SPFLite Edit Macros - V10.1.8351 - 2018-12-16

Table of contents

Introduction

The Edit Macro support in SPFLite provides a means to automate complex or repetitive editing tasks, and to create functions not available with SPFLite's existing built-in edit commands.

Operating Modes

SPFLite supports running macros in two modes.

First, as Primary commands. As primary commands macros can utilize operands coded on the command line following the macro name and can also utilize line ranges marked via the built-in line commands. e.g. a primary macro can reference line ranges marked with **CC**/**CC** markers.

Second, as Line commands. Line command macros can utilize either single line or block mode line marking. Being line commands, these macros usually have short names. Whether a line command is considered a single line or a block mode line command is discussed in the next section "Macro Format and Structure".

For Line command macros, because of their nature, there are some additional restrictions on usage

- Line command macros cannot be entered at the same time as commands on the command line.
- Only one line command macro at a time may be entered; and only one instance of that macro. e.g. for the macro **AX** you could not enter a series of individual **AX** line commands. A pair of block mode **AXX** lines is considered **one** instance.
- Line command macros can not be entered at the same time as other built-in line commands which are considered 'source' line markers, like **CC** or **MM**, since line command macros are themselves always treated as a source range.

Script Engine

The script engine chosen to support the macro facility is called **thinBasic** (www.thinBasic.com).

Don't let the word "**thin**" fool you. **thinBasic** is no "toy" language. This is a very capable script engine written in, and modeled after, PowerBasic (www.PowerBasic.com). Since SPFLite itself is written in PowerBasic, the interface between SPFLite and **thinBasic** is quite straightforward, which makes possible a highly efficient implementation. All of our test scripts have run very quickly, and you should see similar good performance as you write your own scripts.

Being a BASIC variant, users with experience in typical programming or scripting languages should recognize most of the **thinBasic** syntax. Many sample macros have been provided to help you get started.

As you will quickly notice, keywords and syntax in **thinBasic**, like SPFLite, are case-

insensitive. When you read the documentation and sample code that follows, the choice of capitalization you see is simply a style choice. You are free to capitalize (or not) any way you see fit.

thinBasic components are included in the normal SPFLite install, so no separate install of **thinBasic** itself is required. If however you would like to explore **thinBasic** as a normal scripting language, then you should obtain their full install package and install it. We recommend, as of this writing (May 2013), you install release 1.9.6.

> If you go that route, be prepared for a lot of reading. The full **thinBasic** help document is included in the basic SPFLite install, and can be accessed from the Start Menu SPFLite program group, or from within SPFLite itself by entering `HELP THINBASIC.` You will be surprised how much is in it. By way of full disclosure, you may also find that some of the documentation is incomplete. The developers of **thinBasic** tend to develop first and document later. It's a nice package, but it is what it is.

This document is organized into the following sections.

- **The SPFLite interface**. This contains the reference documentation for each of the SPFLite interface calls.
- **Working with the interface.** This contains tutorials on interface techniques: How to handle command parameters, how to search for and manipulate text lines, how to create a new Line command, how to control the edit cursor location, etc.
- **Sample macros.** Several of the included sample macros are shown with full comments.
- **thinBasic essentials.** This is a simplified introduction to **thinBasic**. It is not a complete and definitive description of all **thinBasic** capabilities and functions, but should be enough to get you by. For more than that, refer to the **thinBasic** Help file as described above.

Working with the thinBasic Interpreter

As interpreters go, **thinBasic** is quite sophisticated. However, it remains a true interpreter. What that means to you is that it does not perform a "full program parse" before beginning to execute your script. Instead, it reads, parses and executes a line at a time. So, if you have a macro that's 10 lines long, and there is an error in line 10, it will perform the first 9 statements, then halt with an error-detected message. You will have to correct your errors one at a time until you get past this.

In practical terms, you may find it beneficial to write larger macros in pieces, starting small and testing a bit at a time, and adding more code as you go along, to minimize any problems from syntax errors you might introduce into your code.

You can also develop "pieces" of **thinBasic** code and **#INCLUDE** them into your macro. Doing this is a way to 'modularize' your code, so that you can deal with tested pieces of **thinBasic** code that you know will work correctly. This is a technique that might help you if your macro is quite large. See the **thinBasic** documentation regarding the **#INCLUDE** statement. **Note:** At this point, the **#INCLUDE** statement does not appear to handle unqualified file names consistently. Until this is resolved, it would be prudent to fully qualify the included filenames.

Macro format and structure

Contents of Article

Introduction

The format of a macro is a standard text file of thinBasic source statements stored in the SPFLite MACROS folder.

For most users, the full path of this folder will be:

C:\Users**username**\My Documents\SPFLite\MACROS

The macro should be named **macname**.MACRO, where macname is the name with which you will be invoking the macro.

The string "**macname**" is also what is returned by the Get_MacName$ function. As with everything else in SPFLite, macro names are case insensitive.

Note: Because the macro's name gets parsed as part of a primary command or line command, you cannot use all legal Windows file name characters in a macro name, but should generally limit them to letters, digits and underscore.

Further, for line-command macros, the name must be even more restricted; the name cannot contain any digits, because they will be confused with numeric line-command operands. For line commands, the macro name should normally only contain letters to avoid problems. A few special characters were found to be acceptable, such as $ and @. We have not exhaustively tested every possible character, but if you can create a legal Windows file name with it, it will usually be valid as a macro name, except for the characters **+ - * ? : .** / and ****.

