.
`oInfo`	The object specific hook returned by the **esCreate** function.
`Comment`	The `hss` structure is used to return driver status information to HAPI.

```
void (*esStop)(hapiSrcDriverStatus *hss, hapiSrcDriverObjectInfo oInfo);
```

This function is called to stop the source collecting data. This call should stop the driver capturing further data but keep any buffered data until HAPI has read it.

`hss`	Driver status information.
`oInfo`	The object specific hook returned by the **esCreate** function.
`Comment`	The `hss` structure is used to return driver status information to HAPI.

```
hapiRemainingStatus (*esNumSamples)(hapiSrcDriverStatus *hss,
                                    hapiSrcDriverObjectInfo oInfo);
```

In order to avoid blocking in the driver whilst keeping up with the data as it arrives HAPI uses this function to determine how many samples to read at each opportunity. In practice this means that whilst this function indicates that a complete frame can be read without blocking HAPI will read it immediately.

`hss`	Driver status information.
`oInfo`	The object specific hook returned by the **esCreate** function.
`Return`	The returned value indicates the number of samples that are immediately available from the driver. Negative numbers imply that more data will appear and that any attempt to read more than the number buffered (one less than the absolute return value) may result in the driver blocking waiting for more samples. Positive values indicate data capture has stopped and so the driver will not block.
`Comment`	The `hss` structure is used to return driver status information to HAPI.

```
int32 (*esGetSamples)(hapiSrcDriverStatus *hss, hapiSrcDriverObjectInfo oInfo,
                      int n, void *data);
```

Read data from the driver and add it to the HAPI buffer whilst parameterising it. If the requested number of samples are not available this function should block whilst the required data is collected.

`hss`	Driver status information.
`oInfo`	The object specific hook returned by the **esCreate** function.
`n`	Number of samples requested.
`data`	Buffer to hold returned samples.
`Return`	The number of samples read correctly. If this is less than the number requested HAPI assumes that data capture has finished and the end of the utterance has been reached.
`Comment`	The `hss` structure is used to return driver status information to HAPI.

A.6 hapiCoderObject

The coder object is responsible for reading the audio data from the source, converting the data into the correct type of observations and providing the recogniser and the application with access to the data. It provides buffering with its input running synchronously with the source with as little delay as possible from real time while its output observations are read at whatever speed the application or recogniser requires.

Although the coder object provides the application with direct access to the observations their underlying structure is effectively hidden as an observation is defined as

```
typedef void *hapiObservation;
```

Actual structure of the observation is hidden and so the application does not have access to the individual elements and the only use of an observation is to pass it between objects in HAPI.

`Comment` Since the actual observation structure is hidden it is allocated by HAPI and the application can use this reference to the observation as a buffer to transfer information between HAPI objects.

Module	Name	Default	Description
HWAVE	NSAMPLES		Num samples in alien file input via a pipe
	HEADERSIZE		Size of header in an alien file
	STEREOMODE		Select channel: RIGHT or LEFT
HPARM	TARGETKIND	ANON	Parameter kind of target
	SAVECOMPRESSED	F	Save the output file in compressed form
	SAVEWITHCRC	T	Attach a checksum to output parameter file
	ZMEANSOURCE	F	Zero mean source waveform before coding
	WINDOWSIZE	256000.0	Analysis window size in 100ns units
	USEHAMMING	T	Use a Hamming window
	PREEMCOEF	0.97	Set pre-emphasis coefficient
	LPCORDER	12	Order of lpc analysis
	NUMCHANS	20	Number of filterbank channels
	LOFREQ	-1.0	Low frequency cut-off in fbank analysis
	HIFREQ	-1.0	High frequency cut-off in fbank analysis
	V1COMPAT	F	HTK V1 compatibility setting
	USEPOWER	F	Use power not magnitude in fbank analysis
	NUMCEPS	12	Number of cepstral parameters
	CEPLIFTER	22	Cepstral liftering coefficient
	ENORMALISE	T	Normalise log energy
	ESCALE	0.1	Scale for log energy
	SILFLOOR	50.0	Normalised energy silence floor in dBs
	DELTAWINDOW	2	Delta window size
	ACCWINDOW	2	Acceleration window size
	VQTABLE	NULL	Name of VQ table
	SIMPLEDIFFS	F	Use simple differences for delta calculations
	RAWENERGY	T	Use raw energy

The coder also provides direct access to the audio waveform. This makes it possible for the application to replay sections of the utterance and perform further processing of the utterance. It also includes facilities for saving the data (both observations and waveforms) used for recognition thus allowing further system development and evaluation using real data captured from the application when in use.

```
hapiCoderObject hapiCreateCoderObject(hapiSourceObject source);
```

Create a Coder object and connect it to a particular source, initialise parameters from configuration and return a reference to object created.

`source`	Source to connect to coder.
`Return`	Reference to Coder object or `NULL` in case of error.

```
hapiStatusCode hapiDeleteCoderObject(hapiCoderObject coder);
```

Free all memory associated with a Coder object.

`coder`	Reference to Coder object.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiCoderSetAttachment(hapiCoderObject coder, Ptr ptr);
```

Set the Coder object general purpose attachment to `ptr`.

`coder`	Reference to Coder object.
`ptr`	New attachment
`Return`	Status code less than zero in case of error.

```
Ptr hapiCoderGetAttachment(hapiCoderObject coder);
```

Return Coder object general purpose attachment.

`coder`	Reference to Coder object.
`Return`	Attachment to Coder object or `NULL` in case of error.

```
hapiStatusCode hapiCoderInitialise(hapiCoderObject coder, char8 *chan);
```

Initialise a `coder` *reading via particular channel* `chan`. *The source associated with the coder should be initialised prior to this call.*

`coder`	Reference to Coder object.
`chan`	Channel through which to process data.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiCoderCalibrateSpeech(hapiCoderObject coder,
                                        float32 *cal);
```

Some sources need calibrating before they can be used successfully. For example the standard energy based speech detector needs an estimate of the background noise level. This function is used to obtain information about average speech levels, background noise levels and clipping. When called it expects the user to speak normally for its duration (a few seconds) and so will require the application to prompt the user (and normally wait for the user to indicate that they are ready) before it is called.

`coder` Reference to Coder object.

`cal` Array (of size `HAPI_CAL_size`) of parameters to be measured, values less than zero indicate parameter could not be measured. If `NULL` don't measure the parameters just indicate whether they need to be measured.

`Return` A positive number if measurement is required, zero if measurement is not required (or has already been made) or a status code less than zero in case of error.

`Comment` The program can determine whether this calibration measurement is needed by calling this function with `cal` equal to `NULL` before actually prompting the user and performing the measurement. Following calibration the returned values should be indexed by the following enumerated constants (the last of which is actually the array size rather than an index into the array).

Mode	Description
`HAPI_CAL_speech`	Estimate of mean speech level in dB
`HAPI_CAL_background`	Estimate of average background noise level in dB
`HAPI_CAL_snr`	Estimate of expected speech signal to background noise level in dB
`HAPI_CAL_peak`	Estimate of peak to peak input signal amplitude ($0.0 - 1.0$)
`HAPI_CAL_size`	Number of elements in `cal` array.

```
hapiStatusCode hapiCoderCalibrateSilence(hapiCoderObject coder,
                                         float32 *cal);
```

Some sources need calibrating before they can be used successfully. This function is used to obtain information about background noise levels and expects the user to remain quiet for its duration (a few seconds).

`coder` Reference to Coder object.

`cal` Array of parameters to be measured. If `NULL` don't measure the parameters just indicate whether they need to be measured. See above for meaning of returned elements.

`Return` A positive number if measurement is required, zero if measurement is not required (or has already been made) or a status code less than zero in case of error.

`Comment` The program can determine whether this calibration measurement is needed by calling this function with `cal` equal to `NULL` before actually prompting the user and performing the measurement. In general **hapiCoderCalibrateSpeech** should be tested before this function as it will normally remove the need for this one to be used (although the converse is not necessarily true).

```
hapiStatusCode hapiCoderPrepare(hapiCoderObject coder);
```

Prepare the `coder` *prior to coding each utterance. This function should be called before the source is started by calling* **hapiSourceStart**.

`coder` Reference to Coder object.

`Return` Status code less than zero in case of error.

```
hapiStatusCode hapiCoderComplete(hapiCoderObject coder);
```

Free up the coder data for the current utterance. This call should only be made after all processing of the current utterance is complete.

`coder` Reference to Coder object.

`Return` Status code less than zero in case of error.

```
hapiStatusCode hapiCoderResetSession(hapiCoderObject coder);
```

Indicate to the coder that a new session has started and any channel adaptation should be flushed.

`coder` Reference to Coder object.

`Return` Status code less than zero in case of error.

```
hapiObservation hapiCoderMakeObservation(hapiCoderObject coder)
```

Return an observation buffer for the current coder object.

`coder` Reference to Coder object.

`Return` Pointer to observation buffer or `NULL` in case of error.

```
hapiStatusCode hapiCoderAccessObs(hapiCoderObject coder,
                                  int32 i, hapiObservation obs);
```

Once an utterance is complete (and **hapiCoderRemaining** *returns* `HAPI_REM_done`*) this function enables random access to all the observations which have been coded. It copies observation number* `i` *into the supplied observation buffer* `obs`*.*

`coder`	Reference to Coder object.
`i`	Frame number of observation to return.
`obs`	Buffer for observation (probably allocated using **hapiCoderMakeObservation**).
`Return`	The number of observations correctly read (either one if observation read okay or zero if not) or a status code less than zero in case of error.

```
int32 hapiCoderAccessWave(hapiCoderObject coder,
                          int32 st, int32 en, short16 *data);
```

Once an utterance is complete this function enables random access to the complete utterance waveform. It returns the number of 16 bit linear samples between observations `st` *and* `en` *and optionally copies those samples into a buffer* `data`*, supplied by the application.*

`coder`	Reference to Coder object.
`st`	Frame number of first observation for which the waveform is required.
`en`	Frame number of last observation for which the waveform is required.
`data`	Either `NULL` or a waveform buffer into which the required samples will be copied. Note that the buffer must be large enough - the required size can be found with an initial call to this function with `data` equal to `NULL`.
`Return`	The number of samples from `st` to `en` or zero if this part of the waveform is not available or a status code less than zero in case of error.

```
hapiStatusCode hapiCoderSaveObs(hapiCoderObject coder, char8 *fn);
```

Once an utterance is complete this function enables the parameterised data (i.e. the observations) to be saved to a file for later processing.

`coder`	Reference to Coder object.
`fn`	Name of output file.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiCoderSaveSource(hapiCoderObject coder, char8 *fn);
```

Once an utterance is complete this function enables the complete utterance waveform to be saved to a file for later processing.

`coder`	Reference to Coder object.
`fn`	Name of output file.
`Return`	Status code less than zero in case of error.

```
hapiRemainingStatus hapiCoderRemaining(hapiCoderObject coder);
```

Return an indication of the number of frames available from the coder.

`coder`	Reference to Coder object.
`Return`	Remaining status or INT_MIN in case of error.
`Comment`	This is one of the function that gives HAPI and opportunity to read, parameterise and buffer audio data. If an application wishes to delay processing (but not capture) of data this function can be called to give HAPI an opportunity to acquire and buffer any data that is available.

```
hapiRemainingStatus hapiCoderReadObs(hapiCoderObject coder,
                                     hapiObservation obs);
```

Attempt to read next observation from coder into observation buffer and return remaining status of coder.

`coder`	Reference to Coder object.
`obs`	Buffer to hold observation. Will not be altered if coder completes without another frame becoming available in which case this function returns `HAPI_REM_done`.
`Return`	Remaining status or INT_MIN in case of error.

```
hapiTime hapiCoderObservationWindowAdvance(hapiCoderObject coder);
```

Return the observation window advance (the number of milliseconds between subsequent observations).

`coder`	Reference to Coder object.
`Return`	Observation window advance or `HAPI_NAN` in case of error.

```
char8 *hapiCoderObservationParameterKind(hapiCoderObject coder);
```

Return a string describing the observation parameterisation.

`coder`	Reference to Coder object.
`Return`	Observation parameter kind or `NULL` in case of error.

A.7 hapiHMMSetObject

The HMMSet object provides the recogniser with access to the HMMs needed for recognition. The only application level access is for specifying where the HMM set is to be loaded from. The actual HMM parameters are only available within HAPI.

The following configuration parameters can have an effect on the treatment of HMMSets.

Module	Name	Default	Description
HAPI	HMMLIST	"list"	List of names of acoustic HMMs.
	HMMDIR	"."	Directory in which to find acoustic HMMs.
	HMMEXT	""	Extension for HMM files.
	MMF	"MODELS"	Name of Master Model File. or
	MMF[1-9]		Name of Multiple Master Model Files.
HMODEL	HMMSETKIND	NULL	Kind of HMM Set
	ALLOWOTHERHMMS	T	Allow MMFs to contain HMM definitions which are not listed in the HMM List

---

```
hapiHMMSetObject hapiCreateHMMSetObject(void);
```

Create an HMMSet object and initialise parameters from configuration, return a reference to object created.

`Return` Reference to HMMSet object or `NULL` in case of error.

`Comment` In common with all objects, calls to generic configuration override functions must be made before this creation call to be effective as only specific HMMSet override functions listed below will subsequently affect this object

---

```
hapiStatusCode hapiDeleteHMMSetObject(hapiHMMSetObject hmms);
```

Free all memory associated with an HMMSet object.

`hmms` Reference to HMMSet object.

`Return` Status code less than zero in case of error.

```
hapiStatusCode hapiHMMSetSetAttachment(hapiHMMSetObject hmms, Ptr ptr);
```

Set the HMMSet object general purpose attachment to `ptr`.

`hmms` Reference to HMMSet object.

`ptr` New attachment

`Return` Status code less than zero in case of error.

```
Ptr hapiHMMSetGetAttachment(hapiHMMSetObject hmms);
```

Return the HMMSet object general purpose attachment.

`hmms`	Reference to HMMSet object.
`Return`	Attachment to HMMSet object or `NULL` in case of error.

```
char8 *hapiHMMSetOverrideList(hapiHMMSetObject hmms, char8 *list);
char8 *hapiHMMSetOverrideDir(hapiHMMSetObject hmms, char8 *dir);
char8 *hapiHMMSetOverrideExt(hapiHMMSetObject hmms, char8 *ext);
char8 *hapiHMMSetOverrideAddMMF(hapiHMMSetObject hmms, char8 *mmf);
```

Specific configuration override functions for the HMMSet object configuration.

`hmms`	Reference to HMMSet object.
`list`	HMMSet list file name.
`dir`	Directory containing HMMSet HMM files.
`ext`	Extension of HMMSet HMM files.
`mmf`	Master model file name to add to current set.
`Return`	Current value of parameter or `NULL` if not set or an error occurs. See comment for extra information about **hapiHMMSetOverrideAddMMF**.
`Comment`	All four functions share a common buffer for return values. The application must copy the returned value to another location to prevent the string being over written. The return value of **hapiHMMSetOverrideAddMMF** cycles through the set of model files that make up the current set. The end of the set is indicated by a `NULL` return whilst an empty set is indicated by two `NULL` returns in a row.

```
hapiStatusCode hapiHMMSetOverrideClearMMF(hapiHMMSetObject hmms);
```

Clears the set of default model files specified in the configuration file so that only those specified via calls to **hapiHMMSetOverrideAddMMF** *will be used.*

`hmms`	Reference to HMMSet object.
`Return`	Status code less than zero in case of error.
`Comment`	Calling this function has no effect on the model files added explicitly with calls to **hapiHMMSetOverrideAddMMF**.

```
hapiStatusCode hapiHMMSetInitialise(hapiHMMSetObject hmms);
```

Load and initialise the HMMSet.

`hmms`	Reference to HMMSet object.
`Return`	Status code less than zero in case of error.
`Comment`	Following a call to this function subsequent calls to override location or make-up will be ignored.

A.8 hapiTransformObject

The transform object is responsible for performing speaker and environmental adaptation by estimating a set of transforms which can be applied to the HMMSet. The transform object is responsible for loading and saving transforms (stored in a transform model file or TMF), updating the transforms given new adaptation data, applying the transforms to an HMMSet object and even continuing adaptation from a saved TMF. The type of transformation available is chosen using the `hapiTransformType`. The possibilities are shown in table 10.1.

```
hapiTransformObject hapiCreateTransformObject(hapiHMMSetObject hmms,
                                              hapiTransformType type);
```

Create an empty transform object, and return a reference to the object created.

`hmms` Reference to HMMSet object.

`type` Type of transform to be created. The various possible `hapiTransformType` settings are shown in table 10.1. Note that `HAPI_TRANS_dynamic` is implicitly set for this function

`Return` Reference to the Transform object or NULL in case of error.

`Comment` This call is used in conjunction with **hapiTransformInitialise** to create a transform object where adaptation begins from scratch. The transform type can be chosen from the following values.

Mode	Description
`HAPI_TRANS_default`	This default setting will load a transform set, without variance transforms and without any statistics. Adaptation will not be available.
`HAPI_TRANS_dynamic`	Enable adaptation, including loading the statistics saved with the transform set. This will enable the user to continue to perform adaptation from where he/she last stopped a saved session. If this is not set, then only loading, applying and reverting a transform object is permissible.
`HAPI_TRANS_use_variance`	Create/update/apply the variance transformations.
`HAPI_TRANS_adapt_sil`	Create/update/apply the silence transformation.
`HAPI_TRANS_global`	Apply a global transformation to the HMMSet object.
`HAPI_TRANS_class`	Apply a set of class-based transformations to the HMMSet object.
`HAPI_TRANS_full`	Apply a full set of transformations to the HMMSet object. Note that this application means that the original model set state (before the transformation is applied) cannot be recovered.

```
hapiTransformObject hapiTransformLoad(hapiHMMSetObject hmms,
                                      char8 *fn, hapiTransformType type);
```

Create a transform object, load the transform and return a reference to the object created.

`hmms`	Reference to HMMSet object.
`fn`	File name of transform model file to load.
`type`	Type of transform to be created. The various possible `hapiTransformType` settings are shown in table 10.1.
`Return`	Reference to the Transform object or NULL in case of error.
`Comment`	This call initialises the transform object and loads the transform set from file.

```
hapiStatusCode hapiTransformInitialise(hapiTransformObject trans);
```

Initialise the currently specified hapiTransformObject ready for adaptation. Further calls to this function (or after **hapiTransformLoad***) clear the statistics collected by the* **hapiTransformAccumulate***, or the statistics stored in the TMF.*

`trans`	Reference to Transform object.
`Return`	Status code less than zero in case of error.
`Comment`	The permissible useful `hapiTransformType` settings are shown in table 10.1.

```
hapiStatusCode hapiDeleteTransformObject(hapiTransformObject trans);
```

Free all memory associated with a Transform object.

`trans`	Reference to Transform object.
`Return`	Status code less than zero in case of error.
`Comment`	If the transform is being used, the HMM set is reverted back to its original state before the transformObject is freed.

