General Overview

SpellTime allows you to incorporate a spell checker into your text based product. The product comes with a set of java classes included in a signed jar file. SpellTime offers the following features:

- SpellTime incorporates two types of dictionaries. The *main dictionary* contains more than 100,000 English words. This dictionary is compressed to occupy only 350 K bytes on the disk. The efficient spell checking routines decompress the main dictionary data at the run time. The *user dictionary* allows your users to enter new words into the dictionary during the spell checking session.
- You can use dictionary utilities to create foreign language dictionaries using the unicode character set.
- The SpellTime routine can interface with your application at different levels. Your application can call SpellTime to check a single word, or to check all words in a buffer, or to check the entire text file.

SpellTime will show the incorrect word on the screen with a number of alternate words to choose from. The user may also add the incorrect word to the user dictionary, or ignore the incorrect word, or type in a replacement word. SpellTime remembers a user action for an incorrect word, so that any subsequent occurrence of the same word is automatically corrected. SpellTime allows your application to highlight the incorrect word before showing the dialog box for correction.

- SpellTime routines are highly optimized.
- You can create multiple speller sessions for each thread.
- SpellTime includes an interface to TE Edit Control. This interface consists of a module that can be linked with your program. This module can check the entire file or just the highlighted parts. The module supports a line or character highlighted block. This module eliminates any need of programming for incorporating spell checking facility into your application.
- The product comes with a set of Java classes included in a signed jar file.

Technical Overview

The SpellTime product consists of two components: a) a set of dictionaries, and b) a set of Java classes to access the dictionaries.

The main dictionary contains more than 100,000 English words. The data for the main dictionary is contained in a file called *dict35.d*. The index for the data is contained in another file called *dict35.i*. The dictionary data is stored in a compressed format. The data is decompressed selectively during the spell checking session. A subset of the main dictionary data resides in a separate file called *dict35.s*. This file contains all words that consist of one or two letters. This separation of small words from the larger ones allows for enhanced optimization of the word look up algorithm.

The second type of dictionary is the user dictionary. The user can add a new word into this dictionary during the spell checking session.

The user dictionary is shipped empty. The detail data formats of all the dictionaries are described in a later chapter.

SpellTime offers a number of java methods to interface with the dictionary. The main method is called *SpellWord*. This method is frequently called during a spell checking session. The SpellWord method handles screen interactions, and word searches. This method also manages a session database of incorrect words. This database is used to supply automatic correction of repetitively occurring words. The SpellWord method calls the SpellDict method to actually search the dictionary for a word. The SpellDict method employs a highly optimized algorithm to deliver fast word look up. This method can also deliver a set of alternative words for an incorrect word.

When the entire contents of a buffer needs to be checked, you can call the StParseLine method repetitively to extract the individual words. The SpellWord method can then be called to check the individual words.

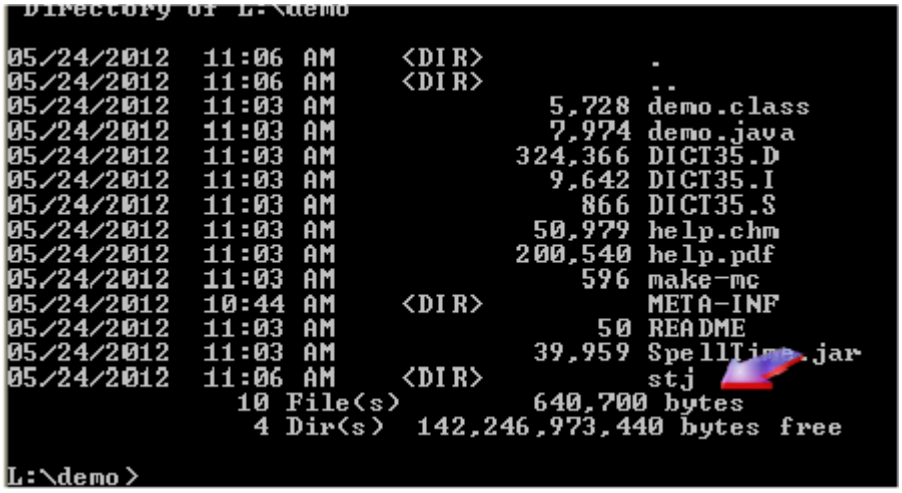
Getting Started

Unzip stje.zip file into a separate folder:

Now use the jar program to unzip the SpellTime.jar in the demo folder:

jar -xfv SpellTime.jar

This would create a new subdirectory called stj where the SpellTime class files are extracted:



'stj' is also the name of the package for this product.