Line Macro names and Block Mode

For single line macros (e.g. do THIS to a single line) there is no problem, the line command = the macro name.

When you want to create a macro that can act as a block mode command (like **CC**/**CC**) the following methods are used:

- Repeating the last character of the line command. Using the same example as above, the block form of **AX** would be requested by entering **AXX**/**AXX**, and the macro still stored as **AX.MACRO**. This applies also to longer macro names. For example the block mode version of a macro called **BOX** would be **BOXX**, or for a macro called **PRINT** it would be **PRINTT**.

- Specifically tell SPFLite in what mode a macro name is to be treated. This is

done by issuing a SET command of the format:

```
SET MACROMODE.macname = BLOCK | LINE
```

If a **SET MACROMODE** has been issued for a macro name, the convention about repeating the last letter does not apply. Any length or format of macro name can be set unconditionally to a specific processing mode.

The only requirement SPFLite imposes on the format of a macro file is the macro prototype (or header), which must be the first line in the macro. Like everything else in SPFLite, the prototype is generally case-insensitive, unless it has default operands used as string values in FIND or CHANGE commands, for instance.

Macro Prototype

The first line of a macro must always be the macro prototype. This is a simple thinBasic comment statement of the format:

```
' macname.MACRO [ def-operand-1 def-operand-2 ... ]
```

The **.MACRO** part of the prototype is required. macname itself is optional, but should normally be the name of the macro. The brackets shown mean the list of operands is optional; don't actually code brackets here.

> Currently, SPFLite does not demand that **macname,** if coded, match the actual file name of the macro. However, for documentation purposes, it is best that you **do** make the name agree, just to keep things straight as you develop and use your macros. It is possible that a later release of SPFLite will require that the prototype name agree with the file name, so it's best to make them agree now.

You may optionally enter default values for macro operands (if your macro uses command-line operands). These are simple space-delimited strings which provide defaults if the relative operand number is not overridden when the macro is called. For example, given the following prototype:

```
' sample.MACRO  aaa  bbb  ccc
```

If the macro were invoked with the primary command line as **Sample** with no supplied arguments, then if the executing macro requested the number of operands via **Get_Arg_Count** it would receive 3. **Get_Arg$(1)** would be **aaa**, **Get_Arg$(2)** would be **bbb**, and **Get_Arg$(3)** would be **ccc**.

More details on retrieving macro operands and working with them will be found in Accessing Command Line Operands.

Macro Code structure

As described above, an SPFLite macro program is essentially "open code" in **thinBasic** and has no "structure" per se. By that, it means there is no MAIN Sub/Function, execution just starts at the first line and continues on.

As a result, there is, by default, no built in mechanism for SPFLite to 'pass in' the operands. This is easily correctable by making the mainline code into a thinBasic SUB routine. If this is

done, then the command line operands can be made available as passed parameters.

Before making this decision you should evaluate how you wish to handle your declared variables. In Open Code, all DIM'd variables are effectively Global and are 'visible' throughout your program. If you convert to structured mode and make the mainline a SUB, then DIM'd variables are only 'visible' throughout the mainline SUB. If you use additional SUB routines, then they will need to be passed any working data they require.

More details of this support will be found in Accessing Command Line Operands.

Locating macros and include files

Contents of Article

Introduction

As stated previously, macros are stored in the SPFLite MACROS folder, which will be

C:\Users**username**\My Documents\SPFLite\MACROS

If you are writing a simple, one-file macro, that is all you need to know. If you are writing a large macro in a modular fashion, or are writing a series of related macros having code in common, you would benefit from using **#INCLUDE** statements to organize your macro code into smaller and more manageable pieces.

The macro engine we use, called **thinBasic**, allows for **#INCLUDE** statements. However, in earlier versions of SPFLite, these were not convenient to use. According to the **thinBasic** documentation, an **#INCLUDE** statement with no explicit path is searched for in the following order:

1. the current script path
2. \thinBasic\inc
3. any #INCLUDEDIR paths in effect

When **thinBasic** is integrated into other software like SPFLite, step 2 becomes "\Program Files\SPFLite\inc". That can be a problem. Putting changeable user data under the Program Files directory is not a good idea, and starting with Windows 7, you may need Administrator privileges to do that. Since SPFLite has already set aside the MACROS folder, it makes more sense to be looking for included macro files there.

Automatic modification of the #INCLUDE statement at run time

To make this process simpler, SPFLite now will modify your **#INCLUDE** statements, so that if it detects a file name that has **no** "path qualifier" at the beginning of the name, the location of the SPFLite MACROS folder is added to the file name. That way, any "simple" names you use on **#INCLUDE** statements will automatically be searched for in the same place that your main macro is located.

> This "modification" to your **#INCLUDE** statements only involves altering an internal copy of these statements in memory, as they are being processed. Your **#INCLUDE** file itself on disk is not changed. SPFLite will never modify your macro or include files.

In deciding when to apply this modification, the beginning of the file name is examined. If that name begins with **/** or **** or **.** or if the second character of the name has a **:** colon, it is

assumed to have an "explicit path", and the file name is not modified.