```
hapiStatusCode hapiTransformSetAttachment(hapiTransformObject trans, Ptr ptr);
```

Set the Transform object general purpose attachment to `ptr`.

`trans`	Reference to Transform object.
`ptr`	New attachment
`Return`	Status code less than zero in case of error.

```
Ptr hapiTransformGetAttachment(hapiTransformObject trans);
```

Return the Transform object general purpose attachment.

`trans` Reference to Transform object.

`Return` Attachment to Transform object or `NULL` in case of error.

```
char8 *hapiTransformOverrideInfo(hapiTransformObject trans,
                                 hapiTransformInfoType tinfo, char8 *str);
```

Override/retrieve information stored in the transform.

`trans` Reference to Transform object.

`tinfo` Index to an array (of size `HAPI_TINFO_size`) of parameters to override or retrieve.

`str` String to override information parameter with. If `str` is set to `NULL` then retrieve information parameter only.

`Comment` The transform information type can be chosen from the following values

Mode	Description
`HAPI_TINFO_uid`	Transform's speaker/user identifier.
`HAPI_TINFO_uname`	Transform's speaker/user full name.
`HAPI_TINFO_hmmsetid`	HMMSet identifier that this transform is associated with.
`HAPI_TINFO_rcid`	Regression class transform identifier.
`HAPI_TINFO_chan`	Transform's channel description.
`HAPI_TINFO_desc`	General description.
`HAPI_TINFO_size`	Number of elements in `tinfo` array.

```
hapiStatusCode hapiTransformApply(hapiTransformObject trans,
                                  hapiTransformType applyType);
```

Apply a transformation to the HMMSet object.

`trans` Reference to Transform object.

`applyType` The kind of transform to be applied to the HMMSet object. The various possible `hapiTransformType` settings are shown in table 10.1.

`Return` Status code less than zero in case of error.

`Comment` If adaptation is being performed, then the process of calling this function automatically updates the transform set based on any newly accumulated statistics, before the transform is applied to the HMMSet object. If the HMM set has been transformed, the HMM set is automatically reverted back to its original state, before a new transform is calculated and applied. Note that once a full transformation has been applied (`HAPI_TRANS_full`), the transformed HMMSet cannot be reverted.

```
hapiStatusCode hapiTransformRevert(hapiTransformObject trans);
```

Apply an inverse transformation to the HMMSet object to recover the original (pre-transformed) HMMSet parameters. This call can be made at any time after the function `hapiTransformApply` *has been called.*

`trans`	Reference to Transform object.
`Return`	Status code less than zero in case of error.
`Comment`	If a full transformation has been applied (`HAPI_TRANS_full`), the transformed HMMSet cannot be reverted.

```
int32 hapiTransformAccumulate(hapiTransformObject trans,
                              hapiLatObject lat, hapiCoderObject coder);
```

An alignment of the speech observations to the models in the HMMSet is performed based on the lattice and the data stored in the coder object. This alignment is used to gather the statistics necessary for calculating a transform, such that when the transform is applied to the HMMSet, it becomes more representative of the adaptation data accumulated. Note that this function must be called for the transform parameters to be updated, and should be called for each utterance.

`trans`	Reference to Transform object.
`lat`	Reference to Lattice object. The Lattice object can either be a phone/alignment lattice (generated using `hapiResLatGenerate`) or a word graph lattice generated by HAPI lattice functions. .
`coder`	Reference to a coder object. The coder object operation must be completed (i.e. that there are no more frames left to read in the current utterance) before this function is called.
`Return`	A value of less than zero if error. A return code of zero indicates a failure to align the utterance, otherwise returns the number of frames processed.

```
hapiStatusCode hapiTransformSave(hapiTransformObject transform,
                                 char8 *fn, char8 *format);
```

Save the currently specified transform set to disk (TMF), with the appropriate format.

`trans`	Reference to Transform object.
`fn`	Name of output file.
`format`	String controlling output format. Choose from "BGMS", where "B" saves in binary format, "G" save a global transform only, "M" saves an HMMSet MMF rather than a TMF and "S" saves statistics to TMF (necessary for continued adaptation if using a TMF).
`Return`	Status code less than zero in case of error.

A.9 hapiDictObject

The dictionary object is used to define the recognition vocabulary. To allow for addition of new words during application operation there are functions not just to create and query the dictionary but also ones to modify and extend it.

```
typedef int32 hapiWordId;
```

Words in the dictionary are sorted alphabetically and numbered from one. Their index is used as their word id within HAPI. The following words are treated as special cases.

`!NULL`	Word id `1` is reserved for the virtual word used for labelling points of collation in lattices which are not actually associated with a real word.
`!SUBLAT`	When a node represents a sub lattice attempting to find its word id will give a particular error status. This is a non fatal error that can be used by the application as an indication the node represents a sub lattice.

```
typedef int32 hapiPronId;
```

Pronunciations of each word are stored as an ordered list. The pronunciation id is the position within the list with the first pronunciation having an id of `1`.

`0`	A pronunciation number of zero is used to refer to all pronunciations when passed as a function parameter or means not known when returned.

```
typedef byte8u hapiPhoneId;
typedef int32 hapiXPhoneId;
```

Phone ids are again just integers but with values between 0 and 255. Actual phone ids start at 1 (with zero reserved as an invalid id).

A pronunciation is stored as a zero terminated array of phone ids. To remove the need to find the number of phones in a particular pronunciation before getting the pronunciations HAPI defines a maximum number of phones that can appear in a single pronunciation.

`HAPI_MAX_PRON_LEN`	Maximum phones in single pronunciation. May used as size of pronunciation array that can hold any pronunciation.
`Comment`	The extended phone id type is used for return values that must also indicate status. Since the basic phone type is an unsigned byte it cannot be used to return negative error codes.

Frequently each word token will represent a linguistic word that will be represented by a sequence of phonetic models but these terms are arbitrary and it is possible for example that the word in HAPI to represent a complete phrase made up of a sequence of whole word models.

Module	Name	Default	Description
HAPI	`DICTFILE`	`"dict"`	Name of dictionary file.

```
hapiDictObject hapiCreateDictObject(void);
```

Create an Dict object and initialise parameters from configuration, return a reference to object created.

`Return`	Reference to Dict object or `NULL` in case of error.

```
hapiDictObject hapiEmptyDictObject(hapiDictObject template);
```

Create an Dict object and initialise parameters from configuration, phone set from the template dictionary and return a reference to object created.

`template`	Template Dict object from which phone set is copied.
`Return`	Reference to Dict object or `NULL` in case of error.
`Comment`	This call is used to create a new dictionary intended to hold a subset of the template dictionary. When the new dictionary is created using this call pronunciations may be transferred without needing re-mapping through their textual form. The `hapiPhoneIds` are shared between both dictionaries.

```
hapiStatusCode hapiDeleteDictObject(hapiDictObject dict);
```

Free all memory associated with a Dict object.

`dict`	Reference to Dict object.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiDictSetAttachment(hapiDictObject dict, Ptr ptr);
```

Set the Dict object general purpose attachment to `ptr`.

`dict`	Reference to Dict object.
`ptr`	New attachment
`Return`	Status code less than zero in case of error.

```
Ptr hapiDictGetAttachment(hapiDictObject dict);
```

Return the Dict object general purpose attachment.

`dict`	Reference to Dict object.
`Return`	Attachment to Dict object or `NULL` in case of error.

```
char8 *hapiDictOverrideFileName(hapiDictObject dict, char8 *str);
```

Return and optionally override location of dictionary file.

`dict`	Reference to Dict object.
`fn`	Either `NULL` or new name for file.
`Return`	Current value of file name or `NULL` if not set or an error occurs.

```
hapiStatusCode hapiDictInitialise(hapiDictObject dict);
```

Load and initialise Dict object.

`dict`	Reference to Dict object.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiDictSave(hapiDictObject dict, char8 *fn);
```

Save dictionary to file.

`dict`	Reference to Dict object.
`fn`	Name of output file.
`Return`	Status code less than zero in case of error.

```
int32 hapiDictAllPhoneNames(hapiDictObject dict, char8 **names);
```

Return number of distinct phones in dictionary and optionally load `names` *with their textual names.*

`dict`	Reference to Dict object.
`names`	Either `NULL` or an array of string pointers that will be filled with names of each phone. Note that the array must be large enough - the required size can be found with an initial call to this function with `names` equal to `NULL`.
`Return`	Number of phones in phone set or a status code less than zero in case of error. Also fills in `names` array if not equal to `NULL`.

```
hapiStatusCode hapiDictPhoneName(hapiDictObject dict,
                                 hapiXPhoneId id, char8 *name);
```

Copy name of phone with identity `id` *into buffer* `name`.

`dict`	Reference to Dict object.
`id`	Phone id for which name required.
`name`	Buffer of length `STR_LEN` to hold phone name.
`Return`	Positive number if id valid, zero if id invalid or status code less than zero in case of error.

```
hapiXPhoneId hapiDictPhoneId(hapiDictObject dict, char8 *name);
```

Find id of particular phone name.

`dict`	Reference to Dict object.
`name`	Name of phone for which id required.
`Return`	Phone id if found, otherwise zero if not found or a status code less than zero in case of error.

```
int32 hapiDictAllWordNames(hapiDictObject dict, char8 **names);
```

Return number of words in dictionary and optionally load `names` *with their textual names.*

`dict`	Reference to Dict object.
`names`	Either `NULL` or an array of string pointers that will be filled with names of each word. Note that the array must be large enough - the required size can be found with an initial call to this function with `names` equal to `NULL`.
`Return`	Number of words in dictionary or a status code less than zero in case of error. Also fills in `names` array if not equal to `NULL`.

```
hapiStatusCode hapiDictWordName(hapiDictObject dict,
                                hapiWordId wid, char8 *name);
```

Copy name of word with identity `wid` *into buffer* `name`.

`dict`	Reference to Dict object.
`wid`	Word id for which name required.
`name`	Buffer of length `STR_LEN` to hold word name.
`Return`	Positive number if name valid, zero if `wid` invalid or status code less than zero in case of error.

```
hapiWordId hapiDictWordId(hapiDictObject dict, char8 *name);
```

Find id of particular word `name` *in dictionary* `dict`.

`dict`	Reference to Dict object.
`name`	Name of word for which id required.
`Return`	Word id if found, otherwise zero if not found or a status code less than zero in case of error.

```
int32 hapiDictWordAllProns(hapiDictObject dict, hapiWordId wid,
                           hapiPhoneId **prons);
```

Return number of pronunciation for word `wid` *and optionally load array* `prons` *with the actual pronunciations.*

`dict`	Reference to Dict object.
`wid`	Word id for which number of pronunciations required.
`prons`	Either `NULL` or an array of pronunciation pointers that will be filled with pronunciations of word. Note that the array must be large enough - the exact size can be found with an initial call to this function with `prons` equal to `NULL`.
`Return`	Number of pronunciations for word or a status code less than zero in case of error. Also fills in `prons` array if not equal to `NULL`.

```
int32 hapiDictWordPron(hapiDictObject dict, hapiWordId wid,
                       hapiPronId i, hapiPhoneId *pron);
```

Return number of phones in pronunciation `i` *of word* `wid` *and optionally load array* `pron` *with actual pronunciations.*

`dict`	Reference to Dict object.
`wid`	Word id for which number of pronunciations required.
`i`	Pronunciation number for which number of phones required.
`pron`	Either `NULL` or an array of phone ids to hold pronunciation of word. Note that the array must be large enough - the exact size can be found with an initial call to this function with `pron` equal to `NULL` or an array of size `HAPI_MAX_PRON_LEN` used.
`Return`	Number of phones in pronunciation or status code less than zero in case of error.

```
char8 *hapiDictPronOutSym(hapiDictObject dict, hapiWordId wid,
                          hapiPronId i, char8 *outsym);
```

Return and optionally set output symbol for pronunciation `i` *of word* `wid`.

`dict`	Reference to Dict object.
`wid`	Word id for which output symbol required.
`i`	Pronunciation number for which output symbol required. May be zero to find generic output symbol, in this case `NULL` return indicates either output symbol is empty or output symbols of different pronunciations differ.

`outsym`	If not `NULL` new output symbol for word pronunciation (all variants if `i` is `NULL`). Note that changes to output symbols do not affect the dictionary structure and so may be made without calling **hapiDictModify** and will be effective immediately in all networks using the dictionary.
`Return`	Output symbol. `NULL` may indicate error, an empty output symbol for particular pronunciation or differing output symbols for different pronunciations.

```
hapiStatusCode hapiDictModify(hapiDictObject dict);
```

Prepare and mark dictionary as modifiable.

`dict`	Reference to Dict object.
`Return`	Status code less than zero in case of error.
`Comment`	Note that networks will have to be rebuilt for pronunciation changes to have effect.

```
hapiStatusCode hapiDictFinalise(hapiDictObject dict);
```

Update dictionary to reflect changes (possibly altering word ids) and mark dictionary as static.

`dict`	Reference to Dict object.
`Return`	Status code less than zero in case of error.
`Comment`	Note that word ids may be reordered following a call to this function

---

```
hapiWordId hapiDictAddItem(hapiDictObject dict, char8 *name,
                           hapiPhoneId *pron, char8 *outsym);
```

Add a new word (and optionally a new pronunciation) to the dictionary.

`dict`	Reference to Dict object.
`name`	Name of word for addition.
`pron`	If not `NULL` new pronunciation to add.
`outsym`	Output symbol for new pronunciation. If `NULL` this defaults to the word name.
`Return`	Word id of affected word or status code less than zero in case of error.

```
hapiStatusCode hapiDictDeleteItem(hapiDictObject dict,
                                  hapiWordId wid, hapiPronId i);
```

Delete item (word or pronunciation) from dictionary.

`dict`	Reference to Dict object.
`wid`	Word id to alter.
`i`	Pronunciation number to delete, if zero then delete word and all its pronunciations.
`Return`	Status code less than zero in case of error.

A.10 hapiLatObject

Lattices are used for defining recognition syntaxes and for holding recognition results.

```
typedef int32 hapiLatNodeId;
```

Lattice nodes are numbered from 1 to `nn` *where* `nn` *is the total number of nodes in the lattice. When node ids are passed as function parameters negative values may be used to indicate offsets from the last node so that* `nid 1` *and* `nid -nn` *both refer to the same node as do* `nid nn` *and* `nid -1`.

`Comment`	If the lattice is generated (for example from a recognition results) or regenerated (after modification) the nodes will be topologically and where possible time sorted. This guarantees that the start node will always have `nid 1` and the end one `nid nn`. If the lattice is loaded from disk the numbering of nodes and arcs will be preserved until the lattice is modified.

```
typedef int32 hapiLatArcId;
```

Lattice arcs are numbered from 1 to `na` *where* `na` *is the total number of arcs in the lattice. For results lattices the first few (*`n`*) arc ids are reserved for arcs representing words along the most likely hypothesis. When arc ids are passed as function parameters negative values may be used to indicate offsets from the end of this path so that* `lid 1` *and* `lid -n` *both refer to the same arc which represents the first word of the most likely hypothesis. Whilst* `lid n` *and* `lid -1` *both refer to the last word of the most likely hypothesis.*

`Comment`	Lattices derived from results can be differentiated from simple syntax networks by finding the value of `n` (using **hapiLatSize**). For syntax networks `n` will be zero and the arcs will be unordered. Lattices derived from results will have a meaningful value for `n` and the arcs will be sorted into likelihood order with the first `n` arcs forming the most likely path through the lattice.

Since lattices can be used to define syntaxes for networks functions exist to modify them but note that the act of modifying the lattice can destroy information (such as acoustic likelihoods) that will mean that the lattice can no longer be treated as derived from results.

```
hapiLatObject hapiCreateLatObject(hapiDictObject dict, char8 *name);
```

Create a Lat object called `name` *and initialise parameters from configuration, return a reference to object created.*

`dict`	Reference to Dict object defining vocabulary of lattice
`name`	Unique name assigned to lattice. [NOT USED].
`Return`	Reference to Lat object or `NULL` in case of error.

Module	Name	Default	Description
HAPI	LATFILE	"lat"	Default name of lattice file to load.
	TIMEWIND	10.0	Time window around requested start and end times for determining whether nodes are potential start and end of alternative paths specified for **hapiLatTimeAlternatives**.
	LATWITHLIKES	TRUE	Include likelihoods in lattices read from disk. This allows the scores to be accessed and used for determining likelihoods of paths (thus allowing the generation of alternatives) but uses more memory. If lattices read from disk are just used as word syntaxes this can be set to FALSE (results lattices are always generated with likelihoods).
	SUBOBJECTS	TRUE	A hapiLatObject is created for each sub lattice read in as part of multi level HTK lattice. This allows an application to freely manipulate any lattice contained within a multi level lattice file but again requires more memory. If the application does not perform any lattice manipulation this can be safely set to FALSE.

```
hapiLatObject hapiLoadLatObject(hapiDictObject dict, char8 *fn,
                                char8 *name);
```

Create a Lat object called name*, initialise parameters from configuration and then read the lattice from file name* fn *into lattice object.*

dict	Reference to Dict object defining vocabulary of lattice
fn	Name of file from which to load lattice.
name	Unique name assigned to lattice. [NOT USED].
Return	Reference to Lat object or NULL in case of error.

```
hapiStatusCode hapiDeleteLatObject(hapiLatObject lat);
```

Free all memory associated with a Lat object.

lat	Reference to Lat object.
Return	Status code less than zero in case of error.

```
hapiStatusCode hapiLatSetAttachment(hapiLatObject lat, Ptr ptr);
```

Set Lattice object general purpose attachment.

`lat`	Reference to Lat object.
`ptr`	New attachment
`Return`	Status code less than zero in case of error.

```
Ptr hapiLatGetAttachment(hapiLatObject lat);
```

Return Lattice object general purpose attachment.

`lat`	Reference to Lat object.
`Return`	Attachment to Lat object or `NULL` in case of error.

```
hapiDictObject hapiLatDictObject(hapiLatObject lat);
```

Return Dict object defining vocabulary for lattice object `lat`.

`lat`	Reference to Lat object.
`Return`	Reference to dictionary defining vocabulary of Lat object or `NULL` in case of error.

```
hapiStatusCode hapiLatSize(hapiLatObject lat, int32 *nnp,
                           int32 *nap, int32 *np);
```

Return various numbers defining the size of lattices.

`lat`	Reference to Lattice object.
`nnp`	Either `NULL` or a pointer to integer to be set to the total number of nodes in the lattice.
`nap`	Either `NULL` or a pointer to integer to be set to the total number of arcs in the lattice.
`np`	Either `NULL` or a pointer to integer to be set to the number of word in the most likely path through the lattice. This number will only be valid for lattices derived from recognition results and will be zero otherwise.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiLatName(hapiLatObject lat, char8 *name);
```

Find the name associated with a Lattice object `lat`.

`lat`	Reference to Lat object.
`name`	Buffer of length `STR_LEN` to hold lattice name.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiLatSave(hapiLatObject lat, char8 *fn, char8 *format);
```

Write lattice to file `fn` *in format specified by* `format`.

`lat`	Reference to Lattice object.
`fn`	Name of output file.
`format`	String controlling lattice output format. "tvaldnmr" control the presence of optional fields of the same name, include the letter to include the field in the output lattice. "S" enables re-sorting of the lattice, "A" puts word labels on arcs rather than nodes, "B" writes the lattice in binary format and "I" prevents in-line output of sub lattices.
`Return`	Status code less than zero in case of error.

