FieldPack™

Version 1.1

User's Manual

Software Source
42808 Christy St., Ste. 222
Fremont, CA 94538 USA
Tel +1-510-623-7854
Fax +1-510-651-6039

CONTENTS

---

1. Introduction

FieldPack extends Visual Basic for Windows with three sets of functions combined into a single, compact Dynamic Link Library (DLL).

FieldPack is distributed as shareware, so you can try it before you buy it; the registration fee is US$ 39. Look for the file FLDPAK11.ZIP on shareware bulletin boards, or call Software Source.

Software Source is the creator of VB/ISAM, the fastest database manager available for Visual Basic. Our products are known for professional, well-thought-out design with a simple user interface; compact, efficient code; intelligently written documentation; immediate, friendly support; and low prices.

SuperString Tools for VB (SSTools) pack and unpack any series of data items into and out of variable-length strings; by using those packed strings as memo fields in databases, you can store records of different formats in the same file. You can also nest SuperStrings without limit.

In particular, SuperStrings can store arrays of different numbers of elements into database records. For example, customer records can contain a variable number of sub-records, each representing an open invoice.

==========

Delimited Substring Tools for VB (DSTools) manipulate strings containing character-delimited substring fields, in a design modelled after the Pick Operating System. You can read, write, insert, and remove variable-length fields delimited by any sequence of characters (not just a single character); and, you can nest strings with different delimiters, for hierarchical data processing.

You can also use these functions for text processing, with (CR-LF) delimiting lines or paragraphs and the space character delimiting words.

==========

Utility String Tools for VB (USTools) perform a variety of useful string operations, including left, right, and center justification with a choice of fill character; blank trimming and compression; character translation, "strip out," and "leave only"; and more.

---

2. Plain-Language Software License and Warranty Disclaimer

The *UNREGISTERED* version of FieldPack produces a registration advisory message. When you pay a registration fee to Software Source, you receive a registration password. When you use this password, the message is suppressed, and that copy of FieldPack is considered *REGISTERED*.

You may freely copy and distribute the *UNREGISTERED* version of FieldPack, provided that you (1) don't modify it, (2) distribute all of its components, including this license text, and (3) don't charge more than a reasonable media and/or shipping/handling fee.

You may freely use the *UNREGISTERED* version of FieldPack in your application software development, and distribute its DLL as described in the documentation, provided that you don't modify the DLL or do anything else to suppress or modify its registration advisory messages.

You *ABSOLUTELY MAY NOT* use any FieldPack registration password except one you rightfully obtained by paying a registration fee to Software Source. These passwords, and their security, are extremely valuable to Software Source, and therefore *WE WILL PROSECUTE* anyone who doesn't use all best efforts to keep passwords secret.

You may freely, and without royalty obligation, use your *REGISTERED* version of FieldPack in your application software development, and distribute its DLL as described in the product documentation, provided that you don't modify the DLL, and further provided that the software you create using FieldPack isn't competitive with FieldPack itself.

YOU MAY NOT USE OR DISTRIBUTE THIS SOFTWARE EXCEPT AS SPECIFIED IN THIS SOFTWARE LICENSE.

This software is supplied with no warranties whatsoever. Don't even *think* about suing us.

---

3. Installation

FieldPack consists of 16 files:

(a) A README.TXT file; in particular, this will tell you how to set up and run the demo programs.
(b) A brochure on the VB/ISAM indexed data manager, in the form of a .WRI file: VBISAM.WRI.
(c) This documentation file, FLDPAK11.WRI.
(d) The DLL "engine," FLDPAK11.DLL. Put it into your Windows directory, and distribute a copy of it with your application software, to be installed in your end-users' Windows directories.
(e) A definitions file, FLDPAK11.BAS. Include it in your VB programs ("Add Module...") to make VB aware of the functions available in the DLL engine. FLDPAK11.BAS "declares" the functions and their parameters.
(f) Eleven demo program and data files; their names begin with the characters "FPDEMO."