Now use the following command to run the demo:

java -cp L:\demo; demo

In This Chapter

SpellTime Files

The SpellTime distribution contain the following files:

A. File which interface with your application:

SpellTIme.jar	The signed jar file containing the SpellTime methods.
demo.java	Demo java program
demo.class	demo class file

B. Dictionary files:

dict35.d	Main dictionary data file
dict35.i	Main dictionary index file
dict35.s	Small word dictionary
dict35.u	User Dictionary

License Key

Your license key is e-mailed to you after your order is processed. You would set the license key using the StSetLicenseKey static function. This should be preferably done before creating any instance of SpellTime class.

SpellTime.StSetLicenseKey("xxxxx-yyyyy-zzzzz")

Replace the 'xxxxx-yyyyy-zzzzz' by your license key.

SpellTime into Your Application

Follow these easy steps to incorporate SpellTime into your application:

1. Use the jar program to extract the SpellTime files into one of the folders in the classpath:

jar -xfv SpellTime.jar.

This would create a sub-folder called stj, which is also the package name for SpellTime.

2. Copy the dictionary files (dict35.*) into your project folder, or any folder available at run-time.

3. Use the import statement at the top of your program to import the SpellTime package:

import stj.*;

4. Extract the individual words from your application to spell check. You may use the StParseLine method to parse a buffer containing the text. Call the SpellWord method to check the extracted word. The demo.java file illustrates both the StParseLine and SpellWord method calls.

Please refer to the included demo.java program for an example.

Control Methods

SpellTime includes a few helper classes to help interface with the mail SpellTime class.

In This Chapter
SpellDict
SpellWord
SpellString
StGetAlternateWord
StGetReplacement
StAddVowel
StClearHist
StParseLine
StResetUserDict
StSetFlags
StSwitchDict
ToSpellHist
ToUserDict

SpellDict

int SpellDict(*CheckWord,WordLen,flags*)

String CheckWord;	(input) Word to look up. The valid characters in the word are lower case 'a' to 'z' and an apostrophe character.
int WordLen;	(input) The length of the first argument. The length argument is provide to save the CPU cycles in this very frequently called routine.
int flags;	(input) processing flags. (see the description below)

Description: This method performs a dictionary look up for a given word. The method scans all 3 dictionaries. The main dictionary is scanned only if the word is not found in the user dictionary.

This method does not offer any user interface. Nor does it return the alternative words automatically. If the user interface is desired, use the *SpellWord* function. The *SpellWord* method internally calls the *SpellDict* function. Use the *SpellDict* method only when a lower level interface with the dictionaries is required.

The 'flags' argument may specify the following value:

.

ST_GET_ALTERNATES:	Get the alternate words when the input word is not found in any dictionary. This function, however, does *not* return the alternative words automatically. You must call the StLoadResult method after calling this method to retrieve the alternate words. This flag will reduce the performance drastically. If the alternates are desired, process each word in two steps. First, call the SpellDict method without the ST_GET_ALTERNATE flag. If the word is not found, then call this method again, but this time with the ST_GET_ALTERNATE flag set on. This process will significantly improve the performance by eliminating the extra overhead for valid words.

Return: This method can return one of these values:

STD_FOUND: The word look-up successful.

STD_NOT_FOUND: The word not found in the dictionary.

STD_ERROR: A processing error occurred. A MessageBox displays the error message.