```
int32 hapiLatNodeFollArc(hapiLatObject lat, hapiLatNodeId nid,
                         hapiLatArcId *flinks);
```

Return the number of arcs following a particular node and optionally load `flinks` *their ids.*

`lat`	Reference to Lattice object.
`nid`	Node id for which number of followers required.
`flinks`	Either `NULL` or an array that will be filled with ids of arcs following the node `nid`. Note that the array must be large enough - the required size can be found with an initial call to this function with `flinks` equal to `NULL`.
`Return`	Number of arcs following node or a status code less than zero in case of error. Also fills in `names` array if not equal to `NULL`.

```
int32 hapiLatNodePredArc(hapiLatObject lat, hapiLatNodeId nid,
                         hapiLatArcId *plinks);
```

Return the number of arcs preceding a particular node and optionally their ids as well.

`lat`	Reference to Lattice object.
`nid`	Node id for which number of predecessors required.
`plinks`	Either `NULL` or an array that will be filled with ids of arcs preceding the node. Note that the array must be large enough - the required size can be found with an initial call to this function with `flinks` equal to `NULL`.
`Return`	Number of arcs preceding node or a status code less than zero in case of error. Also fills in `plinks` array if not equal to `NULL`.

```
hapiStatusCode hapiLatNodeSetAttachment(hapiLatObject lat,
                                        hapiLatNodeId nid, Ptr ptr);
```

Set general purpose attachment to lattice node `nid`.

`lat`	Reference to Lat object.
`nid`	Node id to set attachment for.
`ptr`	New attachment
`Return`	Status code less than zero in case of error.

```
Ptr hapiLatNodeGetAttachment(hapiLatObject lat, hapiLatNodeId nid);
```

Return general purpose attachment for lattice node `nid`.

`lat`	Reference to Lat object.
`nid`	Node id for which the attachment should be set.
`Return`	Attachment to HMMSet object or `NULL` in case of error.

```
hapiTime hapiLatNodeTime(hapiLatObject lat, hapiLatNodeId nid);
```

Return timestamp associated to lattice node.

`lat`	Reference to Lat object.
`nid`	Node id for which the time is required.
`Return`	Timestamp of node or `HAPI_NAN` in case of error.

```
hapiWordId hapiLatNodeWordId(hapiLatObject lat, hapiLatNodeId nid,
                             hapiWordId *wid);
```

Return and optionally set the word id associated with a lattice node.

`lat`	Reference to Lat object.
`nid`	Node id for which the word id is required.
`wid`	Either `NULL` or pointer to new word id for node.
`Return`	Current word id associated with node or status code less than zero in case of error.

```
hapiPronId hapiLatNodePronId(hapiLatObject lat, hapiLatNodeId nid,
                             hapiPronId *pnum);
```

Return and optionally set the pronunciation number associated with a lattice node.

`lat`	Reference to Lat object.
`nid`	Node id for which the word id is required.
`pnum`	Either `NULL` or pointer to new pronunciation number.
`Return`	Current pronunciation number associated with node or status code less than zero in case of error.

```
hapiLatNodeId hapiLatArcStart(hapiLatObject lat, hapiLatArcId lid);
```

Return the node id of the start of lattice arc `lid`.

`lat`	Reference to Lat object.
`lid`	Arc id for which the the start node id is required.
`Return`	Node id of start node of arc or status code less than zero in case of error.

```
hapiLatNodeId hapiLatArcEnd(hapiLatObject lat, hapiLatArcId lid);
```

Return the node id of the end of lattice arc `lid`.

`lat`	Reference to Lat object.
`lid`	Arc id for which the the end node id is required.
`Return`	Node id of end node of arc or status code less than zero in case of error.

```
hapiStatusCode hapiLatArcWordName(hapiLatObject lat,
                                  hapiLatArcId lid, char8 *name);
```

Find name of the word associated with lattice arc `lid`.

`lat`	Reference to Lat object.
`lid`	Arc id for which the word name is required.
`name`	Buffer of length `STR_LEN` to hold word name.
`Return`	Positive number if name valid, zero if name not set or status code less than zero in case of error.

```
hapiStatusCode hapiLatArcOutSym(hapiLatObject lat,
                                hapiLatArcId lid, char8 *sym);
```

Find name of the output symbol associated with lattice arc `lid`.

`lat`	Reference to Lat object.
`lid`	Arc id for which the output symbol is required.
`sym`	Buffer of length `STR_LEN` to hold output symbol.
`Return`	Positive number if `sym` valid, zero if output symbol not set or status code less than zero in case of error.

```
hapiStatusCode hapiLatArcTimes(hapiLatObject lat, hapiLatArcId lid,
                               hapiTime *st, hapiTime *en);
```

Find start and end times associated with lattice arc `lid`.

`lat`	Reference to Lat object.
`lid`	Arc id for which start and end times are required.
`st`	Pointer to `hapiTime` to hold returned start time.
`en`	Pointer to `hapiTime` to hold returned end time.
`Return`	Positive number if times valid, zero if times not set or status code less than zero in case of error.

```
hapiWordId hapiLatArcWordId(hapiLatObject lat, hapiLatArcId lid,
                            hapiWordId *wid);
```

Return and optionally set the id of the word `wid` *associated with lattice arc* `lid`.

`lat`	Reference to Lat object.
`lid`	Arc id for which the word id is required.
`wid`	Either `NULL` or pointer to new word id for arc.
`Return`	Current word id associated with arc or status code less than zero in case of error.

```
hapiPronId hapiLatArcPronId(hapiLatObject lat, hapiLatArcId lid,
                            hapiPronId *pnum);
```

Return and optionally set the pronunciation number `pnum` *of the word associated with lattice arc* `lid`.

`lat`	Reference to Lat object.
`lid`	Arc id for which the pronunciation number is required.
`pnum`	Either `NULL` or pointer to new pronunciation number for arc.
`Return`	Current pronunciation number associated with arc or status code less than zero in case of error.

```
float32 hapiLatArcScore(hapiLatObject lat, hapiLatArcId lid,
                        hapiScoreType type, float32 *score);
```

Return and optionally set score of tt type for lattice arc `lid` *to* `score`.

`lat`	Reference to Lat object.
`lid`	Arc id for which the score is required.
`type`	Type of score required.
`score`	Either `NULL` or pointer to new value for arc score.
`Return`	Current score associated with arc or `HAPI_NAN` in case of error.
`Comment`	Rather than return `HAPI_NAN` when scores are not set default values are returned. For example in the case of likelihoods unknown values default to zero. Note that it only makes sense (and is possible) to set a few of the scores.

```
hapiStatusCode hapiLatModify(hapiLatObject lat);
```

Prepare and mark lattice `lat` *as modifiable. Note that only structural changes such as adding or deleting either nodes or arcs require the lattice to be modifiable. Changing words or scores of nodes or arcs can be done to any lattice but remember that networks will have to be rebuilt for changes to lattice have an effect on recognition.*

`lat`	Reference to Lat object.
`Return`	Status code less than zero in case of error.

```
int32 hapiLatFinalise(hapiLatObject lat, hapiLatNodeId st, hapiLatNodeId en);
```

Update lattice `lat` *to reflect changes (possibly altering both node and arc ids) and mark lattice as static. If* `st` *and* `en` *are supplied these are verified as being the start and end nodes of the lattice.*

`lat`	Reference to Lat object.
`st`	Either zero or expected start node of lattice.
`en`	Either zero or expected end node of lattice.
`Return`	Either the number of nodes in the static lattice, or zero if the lattice could not be finalised successfully or a status code less than zero in case of error.
`Comment`	There are many reasons why a lattice cannot be finalised, most common being lack of a single start and end node (or them not matching the expected values).

```
hapiStatusCode hapiLatDeleteNode(hapiLatObject lat, hapiLatNodeId nid);
```

Delete node `nid` *(and all the arcs connected to it) from a modifiable lattice* `lat`.

`lat`	Reference to Lat object.
`nid`	Id of node to be deleted.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiLatDeleteArc(hapiLatObject lat, hapiLatArcId lid);
```

Delete arc `lid` *from a modifiable lattice* `lat`.

`lat`	Reference to Lat object.
`lid`	Id of arc to be deleted.
`Return`	Status code less than zero in case of error.

```
hapiLatNodeId hapiLatAddNode(hapiLatObject lat,
                             hapiWordId word, hapiPronId pron);
```

Add a node representing a `word` *with pronunciation* `pron` *to a modifiable lattice* `lat` *and return its id.*

`lat`	Reference to Lat object.
`wid`	Word id for new node.
`i`	Pronunciation number for new node.
`Return`	Node id of newly created node or a status code less than zero in case of error.

```
hapiLatArcId hapiLatAddArc(hapiLatObject lat,
                           hapiLatNodeId st, hapiLatNodeId en);
```

Add an arc between nodes `st` *and* `en` *in a modifiable lattice* `lat` *and return the id of the arc added.*

`lat`	Reference to Lat object.
`st`	Id of start node for new arc.
`en`	Id of end node for new arc.
`Return`	Arc id of newly created arc or a status code less than zero in case of error.

```
char8 *hapiLatNodeGetTag(hapiLatObject lat, hapiLatNodeId nid);
```

Return the semantic tag attached to node `nid` *if it is present.*

`lat`	Reference to Lat object.
`nid`	Id of node to query.
`Return`	Semantic tag associated with node or `NULL` if no semantic tag is associated with node or in case of error.
`Comment`	Note that application is responsible for copying returned string if required for later use.

```
hapiStatusCode hapiLatNodeSetTag(hapiLatObject lat, hapiLatNodeId nid,
                                 char *tag);
```

Attach a semantic tag to node `nid`.

`lat`	Reference to Lat object.
`nid`	Id of node to query.
`tag`	Semantic tag string.
`Return`	Status code less than zero in case of error

```
char8 *hapiLatNodeSublatId(hapiLatObject lat, hapiLatNodeId nid);
```

Check if node `nid` *represents a sub lattice and if so return its name.*

`lat`	Reference to Lat object.
`nid`	Id of node to query.
`Return`	Name of sub lattice associated with node or `NULL` if no sub lattice is associated with node or in case of error.
`Comment`	Note that application is responsible for copying returned string if required for later use.

```
hapiLatObject hapiLatNodeSublatObject(hapiLatObject lat, hapiLatNodeId nid,
                                      hapiLatObject sub);
```

Return and optionally set `sub` *to the sub lattice associated with lattice node* `nid`.

`lat`	Reference to Lat object.
`nid`	Id of node to query.
`sub`	Either `NULL` or reference to Lat object to associate with node.
`Return`	Reference to Lat object associated with node or `NULL` if no sub lattice is associated with node or in case of error.

```
hapiLatNodeId hapiLatAddSublatNode(hapiLatObject lat, hapiLatObject sub);
```

Add a sub lattice node to a modifiable lattice and return its id.

`lat`	Reference to Lat object.
`sub`	Reference to Lat object to associate with node.
`Return`	Node id of newly created node or a status code less than zero in case of error.

```
int32 hapiLatRouteWordIds(hapiLatObject lat, hapiWordId *words,
                          hapiLatArcId *route);
```

Return the number of word ids of the arcs forming a particular `route` *and optionally load* `words` *with these word ids..*

`lat`	Reference to Lat object.
`words`	Either `NULL` or array to hold word ids. Note that the array must be large enough - the required size can be found with an initial call to this function with `words` equal to `NULL`.
`route`	Either `NULL` indicating that the most likely route through the lattice should be used (only valid for lattices derived from recognition results) or a zero terminated array of lattice arc ids representing a route through the lattice.
`Return`	Number of arcs (and therefore word ids) on route.

```
int32 hapiLatRouteWordNames(hapiLatObject lat, char8 **names,
                            hapiLatArcId *route);
```

Return the number of words forming a particular route through the lattice and optionally load `names` *with their names.*

`lat`	Reference to Lat object.
`names`	Either `NULL` or array to hold the names of the words forming the route. Note that the array must be large enough - the required size can be found with an initial call to this function with `names` equal to `NULL`.
`route`	Either `NULL` indicating that the most likely route through the lattice should be used or a zero terminated array of lattice arc ids representing a route through the lattice.
`Return`	Number of arcs (and therefore word ids) on route.

```
int32 hapiLatRouteOutSyms(hapiLatObject lat, char8 **syms,
                          hapiLatArcId *route);
```

Return the number of words forming a particular `route` *through the lattice and optionally load* `syms` *with their output symbols.*

`lat`	Reference to Lat object.
`syms`	Either `NULL` or array to hold the output symbols of the arcs forming the route. Note that the array must be large enough - the required size can be found with an initial call to this function with `syms` equal to `NULL`.
`route`	Either `NULL` indicating that the most likely route through the lattice should be used or a zero terminated array of lattice arc ids representing a route through the lattice.
`Return`	Number of arcs (and therefore output symbols) on route. Note that `NULL` output symbols are explicitly included in the array.

```
int32 hapiLatWordAlternatives(hapiLatObject lat,int32 st, int32 en,
                              hapiLatArcId *route, int32 n, hapiLatArcId **alt);
```

Return the number of alternative routes (up to `n`*) that could be found for words* `st` *to* `en` *of* `route` *and optionally load* `alt` *with those routes themselves. The alternatives supplied will be ranked in likelihood order and may or may not include the route passed to the function. The alternatives will differ at the word level (only the most likely alternative corresponding to a particular word sequence will be returned) and the remainder of the hypothesis will match the supplied* `route` *at the word level.*

`lat`	Reference to Lat object.
`st`	Index into route at which alternatives should start. Negative numbers may be used to represent offsets from the end of the route (with -1 being the last arc of the route).
`en`	Index into route at which alternatives should end. Negative numbers may be used to represent offsets from the end of the route (with -1 being the last word arc of the route).
`route`	Route through lattice used to define unchanged part of the hypothesis. Either `NULL` indicating that the most likely route through the lattice should be used or a zero terminated array of lattice arc ids representing a route through the lattice.
`n`	Maximum number of alternatives required.
`alt`	Either `NULL` or array of route pointers to hold alternatives found.
`Return`	Number of alternatives found or a status code less than zero in case of error.

Comment	The route is used to provide words that the rest of the utterance must match. For example if route represents a path through the lattice corresponding to the words `THE BLACK CAT SAT ON THE BROWN MAT` calling this function with `st=1 en=2` will find alternatives for `BLACK CAT` as part of the sentence `THE ??? SAT ON THE BROWN MAT`. Similarly a call with `st=-1 en=-2` will find alternatives for `BROWN MAT`. Note that the memory used to hold the alternative routes will be overwritten by the next call to get alternatives or by modifying or making the lattice modifiable.

```
int32 hapiLatPronAlternatives(hapiLatObject lat, int32 st, int32 en,
                              hapiLatArcId *route, int32 n, hapiLatArcId **alt);
```

Return the number of alternative routes (up to `n`*) that could be found for words* `st` *to* `en` *of* `route` *and optionally load* `alt` *with those routes themselves. The alternatives supplied will be ranked in likelihood order and may or may not include the route passed to the function. The remainder of the hypothesis will match the supplied* `route` *at the word and pronunciation level. Unlike* **hapiLatWordAlternatives** *the routes returned by this function will differ at the word or pronunciation level and several alternatives representing the same word sequence may be returned.*

`lat`	Reference to Lat object.
`st`	Index into route at which alternatives should start.
`en`	Index into route at which alternatives should end.
`route`	Route through lattice used to define unchanged part of the hypothesis.
`n`	Maximum number of alternatives required.
`alt`	Either `NULL` or array of route pointers to hold alternatives found.
`Return`	Number of alternatives found or a status code less than zero in case of error.
`Comment`	See **hapiLatWordAlternatives** for more details of how **Alternatives** function works.

```
int32 hapiLatOutSymAlternatives(hapiLatObject lat, int32 st, int32 en,
                                hapiLatArcId *route, int32 n, hapiLatArcId **alt);
```

Return the number of alternative routes (up to `n`*) that could be found for words* `st` *to* `en` *of* `route` *and optionally load* `alt` *with those routes themselves. The alternatives supplied will represent different sequences of* `OutSyms` *and will be ranked in likelihood order and may or may not include the route passed to the function. The remainder of the hypothesis will match the supplied* `route` *at the* `OutSym` *level.*

`lat`	Reference to Lat object.

`st`	Index into route at which alternatives should start.
`en`	Index into route at which alternatives should end.
`route`	Route through lattice used to define unchanged part of the hypothesis.
`n`	Maximum number of alternatives required.
`alt`	Either `NULL` or array of route pointers to hold alternatives found.
`Return`	Number of alternatives found or a status code less than zero in case of error.
`Comment`	See **hapiLatWordAlternatives** for more details of how **Alternatives** functions work.

```
int32 hapiLatTimeAlternatives(hapiLatObject lat, hapiTime st, hapiTime en,
                              int32 n, hapiLatArcId **alt);
```

Return the number of alternative routes (up to `n`*) that could be found between times* `st` *and* `en` *irrespective of the rest of the hypothesis and optionally load* `alt` *with those routes themselves. The configuration parameter* `TIMEWIND` *can be used to specify how closely the start and end times must match.*

`lat`	Reference to Lat object.
`st`	Time at which alternatives should start.
`en`	Time at which alternatives should end.
`n`	Maximum number of alternatives required.
`alt`	Either `NULL` or array of route pointers to hold alternatives found.
`Return`	Number of alternatives found or a status code less than zero in case of error.

```
int32 hapiLatNodeAlternatives(hapiLatObject lat, hapiLatNodeId st,
                              hapiLatNodeId en, int32 n, hapiLatArcId **alt);
```

Return the number of alternative routes (up to `n`*) that could be found between nodes* `st` *and* `en`*, irrespective of the rest of the hypothesis, and optionally load* `alt` *with those routes themselves.*

`lat`	Reference to Lat object.
`st`	Id of node from which alternatives should start.
`en`	Id of node at which alternatives should end.
`n`	Maximum number of alternatives required.
`alt`	Either `NULL` or array of route pointers to hold alternatives found.
`Return`	Number of alternatives found or a status code less than zero in case of error.

Module	Name	Default	Description
HAPI	NETFILE	"net"	Filename of lattice to be loaded.
HNET	ALLOWCXTEXP	T	Allow context expansion to get model names
	ALLOWXWRDEXP	T	Allow context expansion across words
	FACTORLM	F	Factor language model likelihoods throughout words rather than applying all at transition into word. Can increase accuracy when language model likelihoods are relatively high.
	CFWORDBOUNDARY	T	For word internal triphone systems, treat context free phones within words as marking word boundaries.
	FORCECXTEXP	F	Force triphone context expansion to get model names (is overridden by ALLOWCXTEXP)
	FORCELEFTBI	F	Force left biphone context expansion to get model names (i.e. don't try triphone names)
	FORCERIGHTBI	F	Force right biphone context expansion to get model names (ie. don't try triphone names)

A.11 hapiNetObject

The network object provides the recogniser with a specification of what it can recognise. Since different types of recogniser can use different specifications (with the standard decoder using a finite state network in the form of a lattice whilst a dictation decoder will use a statistical language model to guide the search) the network object is decoder specific.