After making the DLL available in your Windows directory (actually, anywhere on your DOS PATH will do), and including the .BAS file in your VB source program ("project"), you can write calls to the special functions just the way you call built-in VB functions.**

** [Almost. You can't use a VB *object property* (such as TextBox1.Text or Label1.Caption) as an input parameter unless you enclose it in parentheses. And, you can't use a VB object property as an OUTPUT parameter at all. (Use some simple variable -- like Temp$ -- for the output parameter in the function call, then assign that variable to the object property later.)]

The unregistered version of FieldPack will display a message box when the DLL is first called, to remind you to register the software. When you register, you receive a password. Call the FP_Password function, using the correct password, *before* you call any other FieldPack functions; the message box will not appear. Call the function as follows:

```
ReturnCode% = FP_Password(Password$)
```

This function always returns 0 (even if the password is incorrect).

---

4. SuperString Tools for VB

4.1 SSTOOLS: Concepts and Capabilities

SuperStrings are ordinary VB variable-length strings into which you have packed a series of data items using special tools. You can specify any sequence of data items of various data types -- integers, longs, strings, arrays of any kind, etc.

By using SuperStrings of different formats as Memo fields in database records (or, in general, wherever a database allows variable-length string fields), you can have mixed record types in the same database table (or file). As a simple example, you can pack arrays of varying numbers of elements into different records -- so that an array in the JoeSmith record may have 50 elements, while the corresponding array in the TomJones record has 100 elements.

IMPORTANT NOTE: *Do **NOT** use the VB3 "bound controls" to write or read SuperStrings into or out of database Memo fields. The reason is that Visual Basic strips non-printing characters out of object properties (such as Label1.Caption or TextBox1.Text), and SuperStrings **will** contain non-printing characters. Instead, use code similar to the following (it'll be faster, too):*

```
Dim SuperString As String    'Note, variable length string variable.
...
DataControlName.Recordset.Fields("MemoFieldName").Value = SuperString$   'Write it!
...
SuperString$ = DataControlName.Recordset.Fields("MemoFieldName").Value   'Read it!
```

SuperString Tools for VB is a set of functions that provide two ways to pack and unpack SuperStrings:

(a) All at once: The SS_StoreAll function converts the entire contents of a Type variable into a SuperString. The SS_FetchAll function unpacks a SuperString into the components of an appropriately structured Type variable.

(b) Selectively, by field: SS_StoreItem writes (or re-writes) an individual data item into a specified position within a SuperString. SS_FetchItem reads an individual data item from a specified position within a SuperString.

4.2 SSTOOLS: SuperString Format Descriptions

Since SuperStrings may be packed with any arrangement of data items, you must supply each of the SuperString access functions with a "format description string" parameter that specifies the layout of items within a given SuperString. You must construct these format description strings according to specific rules, given below.

Here are two "preview" examples of simple format description strings:

"%, %, $*15, #"	Two integers, a 15-byte fixed-length string, and a Double.
"!, A100%, $, $*20, %"	A Single, a 100-element array of integers, a variable-length string, a 20-byte fixed-length string, and an integer.

Note that the only way you can unpack data from a SuperString is to use the same format description you used when you built it. Therefore, if you're going to store SuperStrings of different

formats into different database records, each of those records must further include -- in a separate field -- some indication of that record's SuperString format, either as a complete format description string or as a reference to one of several predefined formats.

Rules for constructing format description strings are as follows:

You must describe each "piece" of a SuperString in its format description string. You may (optionally) separate the piece descriptions with spaces and/or commas. A "piece" is either an *individual* data item (such as an integer) or an *array* of individual data items (such as an array of integers).

Represent individual data items with the standard type-declaration characters used in Visual Basic, as follows:

%	Integer
&	Long
!	Single
#	Double
@	Currency
$*n	Fixed-length string, *n* bytes long
$	Variable-length string

(Note that there's no representation for "variants." That's because variants don't really exist.)

