General Overview

SpellTime allows you to incorporate a spell checker into your text based product. The product comes with a Windows DLL and its complete source code. SpellTime offers the following features:

- SpellTime incorporates three 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 *application dictionary* allows an OEM to enter their industry specific words. This feature offers you to customize the dictionary before your product is shipped to the user. The *user dictionary* allows your users to enter new words into the dictionary during the spell checking session.
- 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. A performance benchmark on a 386, 33MHz computer reveals that SpellTime can process 500 to 700 words per second! A similar test on a 486, 33MHz computer indicates a speed of 1000 to 1300 words per second. These benchmarks employ a text file containing normal English words.
- SpellTime employs a flexible memory management technique. The main dictionary components are loaded as needed into the discardable memory blocks. This technique allows the Windows to free up memory blocks in tight memory situations. SpellTime routines needs approximately 20 K bytes of memory for the initial overhead. Additional 20 to 50 K bytes of memory may be needed during the spell checking session for the temporary data structures.
- SpellTime has two sets API functions. The second set of API functions have a suffix of '2' (i.e. StParseLine2()) and can be used to provide thread-safe spell-checking in multi-threaded applications. You can create one or more 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.
- SpellTime routines follow structured programming guidelines with a very limited use of global variables. The routines use meaningful variable names. This manual describes the functionality of the routines in detail.
- The product comes with a Windows DLL and its complete source code that is compatible with Microsoft and Borland 'C' compilers. SpellTime can be compiled in all memory models.

Technical Overview

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

The main dictionary contains more than 100,000 English words. The data for the main dictionary is contained in a DOS file called *dict25.d*. The index for the data is contained in another DOS file called *dict25.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 *dict25.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 called the application dictionary. This dictionary is contained in a DOS file called *dict25.app*. This dictionary allows an OEM to insert their industry specific words in the dictionary. You can use a text editor of your choice to insert a word in the application dictionary. To increase the speed of look up, each word must be delimited by a comma. The first letter of a word must be capitalized and all other words must be in lower case.

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

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

SpellTime offers a number of DLL functions to interface with the dictionary. The main function is called *SpellWord*. This function is frequently called during a spell checking session. The SpellWord function handles screen interactions, and word searches. This function also manages a session database of incorrect words. This database is used to supply automatic correction of repetitively occurring words. The SpellWord function calls the SpellDict function to actually search the dictionary for a word. The SpellDict function employs a highly optimized algorithm to deliver fast word look up. This function 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 function repetitively to extract the individual words. The SpellWord function can then be called to check the individual words.

Getting Started

In This Chapter

SpellTime Files

The SpellTime diskettes contain the following files:

A. File which interface with your application:

SPELL.DLL	The DLL containing the SpellTime functions (SPELL32.DLL for WIN32).
SPELL.LIB	(SPELL32.LIB for WIN32) The import library that must be linked with your program to interface with the SpellTime DLL.
SPELL.H	Header files to be included into your 'C' application.

B. Dictionary files:

DICT25.D	Main dictionary data file
DICT25.I	Main dictionary index file
DICT25.S	Small word dictionary
DICT25.APP	Application dictionary
DICT25.U	User Dictionary

C. Source code files for the SpellTime DLL:

SPELL.C	The 'C' source code for SPELL.DLL (main program file).
SPELL.DEF	Definition file for SPELL.DLL
SPELL.RC	Resource file for SPELL.DLL
SPELL.RES	Compiled resource file for SPELL.DLL
SPELLDLG.DLG	Dialog definition for SPELL.DLL
SPELLDLG.H	Control Ids for the dialog definitions.
SPL_DEF.H	Constant definition file.
SPELLDLG.RES	Compiled dialog definition file. Use the SPELLDLG.H and SPELLDLG.RES files with the Windows' Dialog Manager.

D. Demo files:

DEMO.EXE	The executable demonstration program
DEMO.C	A demo of SpellTime function calls
DEMO.H	Include file for DEMO.EXE

DEMO.DEF	Definition file for the demo program.
DEMO.RC	Resource definition file.
DEMO.RES	Compiled resource file.
DEMO_DLG.DLG	Contains the dialog definitions.
DEMO_DLG.H	Control Ids for the dialogs.
DEMO_DLG.RES	Compiled dialog definition file for the demo program. Use DEMO_DLG.H and DEMO_DLG.RES with the Windows' Dialog Manager.