The following configuration variables are used by the standard type of recogniser to specify where to find the lattice file and how the dictionary and the HMMSet should be combined to produce the recognition network.

```
hapiNetObject hapiCreateNetObject(hapiRecType type, hapiHMMSetObject hmms,
                                  hapiDictObject dict);
```

Create a Net object suitable for use with decoder of `type` *using HMMSet* `hmms` *and dictionary* `dict` *and initialise parameters from configuration, return a reference to object created.*

`type`	Type of decoder for which network required.
`hmms`	Reference to HMMSet object defining acoustic models used to build network.
`dict`	Reference to Dict object defining pronunciations used to build network.
`Return`	Reference to Net object or `NULL` in case of error.

```
hapiStatusCode hapiDeleteNetObject(hapiNetObject net);
```

Free all memory associated with Net object `net`.

`net`	Reference to Net object.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiNetSetAttachment(hapiNetObject net, Ptr ptr);
```

Set Net object general purpose attachment to `ptr`.

`net`	Reference to Net object.
`ptr`	New attachment
`Return`	Status code less than zero in case of error.

```
Ptr hapiNetGetAttachment(hapiNetObject net);
```

Return Net object general purpose attachment.

`net`	Reference to Net object.
`Return`	Attachment to Net object or `NULL` in case of error.

```
hapiDictObject hapiNetDictObject(hapiNetObject net);
```

Return Dict object defining vocabulary of Net object `net`.

`net`	Reference to Net object.
`Return`	Reference to dictionary used by Net object or `NULL` in case of error.

```
hapiHMMSetObject hapiNetHMMSetObject(hapiNetObject net);
```

Return HMMSet Object used for acoustic models in Net object `net`.

`net`	Reference to Net object.
`Return`	Reference to HMMSet used by Net object or `NULL` in case of error.

```
char8 *hapiNetOverrideFileName(hapiNetObject net, char8 *str);
```

Return and optionally override location of file specifying network. Depending upon the type of decoder using the network this filename may refer to a standard word lattice or a language model.

`net`	Reference to Net object.
`fn`	Either `NULL` or new name for file.
`Return`	Current value of file name or `NULL` if not set or an error occurs.

```
hapiStatusCode hapiNetInitialise(hapiNetObject net);
```

Initialise network, loading required information from files, expanding words using pronunciations in dictionary in terms of HMMs from HMMSet. For large and complicated networks (especially ones that read large amounts of data such as language models from disk) this is a time consuming operation.

`net`	Reference to Net object.
`Return`	Status code less than zero in case of error.

```
hapiNetObject hapiLoadNetObject(hapiRecType type, hapiHMMSetObject hmms,
                                hapiDictObject dict, char8 *fn);
```

Load network from particular file `fn`.

`type`	Type of decoder for which network required.
`hmms`	Reference to HMMSet object defining acoustic models used to build network.
`dict`	Reference to Dict object defining pronunciations used to build network.
`fn`	Name of file from which to load network.
`Return`	Reference to Net object or `NULL` in case of error.
`Comment`	Function is included as a convenience and is equivalent to creating network, overriding the filename and then initialisation.

```
hapiNetObject hapiBuildNetObject(hapiRecType type, hapiHMMSetObject hmms,
                                 hapiDictObject dict, hapiLatObject lat);
```

Build a network from word level lattice. Function creates and initialises Net object using `lat` *to define syntax. This function is computationally efficient even for large networks. However the first time a network is initialised using a particular HMMSet (by either* **hapiNetInitialise**, **hapiLoadNetObject** *or this function) some time consuming pre-processing is needed, although subsequent networks using the same HMMSet will be built very quickly. Note that this function is only appropriate for decoders that expect explicit syntaxes and cannot be used for example with a dictation type decoder that uses a language model to define the network instead of a word lattice.*

`type`	Type of decoder for which network required.
`hmms`	Reference to HMMSet object defining acoustic models used to build network.
`dict`	Reference to Dict object defining pronunciations used to build network.
`lat`	Reference to Lat object containing word level syntax for which network is required.
`Return`	Reference to Net object or `NULL` in case of error.

A.12 hapiResObject

The results object acts as a holder for results from the recogniser. When the recogniser and application share memory space the results object normally just contains references to recogniser data for intermediate results (although it will contain the structure holding the final results).

Module	Name	Default	Description
HAPI	NBEST	1	Maximum number of alternative sentence hypotheses for output label file.
	MLFFILE		If set save label files to master label file rather than to individual files.

Note that both of these parameters have no effect on standard HAPI calls but alter the way in which HTK format results are stored for later off line system evaluation and checking.

```
hapiResObject hapiCreateResObject(hapiRecObject rec, hapiResInfo what);
```

Create a Res object to hold results from recogniser `rec` *that include the information defined by* `what`*. When the recogniser is running as a remote process greater efficiency may be achieved by ensuring that* `what` *only specifies the particular results required by the application.*

`rec` Reference to Rec object for which results are required.

`what` Flags defining the results information required by the application.

`Return` Reference to Res object or `NULL` if the required results are not supported or in case of error.

`Comment` The specification of the results that are required for this object is formed by a bitwise or of the following flags.

Flag	Description
HAPI_RI_inter	Intermediate results.
HAPI_RI_final	Utterance final results.
HAPI_RI_lat	NBest results as lattice.
HAPI_RI_trace	Trace information.
HAPI_RI_lev_word	Word level results.
HAPI_RI_lev_phone	Phone level results.
HAPI_RI_inc_times	Boundary times.
HAPI_RI_inc_like	Total likelihoods.
HAPI_RI_inc_acoustic	Explicit access to acoustic likelihoods
HAPI_RI_inc_lm	to language model likelihoods.
HAPI_RI_inc_pron	to pronunciation likelihoods.
HAPI_RI_inc_background	to background model likelihoods.
HAPI_RI_inc_confidence	to confidence scores
HAPI_RI_def_htk	Results supported by default recogniser.
HAPI_RI_def_htk_phone	Results supported by default recogniser which include phone level alignment.

```
hapiStatusCode hapiDeleteResObject(hapiResObject res);
```

Free all memory associated with a Res object `res`.

`res`	Reference to Res object.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiResClear(hapiResObject res);
```

Clear information in Res object `res` *ready for reuse.*

`res`	Reference to Res object.
`Return`	Status code less than zero in case of error.
`Comment`	This function should be called each time results are generated before processing continues. Can be called multiple times without error occurring.

```
hapiStatusCode hapiResSetAttachment(hapiResObject res, Ptr ptr);
```

Set Res object general purpose attachment to `ptr`.

`res`	Reference to Res object.
`ptr`	New attachment
`Return`	Status code less than zero in case of error.

```
Ptr hapiResGetAttachment(hapiResObject res);
```

Return Res object general purpose attachment.

`res`	Reference to Res object.
`Return`	Attachment to Res object or `NULL` in case of error.

```
hapiRecObject hapiResRecObject(hapiResObject res);
```

Return Rec object used with Res object `res`.

`res`	Reference to Res object.
`Return`	Reference to recogniser used by Res object or `NULL` in case of error.

```
hapiLatObject hapiResLatGenerate(hapiResObject res, char8 *name);
```

Final results (generated when processing of utterance is complete) can be converted into the form of a lattice by this function. Although a lattice can be generated when the recogniser is not performing N-Best recognition it will only contain the single most likely path.

`res`	Reference to Res object.
`name`	Unique name assigned to lattice. [NOT USED].
`Return`	Reference to Lat object or `NULL` in case of error.

```
hapiStatusCode hapiResSave(hapiResObject res, char8 *fn, char8 *format);
```

Save results to label file `fn` *containing the information specified by* `format`.

`res`	Reference to Res object.
`fn`	Name of output file.
`format`	String specifying information to be included in label file. Many of the options are only relevant if phonetic or state alignment is being performed. The options include; 'W' delete word labels, 'M' delete model labels, 'S' delete scores, 'T' delete times, 'X' strip triphone names to monophones, 'N' normalise scores, 'C' use times which refer to centre of frames rather than boundaries.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiResTraceInfo(hapiResObject res, int32 trFlags,
                                int32 maxLen, char8 *answer);
```

Produce a string (with up to `maxLen` *characters) of trace information including the information defined by* `trFlags`. *This is provided mainly for convenience and debugging purposes.*

`res`	Reference to Res object.
`trFlags`	Flags defining what information is required in the trace information.
`maxLen`	Maximum length of the resulting string.
`answer`	Buffer to hold returned string.
`Return`	Status code less than zero in case of error.
`Comment`	BUG: There is a known bug when the results string gets longer than the internal 16K buffer. Solution - do not request trace information containing very large amounts of information for long sentences - use the standard access functions instead. The `trFlags` are formed from a bitwise combination of the following flags.

Flag	Format	Meaning
HAPI_TF_frame	@%d	Current frame number
HAPI_TF_word	%s	Word
HAPI_TF_pron	%s#%d	Word and pronunciation
HAPI_TF_like	(%.2f)	Word likelihood
HAPI_TF_acoustic	(.. a=%.2f ..)	Word acoustic likelihood
HAPI_TF_lm	(.. l=%.2f ..)	Word lm likelihood
HAPI_TF_dict	(.. p=%.3f ..)	Word pron likelihood
HAPI_TF_time	-%d-	Word boundary times
HAPI_TF_average	[..%.2f..]	Average frame likelihood
HAPI_TF_total	A=%.2f L=%.2f	Total likelihoods
HAPI_TF_active	[..N=%.1f]	Average number of active nodes during recognition

---

```
int32 hapiResAllWordIds(hapiResObject res, hapiWordId *words);
```

Return the number of words in results hypothesis and optionally load `words` *with their ids.*

`res` Reference to Res object.

`words` Either `NULL` or an array of word ids that will be filled with the id of each word in the hypothesis. Note that the array must be large enough - the required size can be found with an initial call to this function with `words` equal to `NULL`.

`Return` Number of words in hypothesis or a status code less than zero in case of error. Also fills in `words` array if not equal to `NULL`.

```
int32 hapiResAllWordNames(hapiResObject res, char8 **names);
```

Return the number of words in results hypothesis and optionally load `names` *with their names.*

`res` Reference to Res object.

`names` Either `NULL` or an array of strings that will be filled with the name of each word in the hypothesis. Note that the array must be large enough - the required size can be found with an initial call to this function with `names` equal to `NULL`.

`Return` Number of words in hypothesis or a status code less than zero in case of error. Also fills in `names` array if not equal to `NULL`.

```
int32 hapiResAllOutSyms(hapiResObject res, char8 **syms);
```

Return the number of words in results hypothesis and optionally load `syms` *with their output symbols.*

`res` Reference to Res object.

`syms`	Either `NULL` or an array of strings that will be filled with the output symbol of each word in the hypothesis. Note that the array must be large enough - the required size can be found with an initial call to this function with `syms` equal to `NULL`.
`Return`	Number of words in hypothesis or a status code less than zero in case of error. Also fills in `syms` array if not equal to `NULL`.
`Comment`	When words have `NULL` output symbols these will be returned and the application should allow for this and skip them when processing results.

```
hapiWordId hapiResWordId(hapiResObject res, hapiResId rid);
```

Return id of one of the words making up the hypothesised results.

`res`	Reference to Res object.
`rid`	Index of word within hypothesis.
`Return`	Word id if `rid` valid, otherwise zero if not found or a status code less than zero in case of error.

```
hapiStatusCode hapiResWordName(hapiResObject res, hapiResId rid, char8 *name);
```

Copy name of one of the words `rid` *making up the hypothesised results into buffer* `name`.

`res`	Reference to Res object.
`rid`	Index of word within hypothesis.
`name`	Buffer of length `STR_LEN` to hold word name.
`Return`	Positive number if name valid, zero if `rid` invalid or status code less than zero in case of error.

```
hapiStatusCode hapiResOutSym(hapiResObject res, hapiResId rid, char8 *sym);
```

Find output symbol of one of the words `rid` *making up the hypothesised results into buffer* `name`.

`res`	Reference to Res object.
`rid`	Index of word within hypothesis.
`sym`	Buffer of length `STR_LEN` to hold output symbol.
`Return`	Positive number if sym valid, zero if output symbol not set or status code less than zero in case of error.

```
hapiPronId hapiResWordPronId(hapiResObject res, hapiResId rid);
```

Return pronunciation number of one of the words `rid` *making up the hypothesised results.*

`res`	Reference to Res object.
`rid`	Index of word within hypothesis.
`Return`	Pronunciation number or status code less than zero in case of error.

```
hapiStatusCode hapiResWordTimes(hapiResObject res, hapiResId rid,
                                hapiTime *st, hapiTime *en);
```

Find start and end times of one of the words `rid` *making up the hypothesised results.*

`res`	Reference to Res object.
`rid`	Index of word within hypothesis.
`st`	Pointer to `hapiTime` to hold returned start time.
`en`	Pointer to `hapiTime` to hold returned end time.
`Return`	Positive number if times valid, zero if times not set or status code less than zero in case of error.

```
float32 hapiResWordScore(hapiResObject res, hapiResId rid,
                         hapiScoreType type);
```

Return and optionally set one score of `type` *for one of the words* `rid` *making up the hypothesised results to* `score`.

`res`	Reference to Res object.
`rid`	Index of word within hypothesis.
`type`	Type of score required.
`Return`	Current score associated with hypothesised word or `HAPI_NAN` in case of error.
`Comment`	Rather than return `HAPI_NAN` when scores are not set default values are returned. For example in the case of likelihoods unknown values default to zero.

```
int32 hapiResAllPhoneIds(hapiResObject res, hapiResId rid, hapiPhoneId *ids);
```

Return the number of phones in one of the words `rid` *making up the hypothesised results and optionally load* `ids` *with their ids.*

`res`	Reference to Res object.
`rid`	Index of word within hypothesis.
`ids`	Either `NULL` or an array of phone ids to hold pronunciation of word. Note that the array must be large enough - the exact size can be found with an initial call to this function with `ids` equal to `NULL` or use an array of size `HAPI_MAX_PRON_LEN`.
`Return`	Number of phones in pronunciation or status code less than zero in case of error.

```
hapiXPhoneId hapiResPhoneId(hapiResObject res, hapiResId rid, int32 n);
```

Return id of one of the phones `n` *of one of the words* `rid` *making up the hypothesised results.*

`res`	Reference to Res object.
`rid`	Index of word within hypothesis.
`n`	Index of phone for which id required.
`Return`	Phone id or a status code less than zero in case of error.

```
hapiStatusCode hapiResPhoneTimes(hapiResObject res, hapiResId rid,
                                 int32 n, hapiTime *st, hapiTime *en);
```

Find start and end times of one of the phones `n` *of one of the words* `rid` *making up the hypothesised results.*

`res`	Reference to Res object.
`rid`	Index of word within hypothesis.
`n`	Index of phone for which times required.
`st`	Pointer to `hapiTime` to hold returned start time.
`en`	Pointer to `hapiTime` to hold returned end time.
`Return`	Positive number if times valid, zero if times not set or status code less than zero in case of error.

```
float32 hapiResPhoneScore(hapiResObject res, hapiResId rid,
                          int32 n, hapiScoreType type);
```

Return and optionally set score of type for one of the phones `n` *in one of the words* `rid` *making up the hypothesised results.*

`res`	Reference to Res object.
`rid`	Index of word within hypothesis.
`n`	Index of phone for which score required.
`type`	Type of score required.
`Return`	Current score associated with phone or `HAPI_NAN` in case of error.
`Comment`	Rather than return `HAPI_NAN` when scores are not set default values are returned. For example in the case of likelihoods unknown values default to zero.

A.13 hapiRecObject

The recogniser is the object which connects everything together and actually processes the observations through the network to produce some results.

In common with the network each recogniser is tailored to its specific decoder and the choice of decoder is governed by the recognition task the application requires.

Module	Name	Default	Description
HAPI	NTOKS	1	Number of tokens propagated into each state. Storing more than one token in each state allow the generation of multiple alternatives.
	LMSCALE	1.0	Language model scale factor. Alters balance between acoustic likelihoods calculated by HMMs and the prior probabilities in the network. Optimal values improve recognition accuracy.
	WORDPEN	0.0	Word insertion penalty. Also alters balance of HMMs and network.
	GENBEAM	250.0	General beam width controlling which hypotheses are pursued. Larger values can prevent certain types of errors at the expense of greater computation.
	WORDBEAM	75.0	Word-end pruning beam width. Controls extension of hypotheses at the end of words. Again larger values can increase accuracy but will require more computation.
	TMBEAM	10.0	Another pruning parameter but only used with "tied-mixture" HMMs. Value has little effect on accuracy and default is normally okay.
	MAXBEAM	2500	Maximum number of active nodes in the recognition network. Controls worst case speed of operation with smaller values resulting in faster recognition (at the expense of more errors).
HRec	FORCEOUT	F	Normally when no token reaches the end of the network by the last frame of the utterance no results are generated (indicated by number of words equal to zero). If this parameter is set to T then results are based on the most likely token anywhere in the network.

```
hapiRecObject hapiCreateRecObject(hapiRecType type, hapiRecMode mode,
                                  hapiHMMSetObject hmms, hapiNetObject net);
```

Create a Rec object operating in specified `mode` *using decoder of* `type`*, HMMSet* `hmms` *and dictionary* `dict`*, initialise parameters from configuration and return a reference to object created.*

`type` Type of decoder to be used by recogniser.

`mode` Operating mode of recogniser.

`hmms` Reference to HMMSet object defining acoustic models used during recognition.

`net` Either `NULL` or reference to Net object to be used as default recognition network.

`Return` Reference to Rec object or `NULL` in case of error.

`Comment` Note that because default network does not need to be specified the HMMset does. Each recognition object can only use a single HMMSet and so all networks it uses must be created from that HMMSet.

Mode	Description
`HAPI_RM_frame_by_frame`	Application makes explicit calls to start recognition, process a certain number of frames of data and finally complete recognition. This gives the application maximum control and is the mode used when reprocessing an utterance.
`HAPI_RM_callback_sync`	Application calls single function which processes complete utterance from coder. Results are made available to the application via an application callback function which is called to process results. Apart from these callbacks the application surrenders complete control over to the recogniser.
`HAPI_RM_callback_async`	*Similar to synchronous callback mode except that the recogniser runs in separate thread so after checking for initial error conditions but before processing the utterance this function returns allowing the application to continue other tasks. Recognition results are passed to the application via the callback function.*

```
hapiStatusCode hapiDeleteRecObject(hapiRecObject rec);
```

Free all memory associated with a Rec object.

`rec` Reference to Rec object.

`Return` Status code less than zero in case of error.

```
hapiStatusCode hapiRecSetAttachment(hapiRecObject rec, Ptr ptr);
```

Set te Rec object general purpose attachment to `ptr`.

`rec` Reference to Rec object.

`ptr` New attachment

`Return` Status code less than zero in case of error.