If the input word was not found (STD_NOT_FOUND) in the dictionary, and if the input flag has the ST_GET_ALTERNATE flag turned on, the routine will return the alternate words in the *SugWord* global variable array. Similar to the input word, the alternate words are returned in the lower case. The number of alternate words is returned in the global variable *TotalSugWords.* Call the StLoadResult method to retrieve the values of these global variables

Example:

```
strng OneWord="January"

int WordLen;
```

```
            SpellTime st = new SpellTime();
            OneWord=OneWord.ToLower();       /* convert to lower case */
        WordLen=OneWord.Length;
        if (SpellTime.STD_NOT_FOUND==st.SpellDict(OneWord,WordLen,0) {
           /* incorrect word */
           /* find alternate words */
           st.SpellDict(OneWord,WordLen,SpellTime.ST_GET_ALTERNATES);

           TotalAltWords=st.StGetAlternateWordCount();

           for (i=0;i<result.TotalAltWords;i++) {
             MessageBox.Show(st.GetAlternateWord(i),"Alternate
Words",MessageBoxButtons.OK);
           }
        }
```

SpellWord

int SpellWord(*InputWord,flag,ResultCode*)

String InputWord; (input) The input word to spell check. The input word may have any combination of upper/lower case letters 'a' through 'z' and an apostrophe character.

int *flag;* (input) Process control flags.

StInteger ResultCode; (output) Result codes as described below.

Description: The routine provides a high level user interface with the dictionaries. This routine calls the *SpellDict* routine to actually look up the dictionaries. When a buffer containing many words needs to be checked, use the *StParseLine* method to extract the individual words. You can then use the *SpellWord* method to check each extracted word. When the input word is not found in any dictionary, the routine will take an action indicated by the *flag* argument. The *flag* argument may be set to one or more of these constants:

ST_INTERACTIVE Initiate a dialog box to accept user response for an incorrect word.

To indicate more than one flag constants, use the logical *OR* (|) operator.

It is possible to highlight a misspelled word in your application window before showing the SpellTime dialog box for the correction. To accomplish this, first call the SpellWord method without the ST_INTERACTIVE flag. If the method returns with a FALSE value (misspelled word), call the SpellWord method again, but this time with the ST_INTERACTIVE flag turned on (see example).

Return: This method returns a true value if the word is found in the dictionary. The method also returns a true value if the currently *incorrect* word was previously ignored by the user. Therefore, a word that is not found in the dictionary is still considered correct if the user had previously considered the word acceptable.

The method returns a false value when the word is not found in the dictionary or if a processing error occurred. In such a condition, SpellTime returns the result codes (ResultCode.getValue() parameter) as following:

ST_IGNORE: The user ignored this incorrect word. There is no further action needed on the part of the calling routine.

ST_REPLACE: The user wishes to replace the current word with another word. This flag indicates that the replacement word was derived from the history buffer. The replacement word can be retrieve using the StGetReplacment function.

ST_ADD: The user added the current word to the user dictionary. There is no further action needed on the part of the calling routine.

ST_INPUT: The user typed in a replacement word for the current word. The replacement word can be retrieve using the StGetReplacment function.

ST_EXIT: This flag indicates that the user wishes to exit the spell checking session. The calling routine should now take an appropriate action to end the session.

ST_TOO_LONG: This flag indicates that the current word was too long to check.

ST_ERROR: This flag indicates a processing error. The actual error message is displayed using a MessageBox.

The ResultC*ode* variable can have more than one of these flags set. Use the logical *AND* (&) operator to test for a flag, i.e. if (code&SpellTime.ST_EXIT) ...

When the result variable is equal to one of the first four codes, a set of alternative words can be retrieve by using the StGetAlternateWord methods.

Example:

```
string OneWord="January"

int WordLen;

StInteger ResultCode=new StInteger();


SpellTime st=new SpellTime();


if (!st.SpellWord(OneWord,0,out ResultCode) {
  /* incorrect word */
  /* highlight the misspelled word if desired */
  .
  .
  .
  /* call again with the ST_INTERACTIVE flag */
  st.SpellWord(OneWord,SpellTime.ST_INTERACTIVE,ResultCode);


  code = ReturnCode.getValue();
  if ((code&st.ST_EXIT)!=0) return;
  if ((code&st.ST_ERROR)!=0) return;


  string replace=st.StGetReplacement();
  if (replace.Length>0) {  /* replace */
    OneWord=replace;
  }
```

```
}

else {

  MessageBox.Show("Correct Word","",MessageBoxButtons.OK);

}
```

This example calls the SpellWord routine to spell check a word. The first call is made without the ST_INTERACTIVE flag merely to detect a misspelled word. When a misspelled word is detected, the SpellWord method is called again to conduct the correction session. Between these two calls, you can insert the necessary statements to highlight the misspelled word in your application window.