E. Make files to rebuild the DLL and the demo program:

MAKES-BC.BAT	Compiles and links DEMO and SPELL.DLL
MAKES-BC	using the Borland 'C' compiler
MAKES-MC.BAT	Compiles and links DEMO and SPELL.DLL
MAKES-MC	using the Microsoft 'C' compiler

F. Visual Basic Support Files:

SPELL.BAS	Function declaration and constant definition file. This module must be included into your Visual Basic application.
DEMO_VB.MAK	Make file for the demo program
DEMO_VB.FRM	Main form file for the demo program
DEMO_VB1.FRM	Form file for individual word spell checking.
DEMO_VB2.FRM	Form file for file name input

License Key

Your license key is e-mailed to you after your order for SpellTime is processed.

When Using SpellTime with TE Edit Control:

When using SpellTime with TE Edit Control, please use the TerSetStKey method exported by TE Edit Control at the beginning of your program:

```
TerSetStKey("xxxxx-yyyyy-zzzzz")
```

Replace the 'xxxxx-yyyyy-zzzzz' by your *SpellTime* license key. The license key for TE Edit Control will not work with the TerSetStKey method.

Or, you can use the SpellTimeKey property of the Toc ActiveX control.

When using SpellTime stand-alone:

You would set the license key for a stand-alone usage of using the StSetLicenseKey static function. This should be preferably done before calling other SpellTime methods.

```
StSetLicenseKey("xxxxx-yyyyy-zzzzz")
```

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

SpellTime into Your Application

Follow these easy steps to incorporate SpellTime into your application:

1.Change your application's make file to include the SPELL.LIB (SPELL32.LIB for WIN32) in the link statement.
2.Choose a module in your application that will call the spell checker. Add a statement in this module to include the SPELL.H file for proper function declaration.
3.Extract the individual words from your application to spell check. You may use the StParseLine function to parse a buffer containing the text. Call the SpellWord function to check the extracted word. The DEMO.C file illustrates both the StParseLine and SpellWord function calls. These functions are also described in detail in the Callable DLL functions section.

SpellTime in multi-threaded environment

Each API function in SpellTime comes two forms. The first set works with the default speller session, whereas the second set of API functions work with specific speller session. The second set of API functions are suffixed with a letter '2' and are session dependent. For example, StParaseLine function assumes a default speller session, whereas the StParaseLine2 function works with a specific speller session. Before using a session dependent API function, your application needs to create a unique speller session using the StInitSession function. At the end of a spelling session, the spelling session is terminated using the StEndSession function.

The applications that need to use SpellTime within multiple threads, must create a unique speller session for the thread. This speller session id is used to call the session dependent API functions.

Example:

```
DWORD id=StInitSession();

StParseLine2(id, ...);   // parse the bufer and conduct

                            spell checking session

SpellWord2(id, ...);

StEndSession(id);        // terminate the session
```

Callable DLL Functions

The source code for the SpellTime DLL functions is contained in SPELL.C. The prototypes of the routines are contained in the SPELL.H file (Use the SPELL.BAS file with a Visual Basic application). Your program must include this file to ensure proper function declaration (see *Getting Started*). This section describes the routines in an alphabetic order.

The description of each function is divided in two parts. This first part describes the syntax, argument description and a brief description of the functionality. The second part describes the source code for the function.

Most functions have an additional session dependent form which can be used with a specific session id. These functions are suffixed with a letter '2'. A unique session id can be created using the StInitSession. After the spell checking session, the session id must be terminated using the StEndSession function.