```
Ptr hapiRecGetAttachment(hapiRecObject rec);
```

Return Rec object general purpose attachment.

`rec`	Reference to Rec object.
`Return`	Attachment to Rec object or `NULL` in case of error.

```
hapiHMMSetObject hapiRecHMMSetObject(hapiRecObject rec);
```

Return HMMSet Object used for acoustic models in recognition.

`rec`	Reference to Rec object.
`Return`	Reference to HMMSet used by Rec object or `NULL` in case of error.

```
hapiNetObject hapiRecNetObject(hapiRecObject rec);
```

Return Net object currently being used. During recognition this will be the active network and at other times the default network associated with the recogniser.

`rec`	Reference to Rec object.
`Return`	Reference to current network associated with Rec object or `NULL` in case of error.

```
hapiStatusCode hapiRecInitialise(hapiRecObject rec);
```

Allocate recogniser structures and initialise ready for processing utterances. At this point the recogniser may claim exclusive use of the HMMSet and default network.

`rec`	Reference to Rec object.
`Return`	Status code less than zero in case of error.

```
hapiObservation hapiRecMakeObservation(hapiRecObject rec)
```

Return an observation buffer for the current rec object. This can be used to get observations directly from the coder for rescoring.

`rec`	Reference to Rec object.
`Return`	Pointer to observation buffer or `NULL` in case of error.

```
int32 hapiRecOverrideNToks(hapiRecObject rec, int32 *n);
```

Return and optionally override the number of tokens to propagate into each state. With only a single token per state only the single most likely hypothesis will be found with more tokens more alternatives can be found.

`rec`	Reference to Rec object.
`n`	Either `NULL` or new number of tokens to propagate.
`Return`	Current number of tokens or `NULL` if not set or an error occurs.

```
int32 hapiRecOverrideNBest(hapiRecObject rec, int32 *n);
```

Return and optionally override the maximum number of alternatives written to the output file.

`rec`	Reference to Rec object.
`n`	Either `NULL` or new number of alternatives.
`Return`	Current number of alternatives or `NULL` if not set or an error occurs.
`Comment`	This has no effect on the number of alternatives available from results lattices.

```
float32 hapiRecOverrideLMScale(hapiRecObject rec, float32 *scale);
```

Return and optionally override the language model scale factor.

`rec`	Reference to Rec object.
`scale`	Either `NULL` or pointer to new scale factor.
`Return`	Current language model scale factor.

```
float32 hapiRecOverrideInsPenalty(hapiRecObject rec, float32 *pen);
```

Return and optionally override the word insertion penalty.

`rec`	Reference to Rec object.
`scale`	Either `NULL` or pointer to new insertion penalty.
`Return`	Current insertion penalty.

```
float32 hapiRecOverrideGenBeam(hapiRecObject rec, float32 *gen);
float32 hapiRecOverrideWordBeam(hapiRecObject rec, float32 *we);
float32 hapiRecOverrideTiedMixBeam(hapiRecObject rec, float32 *tm);
int32 hapiRecOverrideMaxActive(hapiRecObject rec, int32 *max);
```

Return and optionally override pruning parameters. The `gen` *beam is the overall beam width controlling which hypotheses are pursued, the* `we` *beam controls which paths are pursued from the end of words, the* `tm` *beam controls which components of the probability distribution are used to calculate likelihoods for systems using tied mixture HMMs and the* `max` *beam controls the maximum number of models actively considered per frame.*

`rec`	Reference to Rec object.
`gen`	Either `NULL` or pointer to value for the general beam width.
`we`	Either `NULL` or pointer to value for the word end beam width.
`tm`	Either `NULL` or pointer to value for the tied mixture beam width.
`max`	Either `NULL` or pointer to value for the maximum number of models actively considered.
`Return`	Current value of appropriate beam width or an error code. This is `HAPI_NAN` for functions returning `float32` otherwise a negative integer.

Frame-by-frame operation

```
hapiStatusCode hapiRecPrepare(hapiRecObject rec, hapiCoderObject coder,
                              hapiNetObject net);
```

Prepare for processing utterance.

`coder`	Either `NULL` if the application is going to supply observations for rescoring or a reference to the Coder object which will supply the observations for the utterance.
`net`	Either `NULL` if the default network is to be used or a reference to the Net object for which recognition is required.
`Return`	Status code less than zero in case of error.

```
hapiStatusCode hapiRecProcess(hapiRecObject rec,
                              int32 nFr, hapiResObject res);
```

Process `nFr` *observations and transfer results to* `res`.

`rec`	Reference to Rec object.
`nFr`	Maximum number of observations to process.
`res`	Either `NULL` if intermediate results are not required or a reference to the Res object into which results will be copied.
`Return`	The number of frames correctly processed (which can be zero at the end of the utterance) or a status code less than zero in case of error.

```
hapiStatusCode hapiRecProcessObs(hapiRecObject rec, hapiObservation obs,
                                 hapiTime adv, hapiResObject res);
```

Process a single observation `obs` *supplied by the application and transfer results to* `res`.

`rec`	Reference to Rec object.
`obs`	Buffer holding observation to process.
`adv`	Period between this observation and next - must be constant across observations.
`res`	Either `NULL` if intermediate results are not required or a reference to the Res object into which results will be copied.
`Return`	The number of frames correctly processed (i.e. one) or a status code less than zero in case of error.

```
hapiRemainingStatus hapiRecRemaining(hapiRecObject rec);
```

Return an indication of the amount of the utterance remaining.

`rec`	Reference to Rec object.
`Return`	Remaining status or INT_MIN in case of error.
`Comment`	This is one of the function that gives HAPI and opportunity to read, parameterise and buffer audio data. If an application wishes to delay processing (but not capture) of data this function can be called to give HAPI an opportunity to acquire and buffer any data that is available.

```
hapiStatusCode hapiRecComplete(hapiRecObject rec, hapiResObject res);
```

Once utterance is complete (and **hapiRecRemaining** *returns* `HAPI_REM_done`*) finish processing utterance, freeing all recognition structures specific to the utterance and transfer final results to* `res`*.*

`rec`	Reference to Rec object.
`res`	Either `NULL` if final results are not required or a reference to the Res object into which results will be copied.

Callback operation

```
int32 hapiRecRecognise(hapiRecObject rec, hapiNetObject net,
                       hapiCoderObject coder, hapiResObject res,
                       int32 callBackInterval, hapiResObject callBackRes,
                       int32 (*callBack)(hapiResObject res));
```

Recognise an utterance from `coder` *using network* `net` *and transferring final results to* `res` *before calling the applications callback function* `callBack` *passing it a reference to the final results object.*

If intermediate results are required `callBackInterval` *should be set to a positive value indicating the number of frames to process before transferring the results to* `callBackRes` *and calling the same* `callBack` *function passing it a reference to these intermediate results. The application callback function should return an integer status value. If this is negative it indicates that processing of this utterance should be aborted immediately. Otherwise processing should continue, with the number of frames before the next callback equal to the number returned, unless this is zero in which case the callback occurs after the same interval as the current call.*

`rec`	Reference to Rec object.
`net`	Either `NULL` if the default network is to be used or a reference to the Net object for which recognition is required.
`coder`	A reference to the Coder object which will supply the observations for the utterance.

`res` Either `NULL` if final results are not required or a reference to the Res object into which results will be copied before being passed as the argument to the `callBack` function.

`callBackInterval` Number of observations to process before the `callBack` function is called.

`callBackRes` Either `NULL` if intermediate results are not required or a reference to the Res object into which results will be copied before being passed as the argument to the `callBack` function.

`callBack` Application function to call for processing of results.

`Return` In asynchronous mode this function returns a status code (less than zero in case of an error) immediately after performing some initial argument checks. However in synchronous mode recognition is completed before return and the status code reflects errors that may have occurred during recognition.

Appendix B

HAPI from JAVA (JHAPI)

The JAVA HTK Applications Programming Interface (JHAPI) is a package providing the programmer with a set of classes for the development of speech applications in the JAVA programming language. HAPI and HTK are written in the "C" programming language. JHAPI is built upon HAPI via the JAVA native method invocation mechanism.

B.1 JHAPI data types and classes

In JAVA, primitive data types are strictly defined in terms of their size and sign. The primitive data types used in JHAPI and their HAPI equivalents are shown below.

HAPI type	JHAPI type	Description
`short16`	short	16 bit signed integers
`int32`	int	32 bit signed integers
`float32`	float	32 bit IEEE floating point values
`double32`	double	64 bit IEEE floating point values

Table B.1: HAPI numeric data types and their JAVA equivalents

In addition, all integer-derived data types such as `hapiSourceType`, `hapiRecMode`, etc. have been mapped into the JAVA primitive type `int`. All enumeration constants relating to a particular integer-derived data type have been compiled into a JAVA interface.

Figure B.1 demonstrates the translation of HAPI enumeration types into JHAPI interfaces. In HAPI, `hapiSourceType` type and its values are defined as follows

```
enum {
   HAPI_SRC_none,
   HAPI_SRC_file,
   HAPI_SRC_script,
   ...
   HAPI_SRC_last
};
```

```
typedef int32 hapiSourceType;
```

The resulting JAVA class is named `JHAPI.SourceType` and defines the class variables `none`, `file`, `script`, etc.

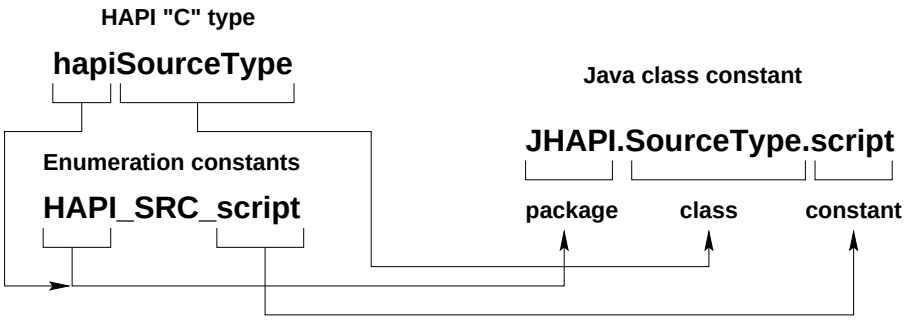

Fig. B.1 Enumeration types

The following table lists the enumerated data types in HAPI and their corresponding JAVA interfaces.

HAPI type	JHAPI interface	Description
`hapiRecMode`	`JHAPI.RecMode`	recogniser modes
`hapiRecType`	`JHAPI.RecType`	recogniser types
`hapiRemainingStatus`	`JHAPI.RemainingStatus`	remaining status codes
`hapiResInfo`	`JHAPI.ResInfo`	results info flags
`hapiResTrace`	`JHAPI.ResTrace`	results trace flags
`hapiScoreType`	`JHAPI.ScoreType`	results score types
`hapiSourceType`	`JHAPI.SourceType`	source types
`hapiStatusCode`	`JHAPI.StatusCode`	status codes
`hapiTransformInfoType`	`JHAPI.TransformInfo`	transform info fields
`hapiTransformType`	`JHAPI.TransformType`	adaptation transform type

Table B.2: HAPI enumerated types and their equivalent JAVA interfaces

Non-primitive data types in JAVA are objects and arrays. Pointers to primitive data types in "C" such as `int`, `float`, `double` have been mapped to the corresponding wrapper classes `Integer`, `Float`, `Double` from the `java.lang` package. The following table shows the mapping of HAPI reference data types into the corresponding JAVA classes.
HAPI declares the following core data types (objects) to represent the different components of a speech recognition application.

- `hapiSourceObject` - source of input speech data
- `hapiCoderObject` - paramaterisation of input data
- `hapiHMMSetObject` - HMM set of acoustic models
- `hapiTransformObject` - speaker adaptation transforms

HAPI type	JAVA type	Description
`(char8 *)`	`String/StringBuffer`	character strings
`(int *)`	`Integer`	integer value holder
`(float *)`	`Float`	float value holder
`(double *)`	`Double`	double value holder
`Ptr`	`Object`	generic object

Table B.3: HAPI pointer types and their equivalent JAVA reference types

- `hapiDictObject` - recognition vocabulary
- `hapiNetObject` - recognition network or language model
- `hapiLatObject` - lattices of alternative hypotheses
- `hapiResObject` - recognition results
- `hapiRecObject` - speech recogniser

HAPI also defines methods for creation manipulation and destruction of objects from each of the above data types. The general rules for composing HAPI function names relating to object `XXX` are as follows

- `hapiCreateXXXObject` - object creation
- `hapiLoadXXXObject` - alternative object creation
- `hapiBuildXXXObject` - alternative object creation
- `hapiDeleteXXXObject` - object destruction
- `hapiXXXYYY` - perform YYY on object

The following example code shows the life-cycle of the acoustic model component in a speech application using HAPI.

```
{
   hapiHMMSetObject hmms;

   hmms = hapiCreateHMMSet();
   hapiHMMSetOverrideList(hmms,"hmm.list");
   hapiHMMSetInitialise(hmms);
   ...
   hapiDeleteHMMSetObject(hmms);
}
```

In JAVA each of the core HAPI objects is mapped into a JHAPI class. All functions relating to a single HAPI object are mapped into instance methods of the corresponding JHAPI class. The following example code shows the acoustic model component life-cycle in a JAVA speech application using JHAPI.

```
{
   JHAPI.HMMSetObject hmms;

   hmms = new JHAPI.HMMSetObject();
```

```
    hmms.overrideList("hmm.list");
    hmms.initialise();
    ...
    hmms.delete();
}
```

Figure B.2 shows the general mechanism for translating HAPI object functions into the corresponding JHAPI instance methods.

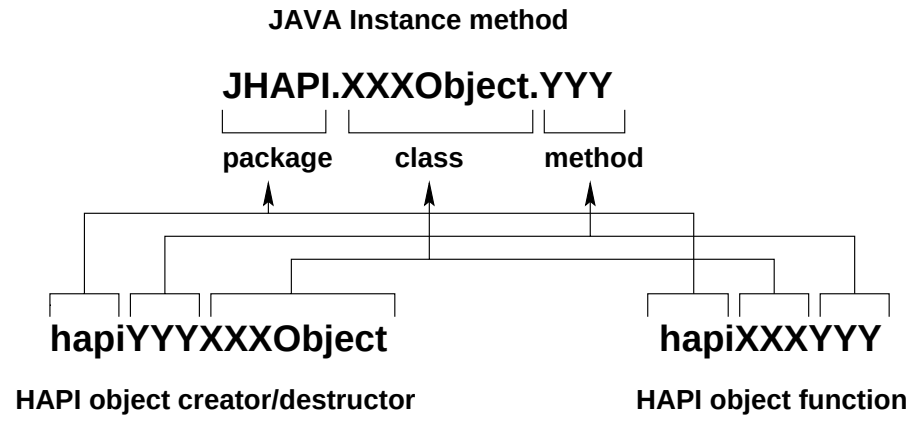

Fig. B.2 Translation of HAPI functions into JHAPI methods

Each object creation function is also mapped into a JAVA constructor for the corresponding class. However, JAVA constructors will return a reference to the newly created instance even when the creation of the underlying HAPI object fails. For convenience reasons, each constructor is also available as a class method with identical method signature which will return `null` if the object creation fails. If class constructors are used, the outcome of the operation can be established using the `JHAPI.hapi.currentStatus` method.

Note that the JAVA language interpreter uses a built-in garbage collector to reclaim storage used for objects to which the user application no longer holds a reference. However, this mechanism does not apply[1] to JHAPI objects, thus the user application should destroy such objects explicitly using the `delete()` instance method of the core JHAPI classes.

B.2 Auxiliary JHAPI classes and interfaces

This section describes the auxiliary classes and interfaces used in the JHAPI package.

JHAPI.Callback

```
public interface Callback
```

This interface defines a single method for passing intermediate and final results between the application and the recogniser.

[1] Instances of core JHAPI classes merely hold a reference to the underlying HAPI objects which reside on the "C" language heap.

Abstract instance methods

```
public abstract int callback(ResObject res)
```

Recogniser callback method for passing intermediate and final results to the application. A negative return value indicates that processing of the utterance should be aborted immediately. A positive value is interpreted as the number of frames to process before the next callback. A value of zero indicates that callback operation should continue at the currently set frame interval.

Notes

The abstract `callback` method is the JAVA equivalent to the function pointer `int32 (*callBack)(hapiResObject res)` passed as the last parameter to the HAPI function `hapiRecRecognise`.

JHAPI.HError

```
public abstract interface HError
```

This interface declares a single method which can be used for graceful shutdown of the user application when a fatal HAPI error occurs. This is normally accomplished by passing an object which implements the `HError` interface to the `JHAPI.hapi.init` method.

Abstract instance methods

```
public abstract void fatalError(int err)
```

Application method for handling fatal HAPI errors. The HTK/HAPI error code is passed via the `err` parameter.

Notes

`fatalError` must not return.

JHAPI.RecMode

```
public interface RecMode
```

Definitions of recogniser operating modes.

Class constants

```
public static int none
public static int frame_by_frame
public static int callback_sync
public static int callback_async
public static int callback_remote
public static int last
```

Notes

For more information on the different recogniser modes see section A.13. Due to the multi-threaded nature of the JAVA language instances of `JHAPI.RecObject` support asynchronous callback mode of operation even when the underlying HAPI does not provide such support.

JHAPI.RecType

```
public interface RecType
```

Recogniser type definitions.

Class constants

```
public static int any
public static int phone
public static int state
public static int sil
public static int bkgrd
public static int htk
public static int htk_phone
public static int htk_state
public static int htk_bkgrd
public static int lvr
public static int lvr_phone
public static int lvr_state
public static int lvr_sil
public static int last
```

Notes

For more information on the different recogniser types see section A.13.

JHAPI.RemainingStatus

```
public interface RemainingStatus
```

Operation remaining status codes.

Class constants

```
public static int error
public static int more
public static int done
public static int never
```

Notes

For more information on remaining status codes and their usage see section A.1.

JHAPI.ResInfo

```
public interface ResInfo
```

Results information flags.

Class constants

```
public static int none
public static int inter
public static int ffinal
public static int lat
public static int trace
public static int lev_word
public static int lev_phone
public static int lev_state
public static int inc_times
public static int inc_like
public static int inc_acoustic
public static int inc_lm
public static int inc_pron
public static int inc_background
public static int inc_confidence
public static int def_htk
public static int def_htk_phone
```

Notes

For more information on the results information flags and their meaning see section A.12. Note that due to the reserved use of the word `final` in JAVA the third flag has been renamed to `ffinal`.

JHAPI.ResTrace

```
public interface ResTrace
```

Results trace information flags.

Class constants

```
public static final int frame
public static final int word
public static final int pron
public static final int like
public static final int acoustic
public static final int lm
public static final int dict
public static final int time
public static final int average
public static final int total
public static final int active
```

Notes

For more information on results trace flags see section A.12.

JHAPI.ScoreType

```
public interface ScoreType
```

Score type definitions for results and lattice objects.

Class constants

```
public static final int lm
public static final int pron
public static final int acoustic
public static final int background
public static final int confidence
public static final int total
```

Notes

For more information on different score types refer to section A.1.

JHAPI.SourceType

```
public abstract class SourceType
```

Source type definitions.

Class constants

```
public static int none
public static int file
public static int script
public static int haudio
public static int haudio_no_det
public static int haudio_with_det
```

Notes

For more information on the different source types see section A.4.

JHAPI.StatusCode

```
public interface StatusCode
```

All JHAPI methods return a status code. In most cases the code is explicitly passed back as the integer value returned by the method. This interface defines constants used as status return values.

Class constants

```
public static int ERR_generic
public static int ERR_status
public static int ERR_fatal
public static int ERR_magic
public static int ERR_object_fatal
public static int ERR_last
```

Notes

For more information on error status codes refer to section A.1. Note that at present JHAPI methods returning `float` values cannot indicate errors. Hence, in such cases error status can be obtained using the `JHAPI.hapi.currentStatus` method.

JHAPI.TransformType

```
public interface TransformType
```

Adaptation transform type definitions.

Class constants

```
public static int fdefault
public static int dynamic
public static int global
public static int use_variance
public static int adapt_sil
public static int fclass
public static int full
```

Notes

For more information on the different transform types see section A.8.

JHAPI.TransformInfo

```
public interface TransformInfo
```

Constants for retrieving and setting adaptation transform info.

Class constants

```
public static int uid
public static int uname
public static int hmmsetid
public static int rcid
public static int chan
public static int desc
public static int size
```

Notes

For more information on transform information fields see section A.8.

B.3 Core JHAPI classes

This section describes the fundamental JHAPI classes. Each HAPI object has been mapped into a corresponding JHAPI class definition. All functions relating to a single HAPI object have been implemented as methods of the corresponding JHAPI class.

B.3.1 JHAPI.hapi

This class provides a collection of static (class) methods for initialisation, status query and final shutdown of the JHAPI package.

Class methods

```
public static void setStringBuffer(StringBuffer s, String value)
```

Set the value of a StringBuffer `s` to `value`. Used internally by JHAPI.