After the second call, the 'exit' and the 'processing error' condition is examined by checking for the ST_EXIT and ST_ERROR flags. Otherwise the *replace* variable is checked for a replacement word. If a replacement word is available (strlen(result.replace)>0), the replacement word is copied to the *OneWord* variable.

You may like to compare this example with the one given with the SpellDict function. This example reveals that the SpellWord method provides a much higher level of user interface compared to the SpellDict function.

SpellString

int SpellString(InString, OutString)

String InString;	(input) The text to be spell-checked.
StString OutBuf;	(output) This StString class variable receives the corrected text.

Description: This method takes a string of text as input and returns the spell-checked text.

Return: This method returns the number of incorrect words found. A negative return value indicates a processing error.

Example:

```
String InString="This is a testss line"

StString OutString = new StString();


if (stj.SpellString(InString, OutString)) {

   String SpellCheckedText = OutString.getValue();

}
```

StGetAlternateWord

String StGetAlternateWord(*WordNumber*)

int WordNumber; (input) Alternate word number to retrieve.

Description: This method can be used to retrieve the alternate words suggested by the SpellWord function.

Return: The method returns the total number of suggested words.

Example:

```
int TotalAlternateWords;

String AlternateWord;

int i;


if (!st.SpellWord(OneWord,0, ResultCode) {

  /* incorrect word */

 TotalAlternateWords=st.StGetAlternateWordCount();

 for (i=0;i<TotalAlternateWords;i++) {

   /* retrieve each alternate word

   AlternateWord=StGetAlternateWord(i);

 }

}
```

StGetReplacement

String StGetReplacement()

Description: This method can be used to retrieve the replacement word suggested by the SpellWord function.

Return: This method returns the replacement word.

Example:

```
String ReplaceWord;

if (!st.SpellWord(OneWord,0,ResultCode) {

  /* incorrect word */

  ReplaceWord=st.StGetReplacement();

}
```

StAddVowel

boolean StAddVowel(chr)

char chr;	A vowel character to be added to the vowel list. This character must be specified in lower case.

Description: This method can be called after initializing a spell-checking session.

Return: This method returns TRUE when successful. Otherwise it returns a FALSE value.

StClearHist

int StClearHist()

Description: Use this method to clear the SpellTime history buffer.

Normally SpellTime will remember the misspelled words that are 'ignored' or 'replaced' by another word. On the subsequent occurrences of these words, SpellTime automatically provides the correction. The misspelled words are stored in the history buffer. This method allows you to clear this buffer any time. For example, you may like to clear the history buffer before starting a new spell checking session.

Return: This method returns a TRUE if successful. Otherwise it returns a FALSE value.

Example:

```
st.StClearHist();

while (!EOF) {

  /* spell check statements here */

}
```

StParseLine

int StParseLine(*buffer,word,ref WordIndex, ref CurIndex,LineLen*)

String buffer;	(input) Pointer to the string containing the words to extract.
StString word;	(output) The StString class variable where the extracted word is to be copied.
StInteger WordIndex;	(output) Starting position of the extracted word with respect to the beginning of the buffer.
StInteger CurIndex;	(input/output) The method begins examining the buffer location as given by this argument. When a word is extracted, this location is updated to contain the pointer after the end of the word. Therefore, the next call to the StParseLine routine will automatically begin the search where the previous call ended.
int LineLen;	(input) The length of the buffer to examine. The length is counted from the beginning of the buffer. If the calling routine inserts or deletes a word in the buffer, it should update this variable appropriately to reflect the updated length of the buffer.