In This Chapter

SpellDict

int SpellDict(*CheckWord,WordLen,flags*)

int SpellDict2(id, *CheckWord,WordLen,flags*)

DWORD id;	Session id.
LPSTR CheckWord;	(input) Word to look up. The valid characters in the word are lower case 'a' to 'z' and an apostrophe character. The word string must be NULL terminated.
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.
unsigned int flags;	(input) processing flags. (see the description below)

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

This function 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* function internally calls the *SpellDict* function. Use the *SpellDict* function 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 function after calling this function 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 function without the ST_GET_ALTERNATE flag. If the word is not found, then call this function 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 function 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 function to retrieve the values of these global variables

Example:

```
char OneWord[20]="January"

int WordLen;

struct StrStResult result;

strlwr(OneWord);       /* convert to lower case */

WordLen=strlen(OneWord);

if (STD_NOT_FOUND==SpellDict(OneWord,WordLen,0) {

   /* incorrect word */

   /* find alternate words */

   SpellDict(OneWord,WordLen,ST_GET_ALTERNATES);


   StLoadResult(&result)

   for (i=0;i<result.TotalAltWords;i++) {

     MessageBox(NULL,result.AltWord[i],"Alternate Words",MB_OK);

   }

}
```

▲ ▼

SpellString

int SpellString(InString, OutBuf, OutBufLen, hWnd)

int SpellString(id, InString, OutBuf, OutBufLen, hWnd)

DWORD id;	Session id.
char far * InString;	(input) The text to be spell-checked.
char far * OutBuf;	(output) This string pointer receives the corrected text.
int OutBufLen;	The length of the OutBuf string variable.
HWND hWnd;	Parent window handle

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

Return: This function returns the number of incorrect words found.

A negative return value indicates a processing error. A value of -2 indicates that the length of the output buffer (OutBufLen) is too small to contain the fully corrected text string. A value of -1 indicates a general processing error.

SpellWord

int SpellWord(*InputWord,flag,hWnd,result*)

int SpellWord2(id, *InputWord,flag,hWnd,result*)

DWORD id;	Session id.
LPSTR 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.
WORD *flag;*	(input) Process control flags.
HWND *hWnd*;	(input) Handle of the parent window or NULL
struct StrStResult *result;	(output) Structure to receive the result codes and alternative words. A non C/C++ application should specify a NULL value here, and use the StGetReplacement and StGetAlternateWord functions to retrieve the results from this function

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* function to extract the individual words. You can then use the *SpellWord* function 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_BEEP	Produce a beep sound to indicate an incorrect word.
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 function without the ST_INTERACTIVE flag. If the function returns with a FALSE value (misspelled word), call the SpellWord function again, but this time with the ST_INTERACTIVE flag turned on (see example).

This function also allows you to specify the handle of your application window. SpellTime uses this handle to disable your application window during the dialog box session. If you do not need this functionality, specify a NULL value for this parameter.

Return: This function returns a TRUE value if the word is found in the dictionary. The function 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 function 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 detail information about the processing in the StrStResult structure variable pointed by the last argument. The *code* variable within this structure will have one of these values:

(If your application language does not permit passing a structure pointer for the StrStResult structure, use the StGetReplacement and StGetAlternateWord functions to get the result).

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 is returned in the StrStResult structure variable *replace*.
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 is returned in the structure variable *replace*.
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 *code* variable can have more than one of these flags set. Use the logical *AND* (&) operator to test for a flag, i.e. if (code&ST_EXIT) ...

When the result variable is equal to one of the first four codes, a set of alternative words is also returned by the StrStResult structure. The *TotalAltWords* variable within the structure indicates the number of alternative words available. Whereas, the *AltWord* array contains the list of alternative words.

Example:

```
char OneWord[20]="January"

int WordLen;

struct StrStResult result;

if (!SpellWord(OneWord,0,NULL,&result) {

  /* incorrect word */

  /* highlight the misspelled word if desired */

  .

  .

  .

  /* call again with the ST_INTERACTIVE flag */

  SpellWord(OneWord,ST_INTERACTIVE,NULL,&result);
```

```
    if (result.code&ST_EXIT) return;
    if (result.code&ST_ERROR) return;
    if (strlen(result.replace)>0) {  /* replace */
      strcpy(OneWord,StReplace);
    }
  }
  else {
    MessageBox(NULL,"Correct Word",NULL,MB_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 function 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 function provides a much higher level of user interface compared to the SpellDict function.

StGetAlternateWord

int StGetAlternateWord(*WordNumber, AlternateWord*)

int StGetAlternateWord2(id, *WordNumber, AlternateWord*)

DWORD id;	Session id.
int WordNumber;	(input) Alternate word number to retrieve.
char far *AlternateWord;	(output) This string pointer receives the alternate word indicated by the first argument.

Description: This function can be used to retrieve the alternate words suggested by the SpellWord function. This function is only useful if your application language does not allow you to pass a structure pointer (StrStResult) to the SpellWord function to retrieve the result.

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

Example:

```
int TotalAlternateWords;

char AlternateWord[30];

int i;

if (!SpellWord(OneWord,0,NULL,NULL) {

  /* incorrect word */

 TotalAlternateWords=StGetAlternateWord(0,AlternateWord);

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

   /* retrieve each alternate word

   StGetAlternateWord(i,AlternateWord);

 }

}
```

▲ ▼

StGetReplacement

int StGetReplacement(*ReplaceWord*)

int StGetReplacement2(id,Replace*Word*)

DWORD id;	Session id.
char far *ReplaceWord;	(output) This string pointer receives the replacement word.

Description: This function can be used to retrieve the replacement word suggested by the SpellWord function. This function is only useful if your application language does not allow you to pass a structure pointer (StrStResult) to the SpellWord function to retrieve the result.

Return: The function returns the result code provided by a previous call to the SpellWord function. Please refer to the SpellWord function for a list of result code constants.

Example:

```
int ResultCode;

char ReplaceWord[30];

if (!SpellWord(OneWord,0,NULL,NULL) {

  /* incorrect word */

  ResultCode=StGetReplacement(ReplaceWord);

}
```

StAddVowel

BOOL StAddVowel(chr)

BOOL StAddVowel2(id, chr)

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

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

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

StLoadResult

int StLoadResult(*result*)

int StLoadResult2(id, *result*)

DWORD id;	Session id.
struct StrStResult far *result;	(output) Points to the structure which will receive the result.

Description: This function can be used to retrieve the alternate words after calling the *SpellDict* function with the ST_GET_ALTERNATES flag (see *SpellDict*). The calling routine passes a pointer to the *StrStResult* structure, which receives the result parameters.

```
struct StrStResult {

  WORD  code;

  char  replace[ST_MAX_WORD_LEN+1];

  int  TotalAltWords;

  char AltWord[ST_MAX_SUG_WORDS][ST_MAX_WORD_LEN+1];

}
```

This structure is defined in the SPELL.H file. Only the last two variables are used by the SpellDict function. The *TotalAltWords* variable within the structure indicates the number of alternative words available. Whereas, the *AltWord* array contains the list of alternative words.

Example:

```
char OneWord[20]="January"

int WordLen;

struct StrStResult result;

strlwr(OneWord);    /* convert to lower case */

WordLen=strlen(OneWord);

SpellDict(OneWord,WordLen,ST_GET_ALTERNATES);


StLoadResult(&result)

for (i=0;i<result.TotalAltWords;i++) {

  MessageBox(NULL,result.AltWord[i],"Alternate Words",MB_OK);

}
```

StClearHist

int StClearHist()

int StClearHist2(id)

DWORD id;	Session id.

Description: Use this function 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 function 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 function returns a TRUE if successful. Otherwise it returns a FALSE value.

Example:

```
StClearHist();

while (!EOF) {

  /* spell check statements here */

}
```

StEndSession

DWORD StEndSession()

Description: Use this function to terminate a speller session.

The session id returned by the StInitSession function can be used to call the session dependent API functions. At the end of the spelling session, call StEndSession to terminate a spelling session.

Return: This function returns a TRUE value when successful.

StInitSession

DWORD StInitSession()

Description: Use this function to create a new speller session.

The session id returned by this function can be used to call the session dependent API functions. At the end of the spelling session, call StEndSession to terminate a spelling session.

Return: This function returns a non-zero session id if successful. Otherwise it returns zero.

Example:

```
char OneWord[20]="January"

int WordLen;

struct StrStResult result;

DWORD id;

Id=StInitSession();        // create a new spell

                              checking session

// spell check one or more words

if (!SpellWord2(id, OneWord,0,NULL,&result) {   // note

   the use of SpellWord2 instead of SpellWord function.

 /* incorrect word */

 /* highlight the misspelled word if desired */

 .

 .

 .