Represent arrays with the letter "A," the number of array elements, and the data type (as above). For example:

A50%	... an array of 50 integers.
A50$*10	... an array of 50 fixed-length strings, each 10 bytes long.

If you want to pack an array of more than one dimension, multiply the bounds to arrive at the total number of elements. For example, a two-dimensional array with 6 rows and 4 columns has 24 elements.

If you have a series of identical consecutive pieces in your format description string, you can (optionally) save space by representing such repeating pieces with a "repeat count": the letter "R," the number of times the piece description would otherwise appear, and the piece description. For example:

R4%	... equivalent to %, %, %, %
R4A50%	... equivalent to A50%, A50%, A50%, A50%
R4A50$*10	... equivalent to A50$*10, A50$*10, A50$*10, A50$*10

*** WARNING: Don't confuse "repeated pieces" with "pieces." For example, the format description string "R5%, #, R10A3@" contains 16 pieces, not three. This is important if you're going to use the auxiliary functions, described later.

You may not repeat groups of pieces, nor may you "nest" repeats; that is, there is no such syntax as, for example, R5(%, #, @) or R5R3%.

[Tech. note for expert hackers: Arrays of variable-length strings are NOT the same as repeats of individual variable-length strings; i.e., R99$ is NOT identical to A99$. Don't get cute.]

There is no length restriction on format description strings.

4.3 SSTOOLS: Function Return Codes

Remember to test the return value of every function call for error conditions. For example:

```
DIM ReturnCode as Integer
ReturnCode = SS_StoreAll (...)
If ReturnCode < 0 Then
    'Process error condition...
```

Eight of the nine functions return integer values. For these functions, negative values signify errors, as follow:

-1 BAD FORMAT. Either the Format Description String is invalid (improperly constructed), or the actual data types are not as described in the format description string (you lied to us -- or maybe used the wrong format description string). ***THE MOST COMMON FORMAT-MISMATCH ERRORS ARE:*** (1) miscounting data items (and, note that we consider all counts to begin with 1, not 0); and (2) confusing fixed-length strings and variable-length strings. (They're different data types!!)

-2 BAD PARAMETER VALUE. One of your input parameters has a nonsensical value; for example, a negative item number.

-3 OUT OF MEMORY (or system error; something didn't work right).

The ninth function, SS_ItemType, returns a string value. Nevertheless, if one of the above three situations occurs, that string value will be "-1", "-2", or "-3", as appropriate.

4.4 SSTOOLS: SS_StoreAll and SS_FetchAll

SS_StoreAll (SourceTypeInstance, SSFormat$, Destination$)

SourceTypeInstance is the name of an instance of a user-defined Type (i.e., a structured variable whose structure you defined with the Type and EndType statements).

SSFormat$ is a format description string, formed according to the rules given above, that describes the structure of the Type.

Destination$ is the name of an ordinary variable-length-string data variable. Do *NOT* specify a VB object property, such as TextBox1.Text or Label1.Caption, for the Destination$.

OPERATION: The function converts the entire contents of the source Type into an output SuperString, replacing the previous value of the Destination string. (The source Type variable is unaffected.)

FUNCTION RETURN VALUES: 0 if OK, else -1, -2, or -3, as described above.

SS_FetchAll (SourceSuperString$, SSFormat$, DestinationTypeInstance)

SourceSuperString$ is the name of a string variable whose contents were generated by either SS_StoreAll or SS_StoreItem.

SSFormat$ is a format description string, formed according to the rules given above, that describes the internal data structure of the source SuperString; in fact, it must be identical to

the format description string that had been used to create the SuperString.

DestinationTypeInstance is the name of an instance of a user-defined Type (i.e., a structured variable whose structure you defined with the Type and EndType statements); the structure of this variable must correspond exactly to the format of the source SuperString (and further to the format description string).