Description: Use this routine to parse a buffer containing words to be spell checked. Each call returns a word. The extracted word contains a combination of upper/lower case characters 'a' to 'z' and the apostrophe character. This word is acceptable to the *SpellWord* function. Therefore, the main usage of this method is to create words for the *SpellWord* function. Your application can call this method repetitively until all words from a buffer are extracted.

Return: The method returns the length of the extracted word. A zero length indicates the end of the buffer.

Example:

```
String line="It pays to increase your word power

                  (Dr. Funk).";

StString CurWord=new StString();

StInteger WordIndex = new StInteger();

StInteger CurIndex = new StInteger();

StInteger ResultCode = new StInteger();


int LineLen;

LineLen=line.length();


SpellTime st = new SpellTime();
```

```
while ((WordLen=st.StParseLine(line, CurWord,
                 WordIndex, LineIndex, LineLen))>0) {

String word=CurWord.getValue();
if(!st.SpellWord(word ,ST_INTERACTIVE, ResultCode){
    // Incorrect Word

    int code = ResultCode.getValue();

    if ((code&ST_EXIT)>0) return;
    if ((code&ST_ERROR)>0) return;

    String replace=st.GetReplacement();
    if (replace.length()>0) {
      /* insert the new word in the line buffer */
      /* update the LineLen variable */
    }
  }
  else {
    // Correct Word
  }
}
```

StResetUserDict

int StResetUserDict(*new,old*)

String *new*; (input) Pathname of the new user dictionary.

StString *old*; (output) Pathname of the previous user dictionary.

Description: This routine closes the current user dictionary and opens the specified new user dictionary. If the new dictionary parameter is "", the current dictionary remains open. However, its contents are written out to the disk file. The second argument receives the pathname of the previous user dictionary. The second argument should point to a string large enough to hold a complete pathname.

This method serves two purposes. First, it is used to update the user dictionary file with the contents of the user dictionary buffer. It is accomplished by calling this method with empty string arguments at the end of the spell checking session. Second, this method can be used to activate a new user dictionary before initiating a spell checking session. By default, the 'dict35.u' file is used as the user dictionary. An application that sets a new user dictionary should activate the previous dictionary at the end of the session. This also causes the new user dictionary file to be updated.

Return: This method returns a TRUE value to indicate the success, and a FALSE value to indicate a processing error. The pathname of the previous dictionary is copied to the string pointed by the second argument.

Example:

1. update the existing user dictionary file

```
st.StResetUserDict("",null);
```

2. open a new user dictionary

```
StString old=new StString();

st.StResetUserDict("mydict",old);
/* spell checking statements here */
.
.
String OldDict = old.getValue();
st.StResetUserDict(OldDict,"");
```

StSetFlags

int StSetFlags(set, flag)

bool set;	TRUE to set the flag, or FALSE to reset it.
int flag;	The flag to set or reset.
	The following flags are available currently:

STFLAG_USE_APOSTROPHE:	SpellTime normally treats the apostrophe character as the possessive case modifier. Instead, you can set this flag in the beginning of your program to treat the apostrophe character as a regular character.
STFLAG_SPANISH_DLG:	Show the word-selection dialog box in Spanish.
STFLAG_NO_NUM_IN_WORD	This flag instructs the StParseLine method to filter words containing numbers.
STFLAG_DUTCH_DLG	Show the word-selection dialog box in Dutch.
STFLAG_GERMAN_DLG	Show the word-selection dialog box in German.
STFLAG_FRENCH_DLG	Show the word-selection dialog box in French.
STFLAG_ALL_CAPS_TO_LOWER	Convert a capitalized word to lower case for spell checking. This feature is useful when using a case-sensitive dictionary.

Return: This method returns the new value of the flag bits.

StSwitchDict

bool StSwitchDict(DictName)

String DictName; The new dictionary file name including the '.d' extension, and any path specification.

```
Example:  mydict35.d

          c:\uk\dict35.d
```

Return: This method returns true when successful.

ToSpellHist

boolean ToSpellHist(*CurWord,flag,ReplaceWord)*

String CurWord;	(input) A word that needs to be inserted into the history buffer.
char *flag;*	(input) The flag that indicates whether the word is (I) ignored by the user or is (R) replaced by another word.
String ReplaceWord	(input) Pointer to the replacement word when the flag is equal to 'R'.