 /* call again with the ST_INTERACTIVE flag */

 SpellWord2(id, OneWord,ST_INTERACTIVE,NULL,&result);


 if (result.code&ST_EXIT) return;

 if (result.code&ST_ERROR) return;

 if (strlen(result.replace)>0) {  /* replace */

   strcpy(OneWord,StReplace);

 }

}
```

```
else {

  MessageBox(NULL,"Correct Word",NULL,MB_OK);

}

StEndSession(id);     // terminate the spell checking session
```

StParseLine

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

int StParseLine2(id, *buffer,word,WordIndex,CurIndex,LineLen*)

DWORD id;	Session id.
LPSTR buffer;	(input) Pointer to the buffer containing the words to extract.
LPSTR word;	(output) Pointer to the location where the extracted word is to be copied.
LPINT WordIndex;	(output) Starting position of the extracted word with respect to the beginning of the buffer.
LPINT CurIndex;	(input/output) The function 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 function is to create words for the *SpellWord* function. Your application can call this function repetitively until all words from a buffer are extracted.

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

Example:

```
char line[100]="It pays to increase your word power

                  (Dr. Funk).";

char CurWord[30];

int LineLen,WordLen,CurIndex,WordIndex;

struct StrStResult result;

CurIndex=0;  /* initialize the beginning of search */

LineLen=strlen(line);

while ((WordLen=StParseLine(line, CurWord, &WordIndex,
         &LineIndex, LineLen) {
  if(!SpellWord(CurWord,ST_INTERACTIVE|ST_BEEP,NULL,&result){
    MessageBox(NULL,"Incorrect Word",NULL,MB_OK);
    if (result.code&ST_EXIT) return;
    if (result.code&ST_ERROR) return;
    if (strlen(result.replace)>0) {
      /* insert the new word in the line buffer */
      /* update the LineLen variable */
    }
  }
  else {
    MessageBox(NULL,"Correct Word",NULL,MB_OK);
  }
}
```

StResetUserDict

int StResetUserDict(*new,old*)

int StResetUserDict2(id, *new,old*)

DWORD id;	Session id.
LPSTR *new*;	(input) Pathname of the new user dictionary.
LPSTR *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 NULL, 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 DOS pathname.

This function 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 function with NULL arguments at the end of the spell checking session. Second, this function can be used to activate a new user dictionary before initiating a spell checking session. By default, the 'dict25.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 function 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


      StResetUserDict(NULL,NULL);   /* c/c++ example */


      call StResetUserDict(ByVal 0&, ByVal 0&) ' VB example


2.    open a new user dictionary

      char old[128];

      StResetUserDict("mydict",old);

      /* spell checking statements here */

      .

      .

      .
```

```
StResetUserDict(old,NULL);
```

StSetDictName

BOOL StSetDictName(DictName)

BOOL StSetDictName2(id, DictName)

DWORD id;	Session id.
LPSTR *DictName*;	(input) New name of the main dictionary excluding the file suffix. Example: dict25

Description: The standard dictionary name is dict25. If you are using a dictionary with a different name, use this function to specify the new name. This function must be called before calling any other SpellTime function.

Return: This function returns TRUE when successful.

StSetFlags

DWORD StSetFlags(set, flag)

DWORD StSetFlags2(id, set, flag)

DWORD id;	Session id.
BOOL set;	TRUE to set the flag, or FALSE to reset it.
DWORD 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 function 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 function returns the new value of the flag bits.

ToSpellHist

int ToSpellHist(*CurWord,flag,ReplaceWord)*

int ToSpellHist2(id, *CurWord,flag,ReplaceWord)*

DWORD id;	Session id.
LPSTR 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.
LPSTR 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 DLL. 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.

ToUserDict

int ToUserDict(*CurWord)*

int ToUserDict2(id, *CurWord)*

DWORD id;	Session id.
LPSTR 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 function at the end of your program to actually write the updated user dictionary to the disk file.

See Also
StResetUserDict

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 (dict25.s) into memory.

Application Dictionary: The memory requirement is equal to the size of the application dictionary. The application dictionary, as shipped by Sub Systems, is empty. To preserve memory, keep the size of the application dictionary to a minimum.

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.

Define Statements

SpellTime uses a number of global constants. Some of these constants are defined in the SPELL.H file. These constants are meant to interface with your application program. Other constants, which are defined in the SPELL.C module, are for the internal use of the SPELL.C module.