> Note that **included** file names do not have to end with **.macro** but can be anything you wish. A common extension is .**inc** but that is entirely up to you.

This automatic modification of **#INCLUDE** statements is done only in the main macro file, **not** in nested files included by the main macro file. That is because these are read by thinBasic directly, and SPFlite does not "see" them.

For instance, if you write an **#INCLUDE** statement like this,

```
#INCLUDE "myfile.inc"
```

The name **"myfile.inc"** does not begin with **/** or **** or **.** or a drive letter like **C:**. SPFLite will see that there is no path qualifier on the file name, and will internally change the statement to something like this:

```
#INCLUDE "C:\Users\username\My
Documents\SPFLite\MACROS\myfile.inc"
```

Automatic insertion of #INCLUDEDIR directive at run time

Like many languages, **thinBasic** allows **#INCLUDE** statements to be nested. When you do this, **thinBasic** itself reads those nested files. SPFLite is not involved in that process, and does not read or see **nested** #INCLUDE statements. Because of this, the special handling described above is **not** done for such nested includes.

To help with this situation, SPFLite will automatically insert an **#INCLUDEDIR** directive just prior to the first **#INCLUDE** statement it detects in your main macro file (again, this is done internally - not to your macro file on disk). The "created" **#INCLUDEDIR** directive will specify the path to the SPFLite MACROS folder, so that nested includes will be searched for there.

> The **#INCLUDE** and **#INCLUDEDIR** statements are documented in the **thinBasic** Help. Issue a **HELP BASIC** command from SPFLite to bring up this Help.

Suppose you wanted you write your own **#INCLUDEDIR** directives as well. Will that work? Yes. You need to specify any desired **#INCLUDEDIR** directives prior to your first **#INCLUDE** statement. This will cause **thinBasic** to look for included files in the locations you specify, in the order you write them. SPFLite will simply ensure that, if an included file is not found in any of **those** locations, it will look in the SPFLite MACROS folder as a "last resort" search location.

This process should be taken into account if you are planning on creating some complicated search-path for a very involved macro or set of macros, that might also involve multiple versions of files (like, "test" vs. "production" versions). If you put a copy of an included file in the SPFLite MACROS folder, **and** you issue an **#INCLUDE** statement that, for whatever reason, does not find this file elsewhere, it **will** look the file in the SPFLite MACROS folder as the default search location. If that is not what you want, you should not put files in that default folder, but only in the explicit locations you need them to be in.

If your macro never issues an **#INCLUDE** statement, it will not insert this **#INCLUDEDIR** directive.

Support for SET names in #INCLUDE statements

You may wish to organize your included files into folders other than the default SPFLite MACROS folder, but you also would probably not like to use absolute path names embedded in your **#INCLUDE** statements. SPFLite now allows you do to this.

If you write an **#INCLUDE** statement like this,

```
#INCLUDE "=abc\myfile.inc"
```

SPFLite will look up the SET name you specify ("abc" in the example) and substitute the value of the SET variable into your **#INCLUDE** statement. SET names are defined by the SET primary command.

Suppose you issued the following SET primary command:

```
SET abc = "C:\mypath"
```

Then, when you run your macro that has the **#INCLUDE** statement above, the SET name is detected and looked up, and it value is substituted prior to thinBasic processing it. So, the file that actually gets included is:

```
#INCLUDE "C:\mypath\myfile.inc"
```

If your SET name "abc" happens to be undefined when you run your macro that has the **#INCLUDE** statement above, SPFLite will not attempt to modify it. You will then get a syntax error message from thinBasic that the name **`"=abc\myfile.inc"`** could not be found.

This modification of **#INCLUDE** statements to substitute SET name values is done only in the main macro file, **not** in nested files included by the main macro file. That is because these are read by **thinBasic** directly, and SPFlite does not "see" them.

Using relative paths for included files

If you want to organize a group of included files into its own folder, but you'd like to keep the process simple, you can use **relative paths** to accomplish this.

Suppose you wanted a set of included files to be stored under a folder called **`"xyz"`**. To do this, create a folder with the name **`"xyz"`** under the SPFLite\Macros directory. Then, you would write your **#INCLUDE** statements like this:

```
#INCLUDE "xyz\myfile.inc"
```

The name **`"xyz\myfile.inc"`** does not itself begin with **`/`** or **`\`** or **`.`** or a drive letter, even though there is a backslash after the **`"xyz"`** part. SPFLite will see that there is no path qualifier on the file name, and will internally change the statement to something like this:

```
#INCLUDE "C:\Users\username\My
Documents\SPFLite\MACROS\xyz\myfile.inc"
```


Accessing Command Line Operands

Contents of Article

Introduction

Except for the simplest of macros, command line operands are necessary to tailor the actions performed by your macro. So, retrieving and 'sorting out' what operands have been entered is a necessary requirement for most macros. The SPFLite macro support provides three methods for handling the operands. Which is best for you? The deciding factors are the number, order and complexity of the operands, whether any are optional, and whether any are keywords which possibly are part of a "keyword group". The chart below will help you by summarizing the differences in the three levels of support.

It is important to understand how the command line operands are handled. SPFLite does an initial parse of the command line operands, which are delimited by spaces or quotes as usual, and stores them into a table, which can be accessed with the Get_Arg$ and Get_Arg_Count functions.