```
public static int currentStatus(StringBuffer buf)
```

Return the status code of the last invoked method. If the value is negative (indicating an error) and the string buffer is not `null`, a short message describing the error will be copied to `buf`.

```
public static int init(String config, HError target)
```

Initialise JHAPI by reading specified configuration file which defaults to "config" if `config=null`. Optionally, the user may supply an object implementing the `HError` interface in which case the `fatalError` method of this object will be called when an error occurs withn JHAPI/HAPI.

```
public static int close()
```

Shutdown JHAPI. No further invocations of class/instance methods are possible.

```
public static String overrideConfStr(String conf, String s)
```

Generic string override function for HAPI configuration parameters.

```
public static Integer overrideConfInt(String conf, Integer i)
```

Generic integer override function for HAPI configuration parameters.

```
public static Float overrideConfFlt(String conf, Float f)
```

Generic float override function for HAPI configuration parameters.

```
public static Boolean overrideConfBool(String conf, Boolean b)
```

Generic boolean override function for HAPI configuration parameters.

Notes

Note that the method `overrideConfBool` uses the JAVA data type `Boolean` rather than `Integer` corresponding to `(int *)` used in the corresponding HAPI function.

B.3.2 JHAPI.SourceObject

```
public class SourceObject
```

An instance of this class defines the source of input data.

Class constants

```
public static final int CAL_speech
public static final int CAL_background
public static final int CAL_snr
public static final int CAL_peak
public static final int CAL_size
```

The above integer constants are analogous `hapiSourceCalibrateParms` and define the size and components of the calibration array passed to the instance methods `calibrateSpeech(float[])` and `calibrateSilence(float[])`.

Constructors

```
public SourceObject(int srcType)
```

This constructor creates a Source object connected to the specified type of input and initialise internal state from configuration parameters. Note that `null` is never returned hence a call to `hapi.currentStatus()` will be necessary to establish if the object creation was successful i.e.

```
SourceObject src = new SourceObject(SourceType.file);
if (hapi.currentStatus < 0)
   System.err.println("Unable to create source");
```

Class methods

```
public static final SourceObject create(int srcType)
```

This method can be considered as an alternative constructor which will return `null` if the object creation fails. For example

```
SourceObject src = SourceObject.create();
if (src==null)
   System.err.println("Unable to create source");
```

Instance methods

```
public final int delete()
```

Free all memory associated with a source object.

```
public final String overrideFilename(String fn)
```

Return and optionally override location of file specifying source.

```
public final int setAttachment(Object obj)
```

Attach an object to the source.

```
public final Object getAttachment()
```

Retrieve the currently attached object.

```
public final int initialise()
```

Connect the source object to the data source and prepare for the first utterance.

```
public final int available()
```

Return availability of source for next utterance.

```
public final int playWave(int n, short data[])
```

Play `n` samples from the data buffer through the output device.

```
public final int calibrateSpeech(float cal[])
```

Obtain information about the average speech level, background noise and clipping.

```
public final int calibrateSilence(float cal[])
```

Obtain information about background noise levels.

```
public final int start()
```

Start the source sampling.

```
public final int stop();
```

Stop the source sampling.

```
public final double samplePeriod(Double sampPeriod)
```

Return and optionally set the sample period of the source.

```
public native final float curVol()
```

Return an estimate in dB of current input level.

Notes

At present, the source object extensions implemented in the "C" version of HAPI are not available in JHAPI}.

B.3.3 JHAPI.CoderObject

```
public class  CoderObject
```

The coder object performs the task of reading data from a `SourceObject`, converting the data into the specified type of observations and providing the application with access to that data.

Class constants

```
public static final int CAL_speech
public static final int CAL_background
public static final int CAL_snr
public static final int CAL_peak
public static final int CAL_size
```

The above integer constants are analogous `hapiCoderCalibrateParms` and define the size and components of the calibration array passed to the instance methods `calibrateSpeech(float[])` and `calibrateSilence(float[])`.

Constructors

```
public CoderObject(SourceObject source)
```

Create a coder object connected to the specified source and initialise parameters from configuration.

Class methods

```
public static CoderObject create(SourceObject source)
```

Alternative constructor - returns null if object creation fails.

Instance methods

```
public int delete()
```

Free all memory associated with the object.

```
public int setAttachment(Object obj)
```

Attach an object to the coder.

```
public Object getAttachment()
```

Retrieve the currently attached object.

```
public int initialise(String chan)
```

Initialise the coder ready for the first utterance.

```
public int prepare()
```

Prepare the coder for operation prior to each utterance.

```
public int complete()
```

Free any storage allocated to the current utterance.

```
public int resetSession()
```

Initiate a new speaker session.

```
public Object makeObservation()
```

Create an observation buffer for the coder.

```
public int accessObs(int i, Object obs)
```

Copy observation number i into the supplied observation buffer.

```
public int accessWave(int st, int en, short data[])
```

Copy samples between observations `st` and `en` into the data buffer.

```
public int saveObs(String fn)
```

Save parameterised data into the specified file.

```
public int saveSource(String fn)
```

Save the complete utterance waveform to the specified file.

```
public int remaining()
```

Return the number of frames currently available from the coder.

```
public int readObs(Object obs)
```

Read the next observation from the coder into the supplied observation holder.

```
public double observationWindowAdvance()
```

Return the observation window advance in ms.

```
public String observationParameterKind()
```

Return a string describing the observation parameter kind.

```
public int getEpts(int tm[]);
```

Return the start (`tm[0]`) and end (`tm[1]`) frame numbers of the portion of the utterance processed by the decoder. If a silence detector is being used then these frame numbers will be

```
tm[0] = SpeechStartTime - SILMARGIN
tm[1] = SpeechEndTime   + SILMARGIN
```

If no silence detector is being used then the start/end of the input is returned. When using a silence detector en is set to the end of the input if silence is not detected. A value of -1 is returned in st or en if the endpoint is not currently known.

B.3.4 JHAPI.HMMSetObject

```
public class  HMMSetObject
```

An instance of this class accommodates an HMM set defining the acoustic model needed to perform recognition.

Constructors

```
public HMMSetObject()
```

Create an HMMSet object and initialise parameters from configuration.

Class methods

```
public static HMMSetObject create()
```

Alternative constructor - returns `null` if object creation fails.

Instance methods

`public int delete()`

Free all memory associated with an instance.

`public int setAttachment(Object obj)`

Attach an object to an HMMSet instance.

`public Object getAttachment()`

Retrieve the currently attached object.

`public int initialise()`

Load HMMSet and prepare for use with a recogniser.

`public String overrideList(String list)`

Override location of HMM set list file.

`public String overrideDir(String dir)`

Override directory pathname of HMM set file.

`public String overrideExt(String ext)`

Override filename extension of HMM set files.

`public String overrideAddMMF(String mmf)`

Add specified master model file (MMF) to current set.

`public int overrideClearMMF()`

Clear the set of default model files specified in the configuration.

Notes

For more information on the override functions and their corresponding configuration parameters refer to section A.7.

B.3.5 JHAPI.TransformObject

`public class TransformObject`

Instances of this class are used to create, calculate and apply speaker adaptation transforms.

Constructors

`public TransformObject(HMMSetObject hmms, int type)`

Create an instance of `TransformObject` associated with `HMMSetObject` `hmms` and initialise parameters from configuration. For information on transform types refer to `JHAPI.TransformType` and section A.8.

`public TransformObject(HMMSetObject hmms, String fn, int type)`

Create a Transform object, initialise parameters from configuration and load transform from TMF file `fn`.

Class methods

```
public static TransformObject create(HMMSetObject hmms, int type)
```

Alternative constructor - returns `null` if object creation fails.

```
public static TransformObject load(HMMSetObject hmms, String fn, int type)
```

Alternative constructor - returns `null` if object creation fails.

Instance methods

```
public int delete()
```

Free all memory associated with a `TransformObject` instance.

```
public int setAttachment(Object obj)
```

Attach an object to an `TransformObject` instance.

```
public Object getAttachment()
```

Retrieve the currently attached object.

```
public String overrideInfo(int infoType, String str);
```

Query or set the transform information strings. For details on the various info fields see section A.8 and `JHAPI.TransformInfo`.

```
public int initialise();
```

Initialise the `TransformObject` ready for adaptation.

```
public int apply(int type);
```

Using the accumulated statistics (from calls to `accumulate()`) a new adaptation transformation is calculated. The adaptation transformation is then applied to the associated `HMMsetObject`.

```
public int revert();
```

Apply reverse transformation to the associated `HMMsetObject` in order to return it to its original state.

```
public int accumulate(LatObject lat, CoderObject coder);
```

Accumulate statistics necessary for the adaptation process using the acoustic data supplied by the `CoderObject` and transcription information from the `LatticeObject`.

```
public int save(String fn, String format);
```

Save transform set to file. For a list of formatting options see section A.8.

B.3.6 JHAPI.DictObject

```
public class  DictObject
```

Instances of this class are used to define the recognition vocabulary.

Class constants

`public static int MAX_PHONE_ID`

Maximum number of phones in dictionary phone set.

`public static int MAX_PRON_LEN`

Maximum number of phones in pronunciation.

Constructors

`public DictObject()`

Create an instance of `DictObject` and initialise parameters from configuration.

`public DictObject(DictObject template)`

Create an instance of `DictObject` and initialise phone set from template dictionary.

Class methods

`public static DictObject create()`

Alternative to constructor `DictObject()` - returns `null` on failure.

`public static DictObject empty(DictObject template)`

Alternative to constructor `DictObject(DictObject)` - returns `null` on failure.

Instance methods

`public int delete()`

Free all memory associated with a `DictObject` instance.

`public int setAttachment(Object obj)`

Attach an object to an `DictObject` instance.

`public Object getAttachment()`

Retrieve the currently attached object.

`public String overrideFileName(String str)`

Return and optionally override location of dictionary file.

`public int initialise()`

Load and initialise a `DictObject`.

`public int save(String fname)`

Save dictionary to a file `fname`.

```
public int allPhoneNames(String names[])
```

Return number of distinct phones in dictionary and optionally their textual names.

```
public int phoneName(int phone_id, StringBuffer name)
```

Copy name of phone with identity `phone_id` into buffer `name`.

```
public int phoneId(String name)
```

Return the id corresponding to a phone `name`.

```
public int allWordNames(String names[])
```

Return the number of words in dictionary and optionally their textual names.

```
public int wordName(int word_id, StringBuffer name)
```

Copy the name of word with identity `word_id` into buffer name.

```
public int wordId(String name)
```

Return the word id corresponding to word name.

```
public int wordAllProns(int word_id, int prons[][])
```

Return number of pronunciations for word `word_id` and optionally fill array `prons` with the actual pronunciations.

```
public int wordPron(int word_id, int pron_id, int pron[])
```

Return the number of phones in pronunciation `pron_id` of word `word_id` and optionally copy actual pronunciation into array `pron`.

```
public String pronOutSym(int word_id, int pron_id, String outsym)
```

Return and optionally set output symbol for pronunciation `pron_id` of word `word_id`.

```
public int modify()
```

Prepare and mark dictionary as modifiable.

```
public int finalise()
```

Update dictionary to reflect changes and mark dictionary as static.

```
public int  addItem(String word, int pron[], String outsym)
```

Add a new word (and optionally a new pronunciation) to the dictionary.

```
public int deleteItem(int word_id, int pron_id)
```

Delete word/pronunciation from the dictionary.

B.3.7 JHAPI.NetObject

```
public class  NetObject
```

Instances of this class define the network used for recognition.

Constructors

```
public NetObject(int rec_type, HMMSetObject hmms, DictObject dict)
```

Create an instance of `NetObject` suitable for use with a recogniser of type `rec_type` using HMM set `hmms` and dictionary `dict`, initialise parameters from configuration.

```
public NetObject(int rec_type, HMMSetObject hmms, DictObject dict, String fn)
```

Equivalent to creating a NetObject, overriding filename and initialising.

```
public NetObject(int rec_type, HMMSetObject hmms, DictObject dict, LatObject lat)
```

Create a recognition network from a word level lattice (LatObject).

Class methods

```
public static NetObject create(int rec_type, HMMSetObject hmms, DictObject dict)
```

Alternative to constructor `NetObject(int,HMMSetObject,DictObject)` - returns `null` on failure.

```
public static NetObject load(int rec_type, HMMSetObject hmms,
                             DictObject dict, String fn)
```

Alternative to constructor `NetObject(int,HMMSetObject,DictObject,String)` - returns `null` on failure.

```
public static NetObject build(int rec_type, HMMSetObject hmms,
                              DictObject dict, LatObject lat)
```

Alternative to constructor `NetObject(int,HMMSetObject,DictObject,LatObject)` - returns `null` on failure.

Instance methods

```
public int delete()
```

Free all memory associated with a `NetObject` instance.

```
public int setAttachment(Object obj)
```

Attach an object to an `NetObject` instance.

```
public Object getAttachment()
```

Retrieve the currently attached object.

```
public DictObject dictObject()
```

Return `DictObject` instance defining the vocabulary of the network.

```
public HMMSetObject hmmSetObject()
```

Return `HMMSetObject` instance used as acoustic models in the network.

```
public String overrideFileName(String str)
```

Return and optionally override location of network file.

```
public int initialise()
```

Initialise netowrk instance by loading required components.

B.3.8 JHAPI.LatObject

```
public class  LatObject
```

Instances of this class are used for defining recognition syntaxes and for holding recognition results. Three distinct constructors are available together with class-method equivalents which return `null` in case of failure.

Instance variables

```
public int nNodes
```

Total number of lattice nodes - set by `size()` method.

```
public int nArcs
```

Number of lattice arcs - set by `size()` method.

```
public int nBestPath
```

Length of best path - `size()` method.

Constructors

```
public LatObject(DictObject dict, String name)
```

Create an instance of `LatObject` with name `name` and initialise parameters from configuration.

```
public LatObject(DictObject dict, String fn, String name)
```

Create an instance of `LatObject` with name `name`, initialise parameters from configuration and read-in lattice from file `fn`.

Class methods

```
public static LatObject create(DictObject dict, String name)
```

Alternative to constructor `LatObject(DictObject,String)` - returns null if object creation fails.

```
public static LatObject load(DictObject dict, String fn, String name)
```

Alternative to constructor `LatObject(DictObject, String, String)` - returns null if object creation fails.

```
public static LatObject build(DictObject dict, String spec, String name)
```

Create an empty LatObject using specification `spec`.

Instance methods

`public int delete()`

Free all memory associated with a `LatObject` instance.

`public int setAttachment(Object obj)`

Attach an object to an `LatObject` instance.

`public Object getAttachment()`

Retrieve the currently attached object.

`public DictObject dictObject()`

Return `DictObject` defining lattice vocabulary.

`public int size()`

Return size of lattice in instance variables `nNodes`, `nArcs`, `nBestPath`.

`public int name(StringBuffer name)`

Return name of lattice in `name`.

`public int save(String fn, String format)`

Write lattice to file `fn` in format specified by `format`.

`public int nodeFollArc(int nodeId, int flinks[])`

Return the number of arcs following a particular node and optionally their ids.

`public int nodePredArc(int nodeId, int flinks[])`

Return the number of arcs preceding a particular node and optionally their ids.

`public int nodeSetAttachment(int nodeId, Object obj)`

Attach an object to the specified lattice node.

`public Object nodeGetAttachment(int nodeId)`

Retrieve attached object.

`public double nodeTime(int nodeId)`

Return time-stamp associated with lattice node.

`public int nodeWordId(int nodeId, Integer wordId)`

Return and optionally set the word id associated with a lattice node.

`public int nodePronId(int nodeId, Integer pronId)`

Return and optionally set the pronunciation number associated with a lattice node.

```
public int nodeXCoord(int nodeId, Integer x)
```

Return and optionally set the pronunciation number associated with a lattice node.

```
public int nodeYCoord(int nodeId, Integer y)
```

Return and optionally set the pronunciation number associated with a lattice node.

```
public int arcStart(int arcId)
```

Return the node id of the start of lattice arc `lid`.

```
public int arcEnd(int arcId)
```

Return the node id of the end of lattice arc id `lid`.

```
public int arcWordName(int arcId, StringBuffer name)
```

Return the word name associated with lattice arc `arcId`.

```
public int arcOutSym(int arcId, StringBuffer sym)
```

Return the output symbol associated with lattice arc `arcId`.

```
public int arcTimes(int arcId, double tm[])
```

Return start (`tm[0]`) and end (`tm[1]`) times associated with lattice arc `arcId`.

```
public int arcWordId(int arcId, Integer wordId)
```

Return and optionally set the id of the word associated with lattice arc `arcId`.

```
public int arcPronId(int arcId, Integer pnum)
```

Return and optionally set the pronunciation id of the word associated with lattice arc `arcId`.

```
public float arcScore(int arcId, int scoreType, Float score)
```

Return and optionally set the specified score of lattice arc `arcId`.

```
public int modify()
```

Prepare and mark lattice as modifiable.

```
public int finalise(int st, int en)
```

Update lattice to reflect any changes made and mark lattice as static.

```
public int deleteNode(int nodeId)
```

Delete a node and all arcs connected to it from a modifiable lattice.