The Constants Defined in the SPELL.H File:

ST_MAX_WORD_LEN	Maximum length of the word processed by SpellTime.
ST_MAX_SUG_WORDS	Maximum number of alternate words returned by SpellTime. You can change this constant if you wish SpellTime to return a different number of alternate words
ST_GET_ALTERNATES	Get alternate words
ST_INTERACTIVE	Invoke spelling correction dialog box
ST_BEEP	Beep on spelling mistake
ST_IGNORE	Ingore the misspelled word
ST_REPLACE	Replace the misspelled word
ST_REFRESH	Refresh the screen
ST_TOO_LONG	Word too long
ST_ERROR	Processing error
ST_EXIT	User clicked on the Exit button on the dialog box
ST_ADD	Add to the user dictionary
ST_INPUT	Flag constant, do not modify.

Interface With TE Edit control

TE Edit control can interface with SpellTime without any coding on your part. Simply move the spell.dll (spell32.dll for Win32) to the directory where the ter.dll or ter32.dll is located. Then move all the dict25.* files to a directory which is accessible during runtime.

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

```
TerCommand(hWnd,ID_SPELL)
```

or, set the command property of the TER ocx:

```
command=ID_SPELL
```

The Spell Checker Demo Program

The SpellTime package comes with a stand-alone spell checker program called DEMO. DEMO is also designed to demonstrate, by example, the usage of the SpellTime routines.

The initial window of the demo program displays two menu selections: File checkup, and Individual Word Look up.

File Checkup:

This selection allows the user to spell check a text file. The user can specify the name of the input file and the update mode. Select the *Read Only* mode if you do not wish to write the corrections to the disk file. The *CheckFile* function conducts the spell checking session. It opens the file to read each text line. The *StParseLine* (see StParseline) function is used to derive individual words from the text line. The individual words are checked using the *SpellWord* (see SpellWord) function. This function allows the user to correct a misspelled word. The user can select a replacement word from a list of alternative words, or type in a new word. The user can also elect to write the word to the user dictionary.

Please note that before the spell checking session is begun, the CheckFile function calls the *StClearHist* function to purge the previous SpellTime history (see StClearHist). This calls instruct SpellTime to purge the previous list of 'ignored' and 'replace' word. This step is optional.

The program creates a backup of the input file before writing the corrected text to it. The name of the backup file is made up of the prefix of the input file and a .ST extension.

Individual Word Look up:

The second selection allows the user to check one word at a time. This option simply accepts a word in a dialog box, and calls the *SpellWord* function to check it. The correction status is displayed on the screen.

Note that the WinMain function toward the end includes a call to the *StResetUserDict* function. This function call ensures that the user dictionary buffer is written out to the disk file.

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 **DECOMP25.EXE** utility to decompress the main dictionary into a number of text files. The text files are named as DCT_A through DCT_Z, DCT_n, DCT_SML. DICT25.map. The DCT_A through DCT_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 (DCT_n) built by concatenating the ASCII value of the character to the prefix 'DCT_'. For Example, the DCT_39 file contains words that start with an apostrophe character, i.e. 'twill. The DCT_SML file is a copy of dict25.s, the small word dictionary. The words in the text files have compression codes in the form of a period (.) character.

The dict25.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:

DECOMP25 [/S]

The optional /S switch suppresses the program messages.

2) Run the **MERGE25.EXE** utility to merge a list of words contained in a word file to the DCT_* files. 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 dict25.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 MERGE25 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 DICT25.APP and DICT25.U files contain the words in the valid format. Thus, these files can be directly merged into the dictionary text files.

Syntax:

```
MERGE MergeFile [/S]
```

MergeFile: Name of the word file.

The optional /S switch suppresses the program messages.

Example:

```
MERGE dict25.u

MERGE dict25.app /S

MERGE YourMergeFile
```

As an alternative, you can also use a text editor to add or delete words form the DCT_A through DCT_Z, and DCT_n 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 dict25.map file. In the standard dictionary, the text files assume that the apostrophe character has a higher collating sequence than the letter 'z'. The MERGE25 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 MERGE25 program instead.

3) Run the **COMP25.EXE** utility to compress the dictionary text files (DCT_*) to form the dict25.d and dict25.s files.