These operands are not initially categorized or sorted in any way. You would simply "get" them, using the functions described next.

There are three types of macro operand access, as follows:

Basic Access

- The number of operands is available via Get_Arg_Count
- Each operand can be accessed as Get_Arg$(n), where **n** is the operand position on the command line; the first operand after the macro name itself is considered operand number 1. Operands are returned as STRING values, as indicated by the $ in the function name.
- If local variables are needed for processing the operand values, they must be declared in a DIM statement and assigned the Get_Arg$() value. These two steps can be combined if you wish, so that you could write **`DIM FIRST_ARG AS STRING = GET_ARG$(1)`** or these can be done as separate steps.
- All validation and handling of the operands is the responsibility of the macro code you write.

SUB Operand

- In creating the SUB structure, the operands are assigned, from left to right, to the arguments defined in the SUB statement.

Access

- Operands are accessed using the variable names used in the SUB statement, making the code much more readable. e.g. if the first operand of the SUB statement were coded as **`start-line AS STRING`** it makes code much more readable to code **`IF start-line = xxx`** than to code **`IF Get_Arg$(1) = xxx`** as in the Basic Access mode.

- Local variables do not need to be declared with a DIM statement for storage of the data.

- All validation and handling of the operands is the responsibility of the macro code you write.

Full Parse Access

- Most built-in SPFLite primary commands utilize 'order-independent' operands. For instance, you can say **`CHG FRED BILL ALL`** or **`CHG ALL FRED BILL`** or **`CHG FRED ALL BILL`** and SPFLite will figure out what you mean. Additionally, many keyword operands have multiple aliases, such as PREFIX, PRE, and PFX. Handling all these variations of operand formats in a macro, using your own code and positional operand access methods, would be difficult and error-prone.

- The function **SPF_Parse** does this 'sorting out' for you, identifying all the various operand types, keywords, and keyword aliases. This allows your macro to provide flexible operand handling with much less effort.

- The main requirement is for you to specify with the **SPF_Parse** function the detailed syntax requirements of your command. This involves informing SPFLite as to how many labels, tags, numbers, string values and keywords you expect to be used.

- **SPF_Parse** is normally be invoked at the beginning of the macro, and, if no scanning errors are reported, all the command line operands will have been categorized and set up for easy retrieval when you need them. Since this is an ordinary function like others provided by SPFLite, you are free to call this function wherever it is most appropriate for your logic.

- The categorization supports operand types such as:
  - Line references (like `.AA .BB .123`)
  - Tag references (like `:DD :S1`)
  - Text literals (like `ABC C'DEF' K5`)
  - Numeric literals (like `10 20 30`)
  - Keywords (like `ALL PREFIX | SUFFIX | WORD`)
    This support includes handling of aliases and mutually-exclusive keywords.

- Optional validation of Line References and Tag operands can be performed for you. Invalid (undefined) label or tag operands will cause SPF_Parse to report a syntax error. Because this process is optional, you can decide if you expect your label or tag operands to already exist, or if they might be new labels or tags that the macro itself would define in the course of its operation. To validate a label using macro code, you can try converting a label to a line pointer using Get_Lptr, where a non-zero result means the label is valid. To validate a tag, you can try issuing a

command like SPF_CMD("LOC :ABC FIRST") and if you get RC=0 the tag should be valid.

- Parameter validation that is specific to your macro's application must be done by your code.

Basic operand access

Using basic access, macro operands are accessed positionally by their position in the command string, from left to right. The total number of operands available is available by calling the Get_Arg_Count function. Each individual operand is obtained by calling Get_Arg$(n) where **n** is the operand number; the first operand after the macro name itself is operand number 1.

The macro may have default operands specified by the macro prototype line (see Macro Prototype) For example, given the following prototype:

```
' sample.MACRO  aaa  bbb  ccc
```

If the macro were invoked with the primary command line as **Sample** with no supplied arguments, then the **Get_Arg_Count** function would return 3. **Get_Arg$(1)** would be **aaa**, **Get_Arg$(2)** would be **bbb**, and **Get_Arg$(3)** would be **ccc**.

If the macro were invoked with the command **Sample RED GREEN BLUE YELLOW,** then the **Get_Arg_Count** function would return 4. **Get_Arg$(1)** would be **RED**, **Get_Arg$(2)** would be **GREEN**, **Get_Arg$(3)** would be **BLUE**, and **Get_Arg$(4)** would be **YELLOW**.

If the macro were invoked with the command **Sample RED,** then the **Get_Arg_Count** function would return 3. **Get_Arg$(1)** would be **RED**, **Get_Arg$(2)** would be **bbb**, and **Get_Arg$(3)** would be **ccc**.

All other validation of the operands is the responsibility of the macro itself.

SUB operand access

If you choose to code your macro in a more "structured" design, then you can have the command line operands provided to you as passed parameters to your mainline SUB subroutine. To do this, certain considerations which must be met.

To convert your macro to a structured format, you must place an initial SUB subroutine header immediately after the macro prototype line, as described in Macro Prototype. If you do this, and you normally use your macro from the primary command line with some fixed maximum number of parameters, SPFLite will assign these for you just like any other SUB subroutine would receive them, so that calls to **Get_Arg$** will not be needed.