OPERATION: The function reads all data items from the SuperString and copies them into the corresponding components of the Type variable. (The source SuperString is unaffected.)

FUNCTION RETURN VALUES: Same as for SS_StoreAll.

4.5 SSTOOLS: SS_StoreItem and SS_FetchItem

SS_StoreItem (TargetSuperString$, SSFormat$, ItemNo%, ItemType$, Data)
SS_FetchItem (SourceSuperString$, SSFormat$, ItemNo%, ItemType$, Data)

ItemNo%: The sequential number of the individual data item within the SuperString. WHEN CALCULATING ITEM NUMBERS, CONSIDER SUPERSTRINGS TO BE LINEAR SEQUENCES OF INDIVIDUAL DATA ITEMS, STARTING WITH ITEM NO. 1. For example, if your SuperString consists of an Integer, a 10-element array of Longs, and a Currency, the Currency is item number 12. SS-StoreItem and SS-FetchItem each deal with a SINGLE data item at a time; note that arrays are NOT single data items.

ItemType$: This is just the data type of the selected item, represented as in format description strings: "%", "&", "!", "#", "@", "$*n", or "$". (Note that for fixed-length strings, we need the actual byte-count, not the letter "*n*".)

FUNCTION RETURN VALUES: 0 if OK, else -1, -2, or -3 as described earlier.

-> *SS_StoreItem operation:*

If the target SuperString is null -- that is, if you've set it to "" prior to calling SS_StoreItem -- the function first initializes the target SuperString. ***IF YOU'RE GOING TO USE A SERIES OF CALLS TO SS_STOREITEM TO BUILD A SUPERSTRING, BE SURE TO SET THE STRING TO NULL -- "" -- BEFORE THE FIRST CALL.*** The function stores the data item (given in the final parameter) into the proper position in TargetSuperString$, as specified by the ItemNo% parameter. (The purpose of the ItemType$ parameter is just to double-check for consistency; it must agree with the format description and with the actual datatype of the data item.) Do *NOT* specify a VB object property, such as TextBox1.Text or Label1.Caption, for TargetSuperString$.

-> *SS_FetchItem operation:*

Reads the data item specified by ItemNo% from the source SuperString, double-checks its type against the ItemType$ parameter, and copies the value of the data item into the variable whose name appears as the last parameter in the function call. BE SURE TO SPECIFY A DESTINATION DATA-ITEM VARIABLE OF THE CORRECT DATA TYPE: If you specify a destination shorter than the actual data, the function call can overwrite memory and louse up your day. Do *NOT* specify a VB object property, such as TextBox1.Text or Label1.Caption, for

the destination.

4.6 SSTOOLS: Auxiliary Functions

SuperString Tools for VB includes five auxiliary functions for examining format description strings at run time. Their purpose is to support complex, general-purpose SuperString operations; most ordinary applications won't use them.

Four of these functions return positive integer values (counts, for example); in case of error, these functions will return -1, -2, or -3, as described above. The fifth function returns a string value; in case of error, it will be "-1", "-2", or "-3".

Note the definitions of "piece" and "item" given in section 4.2, above.

SS_PieceCount (SSFormat$)

Returns the number of "pieces" in the SuperString described by the given format description string.

Examples:

"%, %, A10&, %"	describes four pieces.	(Note, 13 items.)
"%, %, A10&, R100%"	describes 103 pieces.	(Note, 112 items.)
"%, %, A10&, R100A99%"	describes 103 pieces.	(Note, 9912 items.)

SS_PieceItemNo (SSFormat$, PieceNo%)

Returns the item number of the FIRST ELEMENT OF the specified piece of the SuperString described by the given format string. For example, if the format string is "#, R5$*123, @, A10$, $, R4%, !", the first element of piece number 11 (the second integer of the group of four integers) is item number 20.

SS_PieceSize (SSFormat$, PieceNo%)

Returns the number of items in the specified piece. For example, if the format string is "R2@, A50%", the third piece of the corresponding SuperString contains 50 items. (Again, note that the SuperString described by this format description string contains *three* pieces, not two.)