```
public int deleteArc(int arcId)
```

Delete an arc from a modifiable lattice.

```
public int addNode(int wordId, int pronId)
```

Add a node representing a word to a modifiable lattice and return its id.

```
public int addArc(int stNodeId, int enNodeId)
```

Add an arc to a modifiable lattice and return its id.

```
public String nodeSublatId(int nodeId)
```

Check if node represents a sub-lattice and if so return its name.

```
public LatObject nodeSublatObject(int nodeId, LatObject sub)
```

Return and optionally set the sub-lattice associated with lattice node `nodeId`.

```
public int addSublatNode(LatObject sub)
```

Add a sub-lattice node to a modifiable lattice and return its id.

```
public int routeWordIds(int wordId[], int route_arcId[])
```

Return the word ids of the arcs forming the specified route.

```
public int routeWordNames(String names[], int route_arcId[])
```

Return the word names of the arcs forming the specified route.

```
public int routeWordOutSyms(String syms[], int route_arcId[])
```

Return the output symbols of the arcs forming the specified route.

```
public int
     wordAlternatives(int st, int en, int route[], int n, int alt[][])
```

Return the number of alternative routes (up to `n`) that could be found for words `st` to `en` of `route` and optionally load `alt` with those routes themselves. The alternatives will differ at the word level and the remainder of the hypothesis will match the supplied `route` at the word level.

```
public int
     pronAlternatives(int st, int en, int route[], int n, int alt[][])
```

Return the number of alternative routes (up to `n`) that could be found for words `st` to `en` of `route` and optionally load `alt` with those routes themselves. Unlike `hapiLatWordAlternatives`, the routes returned will differ at the word or pronunciation level and several alternatives representing the same word sequence may be returned. The remainder of the hypothesis will match the supplied `route` at the word and pronunciation level.

```
public int
     outSymAlternatives(int st, int en, int route[], int n, int alt[][])
```

Return the number of alternative routes (up to `n`) that could be found for words `st` to `en` of `route` and optionally load `alt` with those routes themselves. The alternatives supplied will represent different sequences of `OutSyms`. The remainder of the hypothesis will match the supplied `route` at the `OutSym` level.

```
public int
     timeAlternatives(double st, double en, int n, int alt[][])
```

Return the number of alternative routes (up to `n`) that could be found between times `st` and `en` irrespective of the rest of the hypothesis and optionally load `alt` with those routes themselves.

```
public int
     nodeAlternatives(int stNodeId, int enNodeId, int n, int alt[][])
```

Return the number of alternative routes (up to `n`) that could be found between nodes `st` and `en`, irrespective of the rest of the hypothesis, and optionally load `alt` with those routes themselves.

B.3.9 JHAPI.ResObject

```
public final class ResObject implements ResInfo, ScoreType
```

Instances of this class are used to obtain and manipulate recognition hypotheses from the recogniser.

Constructors

```
public ResObject(RecObject rec, int what)
```

Create an instance of `ResObject` to hold recognition output which includes information specified by `what`.

Class methods

```
public static ResObject create(RecObject rec, int what)
```

Alternative to constructor `ResObject(RecObject rec, int what)` - returns `null` if object creation fails.

Instance methods

```
public int delete()
```

Free all memory used by a `ResObject` instance

```
public int clear()
```

Clear information held in `ResObject` instance and prepare for re-use.

```
public int setAttachment(Object obj)
```

Attach an object to a `ResObject` instance.

```
public Object getAttachment()
```

Retrieve the currently attached object.

```
public RecObject recObject()
```

Return the `RecObject` instance generating the results

```
public LatObject latGenerate(String name)
```

Convert final results into a lattice of alternative sentence hypotheses.

```
public int Save(String fn, String format)
```

Save results to label file `fn` containing information specified by `format`.

```
public int traceInfo(int trFlags, int maxLen, StringBuffer answer)
```

Generate a string of trace information specified by `trFlags`.

```
public int allWordIds(int words[])
```

Return the number of words in results hypothesis and optionally their ids.

```
public int allWordNames(String names[])
```

Return the number of words in results hypothesis and optionally their names.

```
public int allOutSyms(String syms[])
```

Return the number of words in results hypothesis and optionally their output symbols.

```
public int wordId(int resId)
```

Return id of word `resId` from the recognition hypothesis.

```
public int wordName(int resId, StringBuffer name)
```

Copy name of word `resId` from the recognition hypothesis into buffer `name`.

```
public int outSym(int resId, StringBuffer sym)
```

Obtain output symbol of word `resId` from the recognition hypothesis into buffer `sym`.

```
public int wordPronId(int resId)
```

Obtain pronunciation number of word `resId` from the recognition hypothesis.

```
public int wordTimes(int resId, double tm[])
```

Obtain start (`tm[0]`) and end (`tm[1]`) times of word `rid` from the recognition hypothesis.

```
public float wordScore(int resId, int scoreType)
```

Return the specified score associated with word `resId` from the recognition hypothesis.

```
public int allPhoneIds(int resId, int phoneId[])
```

Return the number of phones in word `resId` and optionally their phone ids.

```
public int phoneId(int resId, int n)
```

Return id of phone `n` in word `resId` from the recognition hypothesis.

```
public int phoneTimes(int resId, int n, double tm[])
```

Obtain start (`tm[0]`) and end (`tm[1]`) times of phone `n` in word `resId` from the recognition hypothesis.

```
public float phoneScore(int resId, int n, int scoreType)
```

Obtain the specified score of phone `n` in word `resId` from the recognition hypothesis.

B.3.10 JHAPI.RecObject

```
public final class RecObject implements Runnable, RecMode, RecType
```

Instances of this class perform recognition of an input utterance using the specified HMM set, dictionary and recognition network. Intermediate and final results can be accessed via an instance of `ResObject`, and the final output can also be transformed into a word lattice.

Constructors

```
public
  RecObject(int rec_type, int rec_mode, HMMSetObject hmms, NetObject net)
```

Create a recogniser object operating in the specified mode using decoder of type `rec_type`, HMM set `hmms`, dictionary `dict` and recognition network `net`. Initialise parameters from configuration.

Class methods

```
public static
  create(int rec_type, int rec_mode, HMMSetObject hmms, NetObject net)
```

Alternative to constructor `RecObject(int,int,HMMSetObject,NetObject)` - return `null` if object creation fails.

Instance methods

```
public int delete()
```

Free all memory associated with a `RecObject` instance.

```
public int setAttachment(Object obj)
```

Attach an object to a `RecObject` instance.

```
public Object getAttachment()
```

Retrieve the currently attached object.

```
public HMMSetObject hmmSetObject()
```

Return the `HMMSet` instance used as acoustic models in recognition.

```
public NetObject netObject()
```

Return the `NetObject` instance used in recognition.

```
public int initialise()
```

Initialise internal state of recogniser and prepare for processing utterances.

```
public Object makeObservation()
```

Return an observation buffer for the current recogniser.

```
public int overrideNToks(Integer ntoks)
```

Return and optionally override the number of tokens that propagate into each HMM state during recognition.

```
public int overrideNBest(Integer nbest)
```

Return and optionally set the maximum number of alternatives written to the output results file.

```
public float overrideLMScale(Float scale)
```

Return and optionally set the language model scale factor.

```
public float OverrideInsPenalty(Float pen)
```

Return and optionally set the word insertion penalty.

```
public float overrideGenBeam(Float beam)
```

Return and optionally override the general pruning beam used in recognition.

```
public float OverrideWordBeam(Float we_beam)
```

Return and optionally override the word-end pruning beam.

```
public float OverrideTiedMixBeam(Float tm_beam)
```

Return and optionally override the tied mixture pruning beam controlling which components of the probability distribution are used to calculate the overall state likelihood.

```
public int OverrideMaxActive(Integer max)
```

Return and optionally override the maximum number of active models considered for each input frame.

```
public int prepare(CoderObject coder, NetObject net)
```

Prepare for processing of utterance.

```
public int process(int nFr, ResObject res)
```

Process `nFr` observations and transfer results to `res`.

```
public int processObs(Object obs, double adv, ResObject res)
```

Process a single observation `obs` supplied by the application and transfer results to `res`.

```
public int remaining()
```

Return remaining status of utterance.

```
public int complete(ResObject res)
```

Free all utterance-specific structures and transfer final results to `res`.

```
public void run()
```

Auxiliary routine for asynchronous processing.

```
public int recognise(NetObject net, CoderObject coder,
     ResObject res, int callbackInterval,
     ResObject callbackRes, Callback cb)
```

Recognise an utterance from `coder` using network `net` and transferring final results to `res` prior to calling the method `callback` of the specified callback object `cb`. If intermediate results are required `callBackInterval` should be set to a positive value indicating the number of frames to process between successive callback method invocations passing `callBackRes`.

B.4 Creating a speech application in JAVA

This section describes the use of JHAPI to develop a simple speech recognition application analogous to the example HAPI application described in chapter 3. The recognition system is designed to recognise a spoken string of digits and produces the recognised telephone number. The program uses the same resources (acoustic models, dictionary, network and configuration file) as the 'C' application.

B.4.1 General information

This tutorial is distributed as part of the JHAPI package. The package is normally installed in a single directory `JHAPI` hierarchy with the actual `libjhapi.so/jhapi.dll` libraries found in the `lib/$CPU` directories for each of the supported machine architectures. All tutorial source files and resources package can be found in the `Tutorial` directory.

In order to compile the program first make sure that the directory containing the JAVA compiler and interpreter is in the search path. Then issue the command

```
% javac Basic.java
```

This will generate the class files

```
Basic.class             FatalHAPIError.class
DialApplication.class   HAPIApplication.class
```

Before executing the program ensure that the appropriate load-library path environment variable (`$LD_LIBRARY_PATH` in most UNIX systems, `$SHLIB_PATH` in `HP-UX` and `$path` in Windows 9x/NT) is setup correctly to include the appropriate `JHAPI/lib/$CPU` directory To execute the program

```
% java Basic.java basic_uk.cfg
```

The JHAPI tutorial is intended to be used in conjucnction with the HAPI tutorial described in chapter 3. The behaviour of the program and its command line arguments are identical to the "C" version. Furthermore, the two programs use the same set of configuration, model, network and dictionary files. For more information on these the reader should refer to section 3.1.

B.4.2 Program Description

The program begins by including all necessary packages. Note the importation of `JHAPI.*` which allows the omition of the `JHAPI` qualifier from all class names.

```
import JHAPI.*;
import java.io.*;
import java.lang.*;
```

B.4.3 Error handling

The handling of HAPI errors is accomplished via the `FatalHAPIError` class implementing the `HError` interface. The class provides an implementation of the `fatalError` method which prints out an error message and terminates the program.

```
class FatalHAPIError implements HError {
   public void fatalError(int error) {
      System.err.println(
        "A fatal HAPI error has caused the application to quit\n");
      hapi.close();
      System.exit(error);
   }
}
```

B.4.4 Basic recognition component

The speech recognition component of the dialer application is built upon a user defined generic class `HAPIApplication` providing the basic recognition capabilities. The class defines several instance variables for holding the components of the recognition system. Recognition is performed in synchronous callback mode and the class provides its own implementation of the `Callback` interface.

```
abstract class HAPIApplication implements Callback {
   // These are set up by the application
   public String config;   // Configuration file name
   public int type;        // Recogniser type
   public int mode;        // Recogniser mode
   public int src;         // Source of input data

   // These HAPI Objects are available to the application
   HMMSetObject hmms;       // acoustic models
```

```
   DictObject dict;         // vocabulary
   NetObject net;           // recognition network
   SourceObject source;     // input data source
   CoderObject coder;       // coder converts waveform to observations
   ResObject intres,finres; // recognition results
   RecObject rec;           // speech recogniser

   // And recognition makes use of the following storage
   boolean abort;          // Set to true for application to abort
   boolean doneRec;        // Set to true to indicate recogniser done
}
```

Initialisation of the speech recogniser components is performed by the method

```
public boolean
   init(String config, int sourceType, int recType, int recMode)
```

First, the JHAPI library is initialised with a call to `hapi.init` passing the name of the configuration file to be read as the first argument and a newly created instance of `FatalHAPIError` for the graceful shutdown of the user program in the case of a fatal error.

```
if (hapi.init(config,new FatalHAPIError())!=0)
   return(false);
```

The next task is to create and initialise the set of objects that provide the basic recognition. The following fragment of code performs initialisation of the components of the recogniser and returns `true` on success.

```
   boolean fail = false;
   fail=fail || ((this.hmms = HMMSetObject.create())==null);
   fail=fail || ((this.dict = DictObject.create())==null);
   fail=fail || ((this.source = SourceObject.create(src))==null);
   fail=fail || ((this.coder = CoderObject.create(this.source))==null);
   fail=fail || ((this.rec = RecObject.create(this.type,
                                              RecMode.callback_sync,
                                              this.hmms,null))==null);
   fail=fail || ((this.intres = ResObject.create(this.rec,resAll))==null);
   fail=fail || ((this.finres = ResObject.create(this.rec,resAll))==null);

   fail=fail || (intres.setAttachment(this)<0);
   fail=fail || (finres.setAttachment(this)<0);

   System.out.println(" Loading HMMs, dictionary and network");
   fail=fail || (this.hmms.initialise()<0);
   fail=fail || (this.dict.initialise()<0);
   fail=fail || ((this.net = NetObject.load(this.type, this.hmms,
                                            this.dict, null))==null);

   // Initialise everything else
   fail=fail || (this.source.initialise()<0);
   fail=fail || (this.coder.initialise(null)<0);
   fail=fail || (this.rec.initialise()<0);
```

Note how the return value of each method is checked to ensure that it has worked as intended. Error status is tracked in the `fail` variable and upon error further

JHAPI calls are prevented through the short-circuit evaluation of the JAVA `boolean` expressions.

When recognition is no longer required, storage for recognition components is freed using the method

```
public boolean delete() {
   boolean fail = false;
   fail=fail || (this.finres.delete()<0);
   fail=fail || (this.intres.delete()<0);
   fail=fail || (this.rec.delete()<0);
   ...
   hapi.close();
   return(!fail);
}
```

The `HAPIApplication` class also provides a method (analogous to `CalibrateAudio` in `basic.c`) for calibration of the audio input device. Calibration of audio input is accomplished via the instance methods `SourceObject.calibrateSpeech` and `SourceObject.calibrateSilence`. The follwoing fragment of code shows the calibration of speech parameters.

```
public boolean calibrateAudio(boolean kpress) {

   boolean fail;
   int needed;
   String buf;
   float levels[] = new float[SourceObject.CAL_size];

   fail = ((needed=this.source.calibrateSpeech(null))<0);
   if (!fail && needed>0) {
      // Now need to set up for interaction
      System.out.println(
         " Recogniser calibration:\n\n" +
         "   If the mean speech level isn't 20-40dB higher than silence\n" +
         "   level please check the audio system and try again\n\n");

      // Now we will measure then display the speech levels
      if (kpress) {
         DataInputStream dis = new DataInputStream(System.in);
         try {
            System.out.println("Press return to measure speech levels");
            buf = dis.readLine();
         } catch (IOException ex) {
            // exception code goes here
         }
         System.out.println(
            " Now say 'She sells sea shells by the sea shore'");
         fail = (this.source.calibrateSpeech(levels)<0);
         System.out.println(" Thank you\n");
         System.out.println(
            " Mean speech level in dBs " + levels[SourceObject.CAL_speech] +
            " (" + levels[SourceObject.CAL_peak]*100.0 + "%)" +
            ", silence " + levels[SourceObject.CAL_background]);
       }
    }
```

```
    ...
    return(!fail);
  }
}
```

The recognition is initiated and performed by the `recogniseWithTrace` method. Note that the final paramater to `RecObject.recognise` is the current instance of `HAPIApplication` which provides its own implementation of the `Callback` inteface.

```
public boolean recogniseWithTrace(int traceInterval) {
   boolean fail = false;

   // Get the input channel ready
   fail = fail || (this.coder.prepare()<0);
   fail = fail || (this.source.start()<0);

   // Do the recognition - with trace and final callbacks
   this.doneRec = false;

   fail=fail ||
      (rec.recognise(this.net, this.coder, this.finres,
                     traceInterval, this.intres, this)<0);

   if (!fail)
      while (!this.doneRec);
   fail = fail || (this.coder.complete()<0);

   return(!fail);
}
```

Finally, the implementation of the `Callback` interface is given below.

```
public int callback(ResObject res) {

   boolean fail = false;

   if (this.finres==res) {
      // We call the application program to let it process them
      fail = fail || this.finalFunc();
      this.doneRec = true;
   } else {
     // Otherwise we just give it the opportunity to print trace info
     StringBuffer str = new StringBuffer(80);
     fail = fail ||
        (res.traceInfo(ResTrace.frame + ResTrace.word, 80, str)<0);
     this.traceFunc(str);
   }

   // Important to tidy up even if something went wrong - note
   // that method called first before testing fail
   fail = (res.clear()<0) || fail;
   if (fail) this.doneRec = true;

   // Return value of 0 means everything is okay and recognition
   // continues with the next callback after the same number of frames.
```

```
   // Return value of -1 indicates that recognition should be
   // aborted immediately
   return(fail?-1:0);
}
```

End of recognition is detected when the parameter `res` passed to `callback` is the final results object `finres`. In this case, the abstract user-defined method `finalFunc` is invoked to perform user-specific handling. Otherwise, intermediate results are printend out. If an error is encountered, `doneRec` is set to `true` and the return value of the method is -1 which informs `RecObject.recognise` to terminate recognition immediately. User-specific handling of intermediate results is provided via the abstract method `traceFunc`.

B.4.5 Dialer recognition component

The dialer recognition application builds upon the basic recognition component and is defined by the class `DialApplication`. The class extends `HAPIApplication` and provides dialer application specific implementations of the `traceFunc` and `finalFunc` methods for the handling of intermediate and final results.

```
class DialApplication extends HAPIApplication {

   String wordString;      // The recognised words
   String dialString;      // The string to dial

   public void traceFunc(StringBuffer str) {
      System.out.println(str.toString());
      System.out.flush();
   }

   public boolean finalFunc() {

      int n = 0;
      String ss;
      boolean ok, fail = false;
      StringBuffer buffer = new StringBuffer(64);

      // Need to know how many words in the result
      fail=fail || ((n=this.finres.allWordNames(null))<0);
      String words[] = new String[n+1];

      wordString="";
      dialString="";
      for(int i=1; i<=n && !fail; i++) {
         // Read name for each word in turn
         ok = (this.finres.wordName(i,buffer)>0);
         ss = buffer.toString();
         //  and append to word string with a space between words and
         //  ignoring special words (starting with !)
         if (ok && !ss.startsWith("!"))
            wordString += ss + " ";
         // Read output symbol for each word in turn
         ok = (this.finres.outSym(i,buffer)>0);
         if (ok)
```

```
            dialString += buffer.toString();
      }

      System.out.println("\nYou said: " + wordString);
      System.out.println("Dialing: " + dialString + "\n");

      return fail;
   }
}
```

B.4.6 Main program

The top level dialer application is contained in the body of the `main` method of the `Basic` class definition. The main control function first parses any command line arguments and creates an instance of the `DialApplication` class. If the chosen source is live audio speech/silence calibration is performed. In the case of file recognition, the program will terminate when all of the files supplied on the command line have been processed. Note that when live audio is used the program will never terminate under normal operating operation.