This can be a quite useful feature, but it requires you to follow some fairly strict rules to take advantage of this**:**

- The initial **SUB** subroutine follows all the rules of **thinBasic** syntax, and must be ended by an **END SUB** statement.

- The SUB keyword must appear immediately following the SPFLite macro prototype line, meaning that the initial SUB keyword must be on line 2 of the MACRO file and no where else, and no comments may appear anywhere on the SUB statement.

- The initial SUB line may be continued on multiple lines if needed, by placing an _ underscore at the end of each continued line except the last one, as required by standard **thinBasic** line-continuation rules.

- The type of **each** parameter to the initial SUB must be **individually** declared with **AS STRING**. No other data type will work. **Note this point carefully. If you fail to do this, your macro will not work correctly**.

- In the body of the initial SUB subroutine, you can optionally issue the HALT statement anywhere you wish to end the macro. Otherwise, an implied HALT will be automatically issued when the subroutine issues an EXIT SUB, or passes through the normal END SUB.

- The initial SUB subroutine may have any valid name you wish.

- The initial SUB subroutine name need not match the name (if any) on the macro prototype header that appears on line 1 of the MACRO file, or the name of the MACRO file itself. However, for clarity and documentation purposes, it is recommended that you **do** make these names match. Future versions of SPFLite may or may not enforce such naming agreement.

- The parameters defined on the SUB statement are provided by merging the actual command line operands with the default operands which may be provided on the Macro Prototype header. When no value exists in either location, a null string "" is passed as the parameter.

- SUB subroutine statements can only define a fixed number of parameters. If you wish to access parameters that were specified on the primary command line beyond these, or need to handle a variable number of parameters, the **`Get_Arg$`** function and other related functions are available and can still be used as documented, even though you used an initial SUB subroutine.

- When you write a macro with an initial SUB subroutine, and the macro is launched as a line-command macro, only the defaults on the macro prototype header will be passed, and if the prototype has no defaults, all of the parameters passed to the SUB will be empty (null) strings.

NOTE: Unless you strictly follow the SUB formatting rules, SPFLite may not detect that you have put an initial SUB subroutine in your macro, and it will not get called.

> We have to perform a clever bit of "magic" to pull this off. SPFLite has to do a "read-ahead" and "peek" at your macro, and quickly analyze it to determine if you actually **have** an initial SUB subroutine like this. When it finds one, it inserts a call to that SUB, passing the appropriate parameters as needed. To do that, and do it reliably, you **must** ensure that the format of the SUB line strictly adheres to these rules.

In the initial SUB subroutine, you are free to call other subroutines and functions as desired. These can be built-in functions, functions imported by USES statements, or additional SUB and FUNCTION routines that you write yourself.

As with Basic Operand Access, validation of the command line operands is still entirely

up to the macro code to perform.

A note about macro parameter data type

It is important that you follow rule noted above to define all operands AS STRING. For example, this macro,

```
' sample.macro abc def
SUB sample (arg1,arg2,arg3 AS STRING)
    ' ... statements
END SUB
```

is **incorrect**. The reason is that, as written, the **AS STRING** clause does **not** apply to all three parameters, but **only to the last one**. This is simply a **thinBasic** syntax rule, and we have no control over it. However, you needn't worry about forgetting this rule, SPFLite will verify this has been done properly and reject the macro if it breaks the rule.

> What actually happens here is that when you have parameters written like "**arg1,arg2,**" with no type information specified, **thinBasic** treats these as parameters of type **VARIANT**. It's too complicated to discuss here, except it's just not what you want. SPFLite will only pass you parameters as strings anyway, so you have to cooperate and do it the right way.

The correct way to write it is like this:

```
' sample.macro abc def
SUB sample (arg1 AS STRING, arg2 AS STRING, arg3 AS STRING)
    ' ... statements
END SUB
```

If you like, you can put each parameter on a separate line, for readability and maintenance purposes, using line continuation like this:

```
' sample.macro abc def
SUB sample (arg1 AS STRING, _
            arg2 AS STRING, _
            arg3 AS STRING)
    ' ... statements
END SUB
```

Full-parse access

This support co-exists with all the Basic Access and SUB Operand Access methods for accessing macro operands. It make **no changes at all** to the **Get_Arg$** and **Get_Arg_Count** functions, or to the support for SUB operand handling.

You are free to choose whatever method suits you best, and you also can combine these methods. For instance, your macro operands might have a "simple" format that does not need to be parsed, and also a more complex format. You can choose to do the full parse if you determine that the simple format is not present.

The idea is that a call to SPF_Parse is made, passing a set of parameters that describe the syntax of your macro's operands. Using these parameters, and the initially-created contents

of the **Get_Arg$** data array, **SPF_Parse** will 'sort out' and validate the macro operands according to your specifications. If all is well, it returns an RC=0, and makes all those parameters available for use. If a parsing or validation error is found, it will issue an RC=8 along with an error message which you can obtain with **Get_Msg$** and display to the user.

Validation consists of the following:

- Depending on the optional validation flags, verification that the number of operands entered of that type meets the specification. Using flags, you can control whether an operand count represents an exact count, or is an upper limit (where the lower limit is zero), or if it is to be taken as an optional, "all or nothing" count.