SS_ItemCount (SSFormat$)

Returns the total number of items (not pieces) described in the given format description string.

**SS_ItemType$** (SSFormat$, ItemNo%) *[Note, this function returns a string value.]*

Returns, in a string, the data-type symbol for the specified item in the given format description string. Examples: "%", "$*789".

5. Delimited Substring Tools for VB

5.1 DSTOOLS: Concepts and Capabilities

Delimited Substring Tools for VB is a set of utility functions for string processing, based on the concepts of the Pick Operating System, in which "fields" are variable-length substrings that are delimited by a character of your choice. (Delimiters can also be a *sequence* of characters, such as carriage-return/line-feed.)

For example, if your chosen delimiter is a semicolon, then the string "abc;def" contains two fields: field 1 is "abc," field 2 is "def." (Note that, with a semicolon delimiter, the string "abc;" also contains two fields; the second is the null string.)

If a string contains n delimiters, it also contains n+1 fields.

By definition, the earliest field is "field 1"; there is no "field 0."

Note that you can NEST fields by using different delimiters in an appropriate manner. For example, you might choose to delimit "major" fields with semicolons, and have comma-delimited "minor" fields within some or all of the major fields.

5.2 DSTOOLS: Main Functions

Four of the five main functions return string values. In case of error, these functions return the null string. Error conditions are:

(a) FieldNumber% parameter less than 1;
(b) Delimiter$ parameter null (no characters);
(c) out of memory, or internal/system error.

DS_PutField (OriginalString$, Delimiter$, FieldNumber%, NewFieldValue$)

Returns a modified copy of the OriginalString$ with a copy of the NewFieldValue$ string inserted into the OriginalString$ as the FieldNumber%-th field, with fields delimited by Delimiter$. If there is already data in the FieldNumber%-th field, it will be replaced by the copy of the NewFieldValue$. NewFieldValue$ may be the null string. If there is no FieldNumber%-th field (i.e., if there are FieldNumber%-2 or fewer occurrences of Delimiter$ in OriginalString$), the function will append the required number of occurrences of Delimiter$, followed by the copy of NewFieldValue$.

Example 1:

OriginalString$:	"abc,def,ghi"
Delimiter$:	","
FieldNumber%:	2
NewFieldValue$:	"NEW DATA"
...function returns:	"abc,NEW DATA,ghi"

Example 2:

OriginalString$:	""
Delimiter$:	","
FieldNumber%:	4
NewFieldValue$:	"NEW DATA"
...function returns:	",,,NEW DATA"

Example 3:

OriginalString$:	"abc,def,ghi"
Delimiter$:	"/"
FieldNumber%:	2
NewFieldValue$:	"NEW DATA"
...function returns:	"abc,def,ghi/NEW DATA"

DS_GetField (SourceString$, Delimiter$, FieldNumber%)

Returns the value of the FieldNumber%-th field within the SourceString$, delimited by Delimiter$.

Example 1:

SourceString$:	"abc,def,ghi"
Delimiter$:	","
FieldNumber%:	2
...function returns:	"def"

Example 2:

SourceString$:	"abc,def,ghi"
Delimiter$:	","
FieldNumber%:	4
...function returns:	""

Example 3:

SourceString$:	"abc,def,ghi"
Delimiter$:	"/"
FieldNumber%:	2
...function returns:	""

Example 4:

SourceString$:	"abc,def,ghi"
Delimiter$:	"def"
FieldNumber%:	2
...function returns:	",ghi"

DS_InsertField (OriginalString$, Delimiter$, FieldNumber%, NewFieldValue$)

Returns a modified copy of the OriginalString$ with a copy of the NewFieldValue$ string inserted into the OriginalString$ as the FieldNumber%-th field, with fields delimited by Delimiter$. By contrast with DS_PutField, any higher-numbered fields will have their field numbers increased by one, since DS_InsertField adds a delimiter. (If there are no higher-numbered fields, the results will be the same as with DS_PutField.)