Syntax:

```
COMP25 [/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 DCT_* 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 (dict25.map). The character map file is a plain ASCII file which contains the characters supported by the language. To retrieve the default map file, run the DECOMP25.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.

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 MERGE25.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 COMP25.EXE program to build the binary dictionary file.

Visual Basic Support

The SpellTime DLL functions are compatible with the Visual Basic environment.

The SPELL.BAS file provides the necessary interface for a Visual Basic application. Include the SPELL.BAS module in your applications. This file contains the DLL function declarations, constant definitions, type declarations, and auxiliary functions. This file essentially is the Visual Basic translation of the SPELL.H file that is used with a 'C' language application.

The SPELL.BAS module contains two additional constants: ST_TRUE and ST_FALSE. Use these constants to compare the result of a function call. Example:

```
if ST_TRUE = SpellWord(.....) then ....
```

The SpellTime manual describes each DLL function in detail. The function syntax description in the manual is applicable to the Visual Basic interface as well. Please refer to the DEMO_VB demo program for the examples of SpellTime function calls.

Output String Normalization: The SpellTime functions that return string output require special handling. Before calling the function the output string must be expanded to the maximum length that can be used by the DLL. You can use the Visual Basic 'space' function for this purpose. Besides, the DLL terminates an output string with a NULL character. This character must be stripped out from the output string. You can use the 'DiscardNull' function for this purpose.

Example 1:

```
Dim CurLine as string

Dim CurWord as string

Dim WordIndex as integer

Dim LineIndex as integer

Dim LineLen as integer

Dim WordLen as integer

CurWord = space(ST_MAX_WORD_LEN+1)   ' allocate enough

      space for the output variable

WordLen=StParseLine(CurLine,CurWord,WordIndex,

   LineIndex,LineLen)

CurWord=DiscardNull(CurWord)   ' output string

                               normalization
```

Note that the 'DiscardNull' function is not called for the CurLine variables, because this variable is used for input only.

Example 2:

```
Dim WordFound as integer

Dim result as StResult      ' StResult defined in SPELL.BAS

Dim ReplaceWord as string

Dim FirstAlternateWord as string

WordFound=SpellWord(CurWord,0,0,result)

ReplaceWord = DiscardNull(result.replace)

FirstAlternateWord = DiscardNull(result.AltWord(0))
```

▲ ▼

Visual C++ Interface

The SPELL DLL can be used with a Visual C++ application without any change. Simply include the SPELL.H file into your application module that calls the SPELL functions.

Recompiling SPELL DLL files

If you need to modify the DLL source code and recompile within the Visual C++ environment, follow these steps to create a Visual C++ project:

Files: SPELL.C, SPELL1.OBJ, SPELL.DEF and SPELL.RC

Executable Type: Windows DLL

Alignment (Compiler Option): 1 Byte

Remaining parameters should be left at their default values.

Dictionary Data Format

This section describes the data format of the various dictionary components. Normally, an application developer only needs to know the data format of the *application dictionary*. The discussion of the application dictionary is provided in the second section.

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: DICT25.D, DICT25.I and DICT25.S. The DICT25.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 DICT25.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 DICT25.I file contains the pointer to each word bucket. The DICT25.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 l 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

l    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 ("l"). 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_WORD) 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.

Application Dictionary

A developer needs to understand the format of the words in the application dictionary to add the words specific to their industry.

The format of the application dictionary is simple. You can use a text editor to add words to the application dictionary. The rules are simple. A word must be entered in lowercase with the first letter capitalized. The word must end with a comma. You can enter as many words in a line as you wish. For example, the contents of a biologist's application dictionary may look something like this:

```
Malpighian,Protonephridia,Excretory,

Solenocytes,Tubules,

Osmosis,Annelid,Haversian,
```

Although, new words can be added very easily to the application dictionary, the number of words in this dictionary should be kept as small as possible. Because, as the size of the application dictionary grows, the fixed memory overhead increases and the longer linear search reduces the overall performance.

User Dictionary

The user dictionary is automatically updated by the SpellTime routines. The internal format of the user dictionary is similar to the format of the application dictionary. However, since the user dictionary is not updated externally, it does not have <CR><LF> characters. All words are stored in lowercase with the first character capitalized. Each word is delimited by the comma character.