- Where Line References are allowed, it can also validate that the line references entered are valid. For a line label like .ABC, the label must exist in your edit file to be "valid". For a line-number pseudo-label like .123, a data line on line 123 must exist, but since these are not actually labels, nothing else need be "defined".

- Where Tag operands are allowed, it can also validate that the Tag operands entered are valid Tags. A tag is valid if at least one tag of that name appears in your edit file.

- Where single keyword operands are specified, it simply tracks whether the keyword was specified or not. It is not an error if the keyword is absent.

- Where mutually exclusive keywords are specified, it validates that one and only one has been entered, This is the same principle used on SPFLite command like FIND, where you cannot say FIND FIRST LAST ABC.

- Where keywords have multiple aliases, it accepts them all and 'normalizes' them to return the first specified keyword. e.g. if the alias list were specified as `(EXCL,EXCLUDE,EXC,X)` then when any of them are entered it would be treated and returned as `EXCL`. This normalization process means that you don't have to check every possible spelling of a keyword, but only the first (left-most) one that appears in the list. This greatly simplifies the amount of work needed to check such keywords, since only value has to be tested.

On the completion of a successful SPF_Parse, there should normally be no need to examine or use the Get_Arg$ array. There will be no unaccounted-for operands 'left over' that would be the responsibility of the macro. (If there **were** unaccounted-for operands, these would be categorized as a syntax error. The fact that you don't **have** a syntax error means the parsing completed successfully).

However, since the initially-created "arg" array is not modified by **Spf_Parse**, nothing prevents access using the functions discussed above, if you have that need.

How to access the parsed-out operands

SPF_Parse sorts and categorizes the command line operands into five groups:

Keywords	For all specified Keywords, you can call a single function to return a TRUE / FALSE indication of whether a particular keyword has been entered or not. For example, to see if the Keyword ALL had been specified, the code would be `IF Get_Arg_KW("ALL") then ...` When keywords are part of a mutually exclusive group (like

	WORD, PREFIX, SUFFIX) you may call a function to return **which** of the keywords were specified. The code for this would be
	`IF Get_Arg_KWGroup$("list-name")`
	`then ...`
	
	This requires the creation of list-name to be used to refer to the group of keywords when specifying the keyword list. This is described in the details for SPF_Parse below.
	
	Use of the list-name and Get_Arg_KWGroup$ function does **not** prevent use of Get_Arg_KW for a specific keyword if you desire.
	
	(Just to be clear, **all keywords are optional**. It would not really make sense to say that a keyword was mandatory every time you used a macro.)

Line References	Your macro may specify that it requires any number of Line References, there is no limit. When retrieving Line References, they are returned by relative position, left to right. Their exact operand location, and whether there are intervening operands of other types is immaterial. You request the first Line Reference, the second Line Reference, etc. The code for this would be `start-line = Get_Arg_LRef$(n)` where 'n' specifies the required relative Line Reference number.
Tag Operands	Your macro may specify that it requires any number of Tag Operands, there is no limit. When retrieving Tag Operands, they are returned by relative position, left to right. Their exact operand location, and whether there are intervening operands of other types is immaterial. You request the first Tag Operand, the second Tag Operand, etc. The code for this would be `tag-name = Get_Arg_Tag$(n)` where 'n' specifies the required relativeTag Operand number.
Text Literals	Text operands are quoted strings or un-quoted strings which are not keywords nor not solely numeric. i.e. like the search and change strings of a CHANGE command. Your macro may specify that it requires any number of Text Literals, there is no limit. When retrieving Text Literals, they are returned by relative position, left to right. Their exact operand location, and whether there are intervening operands of other types is immaterial. You request the first Text Literal, the second Text literal, etc. The code for this would be `srch-string = Get_Arg_TextLit$(n)` where 'n' specifies the required relative Text Literal number.

Numeric Literals	Numeric operands are un-quoted strings which are solely numeric digits. i.e. like the column range operands of a FIND command. Your macro may specify that it requires any number of Numeric Literals, there is no limit. When retrieving Numeric Literals, they are returned by relative position, left to right. Their exact operand location, and whether there are intervening operands of other types is immaterial. You request the first Numeric Literal, the second Numeric literal, etc. The code for this would be `start-col = Get_Arg_NumLit$(n)` where 'n' specifies the required relative Numeric Literal number.

How to specify the macro command syntax

The SPF_Parse function may appear complicated at first, but don't let that hold you back from trying it. Let's have a look:

```
RC = SPF_Parse(TxtLit-number, NumLit-number, LinRef-
number, Tag-number, _
               [ keyword-set, ]
               [ keyword-set, ]
               ...
               )
```

What do we have? Four numeric values, and then a series of optional keyword definitions. Let's see the details.

The first four operands specify how many of each of the non-keyword operands your macro will allow. For example, if your macro operands consists solely of two line reference operands, the coding is simply:

```
RC = SPF_Parse(0, 0, 2, 0)
```

which is pretty straight-forward.

If your macro added an optional "ALL" keyword, the coding becomes:

```
RC = SPF_Parse(0, 0, 2, 0, "ALL")
```

again, not unduly complicated.