Example 1:

OriginalString$:	"abc,def,ghi"
Delimiter$:	","
FieldNumber%:	2
NewFieldValue$:	"NEW DATA"
...function returns:	"abc,NEW DATA,def,ghi"

Example 2:

OriginalString$:	""
Delimiter$:	","
FieldNumber%:	4
NewFieldValue$:	"NEW DATA"
...function returns:	",,,NEW DATA"

Example 3:

OriginalString$:	"abc,def,ghi"
Delimiter$:	"/"
FieldNumber%:	2
NewFieldValue$:	"NEW DATA"
...function returns:	"abc,def,ghi/NEW DATA"

DS_RemoveField (OriginalString$, Delimiter$, FieldNumber%)

Returns a modified copy of the OriginalString$ with the FieldNumber%-th field, and an adjacent occurrence of Delimiter$, removed. Any higher-numbered fields will have their field numbers decreased by one.

Example 1:

OriginalString$:	"abc,def,ghi"
Delimiter$:	","
FieldNumber%:	2
...function returns:	"abc,ghi"

Example 2:

OriginalString$:	"abc,def"
Delimiter$:	","
FieldNumber%:	2
...function returns:	"abc"

Example 3:

OriginalString$:	"abc,def"
Delimiter$:	"/"
FieldNumber%:	2
...function returns:	"abc,def"

DS_FindField (SourceString$, Delimiter$, StartingFieldNumber%, SearchArgument$, Mode%)

Searches the SourceString$, *beginning with the StartingFieldNumber%-th field* (with fields delimited by Delimiter$), for the first (earliest) field that matches*, begins with**, or contains*** the SearchArgument$; if successful, returns that field's field-number as an **integer**. If the search is not successful, returns 0. WARNING: Results are unpredictable if the SearchArgument$ contains one or more occurrences of Delimiter$.

* If Mode% = 1 or 5, requires an *exact match* with the SearchArgument$.
** If Mode% = 2 or 6, succeeds if a field *begins with* (or matches) the SearchArgument$.
*** If Mode% = 3 or 7, succeeds if a field *contains* (or matches) the SearchArgument$.

If Mode% = 1, 2, or 3, the function makes its comparisons case-sensitively (so that, for example, **ABC** and **abc** are *NOT* considered equal)..
If Mode% = 5, 6, or 7, the function makes its comparisons case-**IN**sensitively (so that, for example, **ABC** and **abc** *ARE* considered equal).

If Mode% is less than 1, equal to 4, or greater than 7, the function returns -1 (error).

Note: The intention of the StartingFieldNumber% is to give you a quick way to find successive occurrences of the SearchArgument$. If you find what you're looking for in field n, you can immediately look for further occurrences beginning with field n+1.

Example 1:

SourceString$:	"abc,xyabc,abzz,abcd"
Delimiter$:	","
StartingFieldNumber%:	2
SearchArgument$:	"ab"
Mode%:	2 [search for a field that *begins with* the SearchArgument$]
...function returns:	3

Example 2:

SourceString$:	"abc,xyabc,abzz,abcd"
Delimiter$:	","
StartingFieldNumber%:	2
SearchString$:	"ab"
Mode%:	3 [search for a field that *contains* the SearchArgument$]
...function returns:	2

5.3 DSTOOLS: Auxiliary Functions

DS_FindDlm (StringVariable$, Delimiter$, OccurrenceNumber%)

This function returns a **long** value (&): the byte position of the OccurrenceNumber%-th occurrence of the first character of Delimiter$ within StringVariable$. If not found, or if there's an error (e.g., you specified a null Delimiter$), the function returns 0.