```
public class Basic {

   // need to load the J-HAPI library
   static {
      System.loadLibrary("jhapi");
   }

   public static void main(String args[]) {
      StringBuffer msg = new StringBuffer(256);

      // Command line can be used to switch to AUDIO or no wait mode
      int srcType = SourceType.haudio_with_det;
      int recType = RecType.htk;
      int recMode = RecMode.callback_sync;
      int narg = args.length;
      boolean kpress = true;

      if (narg<1) {
         System.err.println(
            "Usage: Basic <config> [files to process if not audio]");
         System.exit(1);
      }
      if (narg>1) {
         srcType = SourceType.file;
         kpress = false;
      }

      DialApplication dap = new DialApplication();
      if (!dap.init(args[0], srcType, recType, recMode))
         System.exit(1);

      if (srcType==SourceType.haudio ||
          srcType==SourceType.haudio_with_det)
         dap.calibrateAudio(kpress);
```

```
      int n = 1;
      do {
         if (srcType==SourceType.file) {
            if (n>=narg) break;
            if (dap.source.overrideFilename(args[n])==null) break;
            n++;
         }
         else if (dap.source.available()==0)
            break;

         if (kpress) {
            DataInputStream dis = new DataInputStream(System.in);
            System.out.println("Press return to process utterance");
            try {
               String buf = dis.readLine();
            } catch (IOException ex) {
               // exception code goes here
            }
         }

         if (!dap.recogniseWithTrace(50)) {
            int status = hapi.currentStatus(msg);
            if (status==0)
               System.err.println("User function aborted recognition");
            else
               System.err.println(
                  "Recognition failed, " + status + ", " + msg);
            break;
         }
      } while(!dap.abort);

      if (!dap.delete()) {
         int status = hapi.currentStatus(msg);
         System.err.println("DeleteRecogniser: " + status + ", " + msg);
      }
   }
}
```

Note that since the actual JHAPI method implementations reside in a dynamically loadable library module, the JAVA application program itself is responsible for loading the libray at start up. This operation is performed in class' `static` method.

```
static {
   System.loadLibrary("jhapi");
}
```

The location of the library is usually specified by the shell environment variable `$LD_LIBRARY_PATH` on most Unix systems and `$SHLIB_PATH` under `HP-UX`. In Windows 9x/NT, the user's `$path` must be set to include the location of `jhapi.dll`. Furthermore, the JAVA class search path (`$CLASSPATH`) must be set to include the top level `JHAPI` directory.

Appendix C

Error and Warning Codes

When a problem occurs with the use of HAPI with an application, either error or warning messages are produced. The error or warning is produced by either an underlying HTK function, in which case the error/warning will be indented, and/or by a HAPI function, in which case the error/warning is preceded by the text "`HAPI:`".

All errors and warnings contain an error number which are listed in the next section indexed by the error/warning number. The number is followed by the function name in which the problem occurred and a brief textual explanation.

If an error occurs a number of error messages may be produced on standard error. Many of the functions in the HTK Library do not exit immediately when an error condition occurs, but instead produce a message and return a failure value back to the calling HAPI function. This process may be repeated several times. The HAPI Library also operates in a similar fashion with respect to error handling. When the application that called the HAPI function receives the failure value, the developer must decide how to handle the particular error. In most circumstances it is possible to recover from a HAPI error. The following example demonstrates what might happen if an incorrect file name is supplied as a dictionary file. `hapiDictInitialise` fails and the application requests another dictionary file name.

```
  ERROR [+5010]  InitSource: Cannot open source file uk1.dct
  ERROR [+8010]  ReadDict: Can't open file uk1.dct
HAPI: WARNING [-8888]
  hapiDictInitialise: ReadDict failed
Please enter another dictionary name:
uk.dct
Thank you; uk.dct has been loaded.
```

Error numbers in HTK are allocated on a module by module basis in blocks of 100 as shown by the table shown overleaf. Within each block of 100 numbers the first 20 (0 - 19) and the final 10 (90-99) are reserved for standard types of error which are common to all library modules.

All other codes are module specific.

C.1 Generic Errors

HFB	2300-2400		
HShell	5000-5099	HModel	7000-7099
HMem	5100-5199	HTrain	7100-7199
HMath	5200-5299	HUtil	7200-7299
HSigP	5300-5399		
		HDict	8000-8099
HAudio	6000-6099	HLM	8100-8199
HVQ	6100-6199	HNet	8200-8299
HWave	6200-6299	HRec	8500-8599
HParm	6300-6399	HAdapt	8600-8699

+??00 Initialisation failed
The initialisation procedure for the function produced an error. This could be due to errors in the command line arguments or configuration file.

+??01 Facility not implemented
HTK does not support the operation requested.

+??05 Available memory exhausted
The operation requires more memory than is available.

+??06 Audio not available
The audio device is not available, either there is no driver for the current machine, the library was compiled with `NO_AUDIO` set or another process has exclusive access to the audio device.

+??10 Cannot open file for reading
Specified file could not be opened for reading. The file may not exist or the filter through which it is read may not be set correctly.

+??11 Cannot open file for writing
Specified file could not be opened for writing. The directory may not exist or be writable by the user or the filter through which the file is written may not be set correctly.

+??13 Cannot read from file
Cannot read data from file. The file may have been truncated, incorrectly formatted or the filter process may have died.

+??14 Cannot write to file
Cannot write data to file. The file system is full or the filter process has died.

+??15 Required function parameter not set
You have called a library routine without setting one of the arguments.

+??16 Memory heap of incorrect type
Some library routines require you to pass them a heap of a particular type.

+??19 Command line syntax error
The command line is badly formed, refer to the manual or the command summary printed when the command is executed without arguments.

+???? Sanity check failed
Several functions perform checks that structures are self consistent and that everything is functioning correctly. When these sanity checks fail they indicate the code is not functioning as intended. These errors should not occur and are not correctable by the user.

C.2 Summary of Errors by Module

HFB

+2321 Unknown model
Model in HMM List not found in HMMSet, check that the correct HMM List is being used.

+2322 Invalid output probability
Mixture component probability has not been set. This should not occur in normal use.

+2323 Beta prune failed on taper
Utterance is possibly too short for minimum duration of model sequence. Check transcription.

−2324 No path through utterance
No path was found on the beta training pass, relax the pruning threshold.

+2325 Empty label file
No labels found in label file, check label file.

−2326 No transitions
No transition out of an emitting state, ensure that there is a transition path from beginning to end of model.

+2327 Floor too high
Mix weight floor has been set so high that the sum over all mixture components exceeds unity. Reduce the floor value.

+2328 No mixtures above floor
None of the mixture component weights are greater than the floor value, reduce the floor value.

−2330 Zero occurrence count
Parameter has had no data assigned to it and cannot be updated. Ensure that each parameter can be estimated by using more training data or fewer parameters.

+2350 Data does not match HMM
An aspect of the data does not match the equivalent aspect in the HMMSet. Check the parameter kind of the data.

−2389 ALIEN format set
Input format has been set to `ALIEN`, ensure that this was intended.

HSHELL

+5020 Command line processing error

+5021 Command line argument type error

+5022 Command line argument range error
The command line is badly formed. Ensure that it matches the syntax and values expected by the command.

+5050 Configuration file format error
HSHELL was unable to parse the configuration. Check that it is of the format described in section 8.2.

+5051 Script file format error
Check that the script file is just a list of file names and that if any file names are quoted that the quotes occur in pairs.

+5070 Module version syntax error
A module registered with HShell with an incorrectly formatted version string (which should be of the form `"!HVER!HModule: Vers.str [WHO DD/MM/YY]"`).

+5071 Too many configuration parameters
The size of the buffer used by one of the modules to read its configuration parameters was exceeded. Either reduce the total number of configuration parameters in the file or make more of then specific to their particular module rather than global.

+5072 Configuration parameter of wrong type
The configuration parameter is of the wrong type. Check that its type agrees with that shown throughout part II.

+5073 Configuration parameter out of range
The configuration parameter is out of range.

HMEM

+5170 Heap parameters invalid
You have tried to create a heap with unreasonable parameters. Adjust these so that the growth factor is positive and the initial block size is no larger than the maximum. For `MSTAK` the element size should be 1.

+5171 Heap not found
The specified heap could not be found, ensure that it has not been deleted or memory overwritten.

+5172 Heap does not support operation
The heap is of the wrong type to support the requested operation. In particular it is not possible to `Reset` or `Delete` a `CHEAP`.

+5173 Wrong element size for MHEAP
You have tried to allocate an item of the wrong size from a `MHEAP`. All items on a `MHEAP` must be of the same size.

+5174 Heap not initialised
You have tried to allocate an item on a heap that has not yet been created. Ensure that `CreateHeap` is called to initialise the heap before any items are allocated from it.

+5175 Freeing unseen item
You have tried to free an item from the wrong heap. This can occur if the wrong heap is specified, the item pointer has been corrupted or the item has already been freed implicitly by a `Reset/DeleteHeap` call.

HMATH

+5220 Singular covariance matrix
The covariance matrix was not invertible. This may indicate a lack of training data or linearly dependent parameters.

+5270 Size mismatch
The input parameters were of incompatible sizes.

+5271 Log of negative
Result would be logarithm of a negative number.

HSIGP

+5320 No results for WaveToLPC
Call did not include Vectors for the results.

+5321 Vector size mismatch
Input vectors were of mismatched sizes.

−5322 Clamped samples during zero mean
During a zero mean operation samples were clipped as they were outside the allowable range.

HAUDIO

+6020 Replay buffer not active
Attempt to access a replay buffer when one was not attached.

+6021 Cannot StartAudio without measuring silence
An attempt was made to start audio input through the silence detector without first measuring or supplying the background silence values.

+6070 Audio frame size/rate invalid
The choice of frame period and window duration are invalid. Check both these and the sample rate.

−6071 Setting speech threshold below silence
The thresholds used in the speech detector have been set so that the threshold for detecting speech is set below that of detecting silence.

HVQ

+6150 VQ file format error
The VQ file was incorrectly formatted. Ensure that the file is complete and has not been corrupted.

+6151 VQ file range error
A value from the VQ file was out of range. Ensure that the file is complete and has not been corrupted.

+6170 Magic number mismatch
The VQ magic number (normally based on parameter kind) does not match that expected. Check that the parameter kind used to quantise the data and create the VQ table matches the current parameter kind.

+6171 VQ table already exists
All VQ tables must have distinct names. This error will occur if you try to create or load a VQ table with the same name as one already loaded.

+6172 Invalid covariance kind
Entries in VQ tables must have either `NULLC`, `FULLC` or `INVDIAGC` covariance kind.

+6173 Node not in table
A node was missing from the VQ table. Ensure that the VQ table was properly created or that the file was complete.

+6174 Stream codebook mismatch
The number or size of streams in the VQ table does not match that requested.

HWAVE

+6220 Cannot fseek/ftell
Unless the wave file is read through a pipe fseek and ftell are expected to work correctly so that HWAVE can calculate the file size. If this error occurs when using an input pipe, supply the number of samples in the file using the configuration variable `NSAMPLES`.

+6221 File appears to be a infinite
HWAVE cannot determine the size of the file.

+6230 Config parameter not set
A necessary configuration parameter has not been set. Determine the correct value and place this in the configuration file before re-invoking the application.

+6250 Premature end of header
HWAVE could not read the complete file header.

+6251 Header contains invalid data
HWAVE was unable to successfully parse the header. The header is invalid, of the wrong type or be a variation that HWAVE does not handle.

+6252 Header missing essential data
The header was missing a piece of information necessary for HWAVE to load the file. Check the processing of the input file and re-process if necessary.

+6253 Premature end of data
The file ended before all the data was read correctly. Check that the file is complete, has not been corrupted and where necessary `NSAMPLES` is set correctly.

+6254 Data formated incorrectly
The data could not be decoded properly. Check that the file was complete and processed correctly.

+6270 File format invalid
The file format is not valid for the operation requested.

+6271 Attempt to read outside file
You have tried to read a sample outside of the waveform file.

HPARM

+6320 Configuration mismatch
The data file does not match the configuration. Check the configuration file is correct.

+6321 Invalid parameter kind
Parameter kind is not valid. Check the configuration file.

+6322 Conversion not possible
The specified conversion is not possible. Check the configuration is correct and re-code the data from waveform files if necessary.

+6323 Audio error
An audio error has been detected. Check the HAUDIO configuration and the audio device.

+6324 Buffer not initialised
Ensure that the buffer is used in the correct manner.

+6328 Load/Make HMMSet failed
The model set could not be loaded due to either an error opening the file or the data within being inconsistent.

+6350 CRC error
The CRC does not match that of the data. Check the data file is complete and has not been corrupted.

−6351 Byte swapping not possible
HPARM will attempt to byte swap parameter files but this may not work if the floating point representation of the machine that generated the file is different from that which is reading it.

+6352 File too short to parameterise
The file does not contain enough data to produce a single observation. Check the file is complete and not corrupt. If it is, it should be discarded.

+6370 Unknown parameter kind
The specified parameter kind is not recognised. Refer to section 11.5 for a list of allowable parameter kinds and qualifiers.

+6371 Invalid parameters for coding
The chosen parameters are not valid for coding. Choose different ones.

+6372 Stream widths not valid
Cannot split the data into the specified number of streams. Check that the parameter kind is correct and matches any models used.

+6373 Buffer/observation mismatch
The observation parameter kind should match that of the input buffer. Check that the configuration parameter kind is correct and matches that of any models used.

+6374 Buffer size too small for window
Calculation of delta parameters requires a window larger than the buffer size chosen. Increase the size of the buffer.

+6375 Frame not in buffer
An attempt was made to access a frame that does not appear in the buffer. Make sure that the file actually contains the specified frame.

HMODEL

+7020 Cannot find physical HMM
No physical HMM exists for a particular logical model. Check that the HMMSet was loaded or created correctly.

+7021 INVDIAG internal format
Attempts to load or save models with `INVDIAG` covariance kind will fail as this is a purely internal model format.

±7023 varFloor should be variance floor
HMODEL reserves the macro name `varFloorN` as the variance floor for stream `N`. These should be variance macros (type `v`) of the correct size for the particular stream.

+7024 Variance tending to 0.0
A variance has become too low. Start using a variance floor or increase the amount of training data.

+7030 HMM set incomplete or inconsistent
The HMMSet contained missing or inconsistent data. Check that the file is complete and has not been corrupted.

+7031 HMM parameters inconsistent
Some model parameters were inconsistent. Check that the file is complete and has not been corrupted.

±7032 Option mismatch
All HMMs in a particular set must have consistent options.

+7035 Unknown macro
Macro does not exist. Check that the name is correct and appears in the HMMSet.

+7036 Duplicate macro
Attempted to create a macro with the same name as one already present. Choose a different name.

+7037 Invalid macro
Macro had invalid type.

+7050 Model file format error

+7060 HMM List format error
The file was formated incorrectly. Check the file is complete and has not been corrupted.

+7070 Invalid HMM kind
Invalid HMMSet kind. Check that this is specified correctly.

+7071 Observation not compatible with HMMSet
Attempted to calculate an observation likelihood for an observation not compatible with the HMMSet. Check that the parameter kind is set correctly.

HTRAIN

+7120 Clustering failed
Almost certainly due to a lack of data, reduce the number of clusters requested or increase amount of data.

+7170 Unsupported covariance kind
Covariance kind must be either `FULLC`, `DIAGC` or `INVDIAGC`.

+7171 Item out of range
Attempt made to access data beyond expected range. Check that the item number is correct.

+7172 Tree size must be power of 2
Requested codebook size must be a power of 2 when using tree based clustering.

−7173 Segment empty
Empty data segment in file. Check that file has not become corrupted and that the start and end segment times are correct.

HUTIL

+7220 HMMSet empty
A scan was initiated for a HMMSet with no members.

+7270 Accessing non-existent macro
Attempt to perform operation on non-existent macro.

HDICT

+8050 Dictionary file format error
The dictionary file is not correctly formatted. Section 12.2 describes the dictionary file format.

HLM

+8150 LM syntax error
The language model file was formatted incorrectly. Check the file is complete and has not been corrupted.

±8151 LM range error
The specified value(s) for the language model probability are not valid. Check the input files are correct.

HNET

+8220 No such word
The specified word does not exist or does not have a valid pronunciation.

+8230 Contexts not consistent
HNET can only deal with the standard HTK method for specifying context `left-phone+right` and will only allow context free phones if they are context independent and only form part of the word. This may be indicative of an inconsistency between the symbols in the dictionary and the hmms as defined. There may be a model/phone in the dictionary that has not been defined in the HMM list or may not have a corresponding model. See also section 20.1.3 on context expansion.

+8231 No such model
A particular model could not be found. Make sure that the network is being expanded in the correct fashion and then ensure that your HMM list will cover all required contexts.

+8232 Lattice badly formed
Could not convert lattice to network. The lattice should have a single well defined start and a single well defined end. When cross word expansion is being performed the number of `!NULL` words that can be concatenated in a string is limited.

+8250 Lattice format error
The lattice file is formatted incorrectly. Ensure that the lattice is of the format described in section 13.2.

+8251 Lattice file data error
The value specified in the lattice file is invalid.

+8252 Lattice file with multiple start/end nodes
A lattice should have only one well defined start node and one well defined end node.

HRec

±8520 Invalid HMM
One of the HMMs in the network is invalid. Check that the HMMSet has been correctly initialised.

+8521 Network structure invalid
The network is incorrectly structured. Take care to avoid loops that can be traversed without consuming observations (this may occur if you introduce any 'tee' words in which all the models making up that word contain tee-transitions). Also ensure that the recogniser and the network have been created and initialised correctly.

+8522 Lattice structure invalid
The lattice was incorrectly formed. Ensure that the lattice was created properly.

±8570 Recogniser not initialised correctly
Ensure the recogniser is initialised and used correctly.

+8571 Data does not match HMMs
The observation does not match the HMM structure. Check the parameter kind of the data and ensure that the data is matched to the HMMs.

HRec

+8630 Regression tree error
Cannot find the regression class index or the base from class in the regression tree. Ensure that the HMM set contains a regression tree. If a node in the tree cannot be found, ensure that the correct TMF is being loaded for this HMM set.

+8631 Accumulation error
Cannot find the mixture component in the HMM set to accumulate the adaptation statistics for.

±8640 Transform model file (TMF) format error
The transform model file was incorrectly formed. Ensure that the TMF was created properly, and that none of the headers have been removed, and that the number of blocks (`NBLOCKS` configuration) has been set correctly.

+8650 Data does not match HMMs
Data does not match the HMMs or the HMM type is unsuitable for the purpose of adaptation.

+8660 Matrix handling error
A block diagonal matrix operation has failed. Ensure that the the number of blocks (`NBLOCKS` configuration) has been set correctly.

Index