Description: This routine is used to insert a word into the history buffer. All subsequent occurrences of the given word is automatically ignored or replaced by the library. If the word is being replaced by another word, the replacement word is provided by the last argument.

Both the input word and the replacement word must be provided in lower case.

Return Value: This method returns true when successful.

ToUserDict

int ToUserDict(*CurWord)*

String CurWord; (input) A word that needs to be added to the user dictionary buffer.

Description: This routine is used to add a word to the user dictionary. The input word must be provided in lower case.

Also note that your application needs to call the StResetUserDict method at the end of your program to actually write the updated user dictionary to the disk file.

Return Value: This method returns true when successful.

See Also

StResetUserDict

Utility Classes

In This Chapter
StInteger
StString

StInteger

This class is used with a number of SpellTime methods to pass the 'reference' type of integer parameters. For example, the third parameter for the SpellWord method is of the type StInteger. It allows SpellWord to return the 'return code' by updating the StInteger value. The calling application can then retrieve the the return code by using the getValue() method of the StInteger class object.

```
StInteger ReturnCode = new StInteger();


boolean IsCorrect = stj.SpellWord("mountain",0, ReturnCode);


if (!IsCorrect) {   // if misspelled

   int code=ReturnCode.getValue();

}
```

StString

This class is used with a number of SpellTime methods to pass the 'reference' type of String parameters. For example, the second parameter for the StParseLine method is of the type StString. It allows StParseLine to return the 'current word' by updating the StString value. The calling application can then retrieve the the current word by using the getValue() method of the StString class object.

```
StInteger WordIndex = new StInteger();

StInteger CurIndex = new StInteger();

StString  CurWord = new StString();



boolean WordExtracted= StParseLine(TextLine , CurWord,
WordIndex,


      CurIndex,LineLen)



if (WordExtracted) {   // a word extracted from the text line

   String word=CurWord.getValue();

}
```

Memory Considerations

Some SpellTime data objects have a fixed memory requirement, where as other objects have flexible memory requirements. In this section we will discuss each data component. Where possible, we will also indicate ways of reducing memory overhead by curtailing certain functionalities.

Dictionary Index: This component consists of data pointers (4 bytes), data size (4 bytes), and data location (1 byte). There are 784 (ST_SIZE*ST_SIZE) dictionary indices. Therefore the total memory requirement is approximately 7 K bytes ((4+4+1)*784). This memory is allocated in the FAR location.

Small Word Dictionary: At present, SpellTime requires approximately 450 bytes to read the small word dictionary (dict35.s) into memory.

User Dictionary: The memory requirement for this component is equal to the size of the user dictionary plus an allowance (ST_BUF_SIZE) for new words. At present the ST_BUF_SIZE is set to 2 K bytes.

History Buffer: The initial size of the history buffer is equal to 2*ST_BUF_SIZE. The history buffer can expand during the spell checking session as needed. You can reduce the initial memory requirement of this component by assigning a smaller value to the ST_BUF_SIZE global constant.

Main Dictionary Data: The cumulative memory requirement for all objects in the main dictionary data file is approximately 350 K bytes. However, the memory requirement for this component is flexible (minimum memory requirement = 0 K bytes). The main dictionary data is not loaded into the memory during the initialization. The data is read into the discardable memory buffers as needed during the spell checking session.

Interface With TE Edit Control

TE Edit Control can interface with SpellTime without any coding on your part. Simply move the SpellTime classes to the directory where the Tej files are located. Alternatively, you can copy the SpellTime classes to a separate folder, and include this folder in the classpath switch for Java command to launch your application.

Also move all the dict35.* files to the same directory.

After the editor window is created, set the SpellTime key as following.

Tern.TerSetStLicenseKey(LicenseKey)

Your license key for SpellTime is e-mailed to you after your order for SpellTime is processed. Please note that your license key for TE Edit Control is not valid when calling the TerSetStLicenseKey method.