How to specify a variable number of values

Many operands are optional. In our example above, the line references may be optional if the macro also allows you to mark the lines with a CC / CC line block. But if we code just the number **2** as the Line Reference operand with nothing else, that means the operands are mandatory, and there must be exactly 2 of them. To allow a varying number of arguments, it is handled by adding the ARG_VAR as a "flag" to the count value, so that this parameter to the SPF_Parse call is changed from **2** to **2+ARG_VAR**.. The ARG_VAR option will allow from 0 (zero) to the number specified to be entered.

So if the TxtLit-number operand is coded as **2+ARG_VAR** it means there can be from 0 to 2 Text Literals.

The flag name ARG_VAR, like the ARG_OPT and ARG_DEF flags discussed next, are predefined by SPFLite.

How to specify optional values

Some macro parameters may be used in an optional, "all or nothing" manner. This can be handled by using the ARG_OPT validation flag. When ARG_OPT is added to a count value, it indicates that for this type of parameter the number of supplied values must either be zero or the exact count specified.

For example, if the number of line references is specified as `2+ARG_OPT` it indicates that there must be either **no** line references or **exactly two** line references.

You cannot combine the ARG_OPT flag and the ARG_VAR flag, since the meanings conflict. For instance, if a count value is specified as 2+ARG_VAR+ARG_OPT, the ARG_VAR implies that having one argument is valid, but ARG_OPT implies that it isn't, and these cannot both be true. If you attempt to use the flags this way, the SPF_Parse function will fail with a nonzero RC.

How to validate tag and line references

When Tags or Line References are used as operands, you may wish these entered values to be validated to ensure they correctly point at valid data lines. This can be handled by using the ARG_DEF validation option. When this is added to a value, it indicates that for this type of parameter data line reference must be valid. For example, if the number of line references ia entered as `2 + ARG_DEF` it indicates that there must be **two valid** line references.

How to specify keywords

There are two basic types of keywords, the simple "I'm here / I'm not here" type of keyword (like the RAW operand of CUT), and the mutually exclusive list of keywords (like ON / OFF, or WORD / PREFIX / SUFFIX)

For the simple type, the keyword-set operand is just the keyword itself, couldn't be simpler. "ALL" or "TRUNC" etc.

For the mutually exclusive type, the various keywords are just entered, separated by commas. e.g. for an ON/OFF pair, the keyword-set would be coded as `"ON,OFF"`. For the WORD / PREFIX / SUFFIX example it would be `"WORD,PREFIX,SUFFIX"`

When a keyword can be specified in various alias values, the normal single keyword value in the list is replaced by the list of aliases, enclosed in parentheses. As an example, a macro might accept either DELETE or PURGE as operands. These would normally be coded as `"DELETE,PURGE"`. But what if you wanted to accept DEL as an alternate for DELETE and PUR as an alternate for PURGE. It would then look like `"(DEL,DELETE), (PUR,PURGE)"`

Almost done now, just one more wrinkle for keywords. In the above examples, any of the keywords can be tested for using the Get_Arg_KW function, but what if, for the WORD/PREFIX/SUFFIX example, you wanted to ask "Which one was entered" without having to test each one individually? This is possible, but you have to assign a 'list-name' to refer to that specific set of keywords. Then you can ask for which KW in the set was entered by using the Get_Arg_KWGroup$(list-name).

So where does the list-name go? It is inserted at the beginning of the list, followed by a colon (**:**). Our example here, using the list-name WTYPE, would be coded as:

```
"WTYPE:WORD,PREFIX,SUFFIX".
```

Asking for 'which one was entered' would be done by

```
wtype = Get_Arg_KWGroup$("WTYPE")
```

Example: Putting SPF_Parse all together

A picture is worth a thousand words, so lets do an example. Here's a reasonably complex imaginary macro command whose syntax is similar to FIND and looks like:

```
MACNAME search-str
        [ start-col [ end-col] ]
        [ start-line [ end-line] ]
        [ WORD | PREFIX | SUFFIX ]
        [ RED | GREEN | BLUE | YELLOW ]
        [ ALL ]
```

Lets code the SPF_Parse request to parse this command:

```
IF SPF_Parse(1, 2 + ARG_VAR, 2 + ARG_VAR + ARG_DEF, 0, _
             "WTYPE:WORD,PREFIX,SUFFIX", _
             "COLOR:RED,GREEN,BLUE,YELLOW", _
             "ALL") then
   Halt(FAIL, Get_Msg$)
END IF
```

- The first operand **1** says there is 1 mandatory text literal (the search-str in the syntax diagram)
- The second operand `2 + ARG_VAR` says there may be 0 to 2 optional numeric literals (the start-col and end-col in the syntax diagram)
- The third operand `2 + ARG_VAR + ARG_DEF` says there are from 0 to 2 optional line references (the start-line and end-line in the syntax diagram and that if entered, the references should be validated as correct line references)
- The fourth operand **0** says there are no allowable Tag operands.
- The `"WTYPE:WORD,PREFIX,SUFFIX"` operand specifies 3 mutually exclusive keywords, which can be referenced as a group using the list-name value of `WTYPE.`
- The `"COLOR:RED,GREEN,BLUE,YELLOW"` operand specifies 4 mutually exclusive keywords, which can be referenced as a group using the list-name value of `COLOR.`
- The `"ALL"` operand specifies the simple keyword ALL which may be present or not present.
- The entire SPF_Parse function is coded as part of an IF ... THEN statement. Since SPF_Parse returns a non-zero value (8) if any parsing errors occur, it is usually simplest to test this way and use an SPF_SetMsg to issue the error message text provided by SPF_Parse. This text is obtained using a standard Get_ErrMsg$ function.
- Assuming SPF_Parse returns with no error status, the macro can continue at this point to use the parsed values, or to perhaps perform additional validation if required by the circumstances of the macro's function.