Example:

StringVariable$:	"abc/*def/*ghijk/*lm"
Delimiter$:	"/*"
OccurrenceNumber%:	2
...function returns:	9

DS_CountDlms (StringVariable$, Delimiter$)

This function returns an **integer** value (%): the number of occurrences of Delimiter$ that it found within StringVariable$. (So, if you're using the StringVariable$ as a Pick-like container

of fields, the number of fields is one greater than the number of delimiters.) If there are none, or if there's an error (e.g., you specified a null Delimiter$), the function returns 0.

DS_ReplaceDlms (SourceString$, OldDelimiter$, NewDelimiter$)

Returns a modified version of SourceString$ in which every occurrence of OldDelimiter$ has been replaced with NewDelimiter$. Note again that delimiters may consist of more than one character.

DS_TellField (SourceString$, Delimiter$, Position&)

This function returns an **integer** value: the field number of the Delimiter$-delimited field that includes the byte at position Position& in the SourceString$. If that position contains a delimiter character, the function returns the *negative* of the preceding field's field-number. If Position& is not within the range of the size of SourceString$ (with the first byte being position 1), the function returns 0.

Example 1:

SourceString$:	"abc/*def/*ghi"
Delimiter$:	"/*"
Position&:	7
...function returns:	2

Example 2:

SourceString$:	"abc/*def/*ghi"
Delimiter$:	"/*"
Position&:	10
...function returns:	-2

---

6. Utility String Tools for VB

The Utility String Tools for VB functions all return string values, usually a modified version of a "SourceString$" parameter; SourceString$ itself is never affected. In case of error, they will return the null string.

US_LJustify (SourceString$, Width%, FillCharacter$)
US_CJustify (SourceString$, Width%, FillCharacter$)
US_RJustify (SourceString$, Width%, FillCharacter$)

These three functions return a string that consists of a copy of the SourceString$ left/center/right justified within a larger string of Width% characters, filled with the FillCharacter$.

If Width% is less than the length of SourceString$, the returned value will consist of a truncated version of the SourceString$: US_LJustify and US_CJustify truncate the right end; US_RJustify truncates the left end.

If the length of the FillCharacter$ parameter is greater than one, the functions will use only the leftmost character. If FillCharacter$ is the null string, the functions will use a space character instead.

A call to US_CJustify that results in an odd number of fill characters will center-justify with one more fill character on the right than on the left.

US_Proper (SourceString$)

Returns a modified version of the SourceString$ in which the first letter of each word has been changed, if necessary, to upper case, and all other letters in each word have been changed, if necessary, to lower case. (For the purposes of this function, a "word" is defined to begin after a space, period, comma, semi-colon, slash, carriage return, line feed, tab, or the beginning of the SourceString$. Upper-casing is done only if the first character of a word is in fact alphabetic -- so that the "word" **123abc** does *NOT* become **123Abc**.)

US_Trim (SourceString$)

Returns a modified version of the SourceString$ in which:

(a) Any leading space characters have been removed;
(b) any trailing space characters have been removed; and
(c) any embedded sequences of more than a single space have each been replaced with a single space character (i.e., multiple internal spaces have been collapsed).

US_StripOut (SourceString$, CharacterList$)

Returns a modified version of the SourceString$ in which all occurrences of EACH CHARACTER in CharacterList$ has been removed.

US_LeaveOnly (SourceString$, CharacterList$)

Returns a modified version of the SourceString$ in which all occurrences of all characters NOT included in the CharacterList$ have been removed.

US_Translate (SourceString$, OldCharacterList$, NewCharacterList$)

Returns a modified version of the SourceString$ in which each occurrence of each character in the OldCharacterList$ has been replaced with the positionally corresponding character in the NewCharacterList$. The OldCharacterList$ is processed from left to right, one character at a time. OldCharacterList$ and NewCharacterList$ must be the same length (else error).

US_Reverse (SourceString$)

Returns a modified version of the SourceString$ in which the sequence of characters has been reversed. For example, if SourceString$ is "abcd", the function will return "dcba".