To invoke spell checking from within your program, simply add this statement:

tern.TerCommand(ID_SPELL)

or, set the command property of the control:

tern.command=ID_SPELL

Dictionary Update Utilities

The package comes with 3 DOS based utilities to add new words to the main dictionary. Follow these steps to add new words to the main dictionary:

1) Run the **decomp35.EXE** utility to decompress the main dictionary into a number of text files. The text files are named as D35_A through D35_Z, D35_n, D35_SML. dict35.map. The D35_A through D35_Z files contain the words starting with an English alphabet. If the dictionary supports additional characters, those words are written into the files with name (D35_xxxx) built by concatenating the unicodeI value of the character in the hex format to the prefix 'D35_'. For Example, the D35_0027 file contains words that start with an apostrophe character, i.e. 'twill. The D35_SML file is a copy of dict35.s, the small word dictionary. The words in the text files have compression codes in the form of a period (.) character.

Note: All text files (d35_*, dict35.map, etc) used by dictionary update utilities are in Unicode text format. A unicode text file includes a two signature bytes (0xFF and 0xFE) in the beginning of the files. Thereafter, each 2 byte sequence is treated as one unicode character. You must use a unicode aware editor (such as Notepad) to view or edit these files.

The dict35.map contains the character map supported by the dictionary. The file has one line for each supported character. Each line has two letters separated by a comma. The first letter denotes the uppercase form of the letter, and the second letter denotes the lowercase form of the letter. Example: A,a.

Syntax:

decomp35 [/S]

The optional /S switch suppresses the program messages.

2) Run the **merge35.EXE** utility to merge a list of words contained in a word file to the D35_* files. *The word file must be in the Unicode text format (see note above).* The word file should consist of words delimited by a space, comma, or carriage return/new line combination. The individual words can contain the characters supported indicated in the dict35.map file. For example, the standard dictionary supports these characters:

Alphabets 'a' through 'z'

Alphabets 'A' through 'Z'

and an apostrophe character.

The merge utility converts the uppercase characters to the lowercase. A word must not be greater than 40 characters. The apostrophe characters can be used only as an abbreviator, and NOT as a possessive specifier. A merge file may not be larger than 32000 bytes. You can break a large merge file into smaller files and run the merge35 program multiple times.

Examples of valid words in a word file:

cat, dog

cats,dogs

Apple,

he'll

Examples of **invalid** words:

21ST /* numerics not allowed */

cat's /* possessive case not allowed */

apple-growers /* hyphenation not allowed */

The dict35.u file contains the words in the valid format. Thus, these files can be directly merged into the dictionary text files.

Syntax:

MERGE35 MergeFile [/S]

MergeFile: Name of the word file.

The optional /S switch suppresses the program messages.

Example:

MERGE dict35.u

MERGE dict35.app /S

MERGE YourMergeFile

As an alternative, you can also use a unicode aware text editor such as Notepad to add or delete words form the D35_A through D35_Z, and D35_xxxx files. This manual method requires utmost care so as not to disturb the sorting order within the file. The sorting order is governed by the position of the characters in the dict35.map file. In the standard dictionary, the text files assume that the apostrophe character has a higher collating sequence than the letter 'z'. The merge35 utility also inserts the compression codes appropriately into the new words. If you are manually editing the text files, you will need to provide these compression codes to match the neighboring words. Because of these considerations, we encourage the use of the merge35 program instead.

3) Run the **comp35.EXE** utility to compress the dictionary text files (D35_*) to form the dict35.d and dict35.s files.

Syntax:

comp35 [/S]

The optional /S switch suppresses the program messages.

Although these 3 steps are required, you do not necessarily have to run the first step every time. Normally you can delete the D35_* files after the last step. But if you have enough disk space, you may like to retain them. If the text files are retained, you can skip the first step the next time.

Building a Foreign Language Dictionary

The dictionary update utility described in the previous chapter can be used to build a foreign language dictionary. The dictionary update utilities support foreign languages which are based on a single byte character set. Follow these steps to build a foreign language dictionary.

1. Build the character map file (dict35.map). The character map file is a unicode text file which contains the characters supported by the language. To retrieve the default map file, run the decomp35.exe program. The default map file contains these lines:

A,a

B,b

C,c

D,d

E,e

F,f

G,g

H,h

I,i

J,j

K,k

L,l

M,m

N,n

O,o

P,p

Q,q

R,r

S,s

T,t

U,u

V,v

W,w

X,x

Y,y

Z,z

','

If your language uses the English alphabets, then you don't need to modify this file. If you language uses the English alphabets and also some additional characters, add the additional characters after the last line in the file. *You must use a unicode aware editor such as Notepad to edit the map file.*

If your language uses non-English alphabets, delete the existing lines from this file and add new lines, one for each character supported by the language. Each line should contain 2 characters separated by a comma. The first character should be the uppercase form for the letter. The second character should be the lowercase variation for the letter. For some characters the uppercase and the lowercase form may be identical.

The standard English dictionary map has 27 characters ('a' to 'z', and an apostrophe character). SpellTime supports up to 62 characters for a language. SpellTime works more efficiently with dictionaries that have a smaller number of characters in the character set.

2. Once the map file is built, you are ready to merge your list of words by using the merge35.exe program. This program merges a file containing the list of words into the decoded dictionary file set. Please refer to the previous chapter for the description of the merge program.

3. Once all the merge files are merged into the decoded dictionary file set, you can use the comp35.exe program to build the binary dictionary file.

Dictionary Data Format

This section describes the data format of the various dictionary components.

An application developer does not need to understand the dictionary format of the *main dictionary*. Nonetheless, the data format is described here for those developers who have time and inclination to dwell into the complexity of this subject matter.

Main Dictionary

The main dictionary consist of three files: dict35.d, dict35.i and dict35.S. The dict35.s file contains small words that consist of one or two letters. The words in this file are stored in lowercase and are delimited by a comma.

The dict35.d file contains the words that have more than 2 characters. This file is divided into various word buckets. Each word bucket contains words that have a common first two letters. For example, words that start with 'ab' are stored in one bucket, and the words that start with 'ac' are stored in another bucket. The dict35.I file contains the pointer to each word bucket. The dict35.i file also contains the size of each word bucket.

The words in a word bucket are arranged in the alphabetic order. Further, the words are stored in a compressed format. To understand the compression scheme employed in the dictionary, consider these 3 words:

```
cerebra

cerebral

cerebrally
```

These words clearly have common string components. The dictionary will store these words in a series as following:

```
cere bra 1 lly
```

Obviously, this series must be stored in such a fashion so that three individual words can be extracted during the spell checking session. The series has four components. The first word can be reconstructed by combining the first and the second component. The second string can be reconstructed by combining the first, second and the third component. The third word can be reconstructed by combining the first, second and the fourth component. Therefore, it is necessary to delimit these components in the dictionary.

The process of delimiting the components is facilitated by a concept of levels. A level determines the hierarchy of a component in its parent words. In the example above, the individual components will be assigned the following levels:

```
cere 0

bra  1

1    2

lly  2
```

Normally the alphabetic characters are mapped to ASCII 1 to 26. The apostrophe character is mapped to ASCII 27. Therefore, all characters in level zero ("cere") will be mapped to the values between 1 and 26. The first character of the second level ("bra") will be raised to

level one by adding ST_SIZE to its level zero value. The second character of the second level will be at level zero. The last character of the second level will be raised to the highest level (ST_MAX_LEVELS). The ST_MAX_LEVELS indicates an end of the word. With the information provided in this paragraph, you can reconstruct the first word.

To reconstruct the second word, look at the third component ("I"). This component has only one character. This character is raised to the second level. Because there are no additional characters, an additional character (ST_END_OF_int) is appended which marks the end of this word.

To reconstruct the third word, notice that the fourth component is also raised to the second level. By applying the logic of the preceding two paragraphs, the third word can be constructed by combining the first, second, and the fourth component.

The end-of-series is indicated by appending the ST_NEW_STREAM character.

User Dictionary

The user dictionary is automatically updated by the SpellTime routines. All words are stored in lowercase with the first character capitalized. Each word is delimited by the comma character.