A simple macro demo in all three command operand modes

Here is a simple macro coded in the three different modes to show the differences. The syntax for this example is:

```
DEMOFIND srch-string [ ALL ]
```

Note: All examples below should probably use SPF_Quote$ to properly process the srch-string, but this has been left out to simplify the demonstration code.

Basic Operand Access

```
' DEMOFIND.MACRO
DIM OpAll as STRING
IF Get_Arg_Count < 1 then Halt(FAIL, "Missing search
string")
OpAll = ucase$(Get_Arg$(2))
If OpAll <> "" and OpAll <> "ALL" then Halt(FAIL,
"Unknown operand: " + OpAll))
SPF_CMD("FIND " + Get_Arg$(1) + " " + OpAll)
HALT
```

SUB Operand Access

```
' DEMOFIND.MACRO
SUB DEMOFIND(srchstr as string, OpAll as STRING)
IF srchstr = "" then Halt(FAIL, "Missing search string")
If OpAll <> "" and OpAll <> "ALL" then Halt(FAIL,
"Unknown operand: " + OpAll))
SPF_CMD("FIND " + srchstr + " " + OpAll)
Halt
END SUB
```

Full Parse Access

```
' DEMOFIND.MACRO
if SPF_Parse(1, 0, 0, 0, "ALL") then Halt(FAIL,
Get_Msg$")
SPF_CMD("FIND " + Get_Arg_TextLit$(1) +
iif$(Get_KW("ALL"), " ALL", "")
HALT
```

Macros for fun and profit

A brief overview of macros

We have (somewhat whimsically) entitled this section, *Macros for fun and profit*. You might be persuaded that you could profit from using macros, but maybe you think you won't have much fun writing them. Let's see what we can do to change your mind !

Users sometimes feel that the whole topic of macros is too complicated, and they tend to steer clear of them, without even trying to see what they could accomplish by taking advantage of their capabilities.

Are SPFLite macros really that complicated? Well, you could certainly **make** them complicated if you were inclined to - but then that would be **your** doing, and not SPFLite's fault!

There's actually not that much that is inherently "hard" about these macros, once you familiarize yourself with the **thinBasic** syntax and features. It's really the same as with the rest of SPFLite itself, which has a large array of primary and line commands and keyboard functions at your disposal. That doesn't necessarily make it "hard". It just means you have a lot of tools that you **could** use if you needed them. The specific commands you use, and how you use them, are driven by your requirements, which in turn are driven by the nature of the data you are creating or modifying.

The same is true with macros. A given macro is really only as complicated as the editing task you are trying to carry out. A trivial task can be done with a trivial macro, but a very complex task calls for an involved macro.

> That's only fair. If you wrote some external script in Perl, Rexx or C++ to do a complex editing process, **that** would be involved, too. This is computation - not magic. You can't get something for nothing.

So, the main point is: *don't panic*. Most SPFLite users that create macros will probably only need relatively short scripts. But, the tools and features are there to do really cool and complicated things - **if** that's what you need and want to do.

Let's try and show you how straightforward a macro can be. The best way to do that is with an example, so let's choose a simple one that you should find easy to understand.

Example: The ONLY macro

The ONLY macro implements a two-step SPFLite command sequence:

```
EXCLUDE ALL
FIND ALL string
```

This changes the display to show ONLY those text lines containing a specified *string.*

> In the "good old days" of SPFLite 1.0, as well as in other editors like ISPF, early Tritus SPF, and SPF/SE, this is the only way you **could** do that. SPFLite presently allows

you perform this function directly using the built-in command NEXCLUDE. But, bear with us - imagine that SPFLite **didn't** have this command built in, but you needed to do it anyway. After all, eventually you **will** find some task you need to do that SPFLite doesn't do directly, and then you really will need a macro.

What we need is to create a macro which will perform these two commands after we enter the command **ONLY *string*.**

Why can't we just enter the commands themselves directly, like we could do in those "good old days" when SPFLite only supported simple **.SPM** macros? It's because these programmable macros are not merely "dumb" command lines, but are full-fledged programming statements written in the **thinBasic** language. **Everything** you write in an SPFLite programmable macro must conform to the syntax and language rules of **thinBasic**.

Invoking SPFLite commands within a macro is done by calling a function designed for this purpose, the **SPF_Cmd** function, which is used like this:

```
SPF_Cmd("some SPFLite primary command")
```

The command you specify can be any valid thinBasic string expression, but for now we'll start with simple string literals.

By the way, in the example above, "some SPFLite **primary** command" means that you can't (directly) enter **line** commands using this function. However, you **can** use the **LINE** primary command to achieve the same thing. See the LINE primary command in the main Help documentation for more information.

So, what we want to create is a macro script containing the lines

```
SPF_Cmd("EXCLUDE ALL")
SPF_Cmd("FIND ALL XXX")
```

Let's do that, and create the file **ONLY.MACRO** in the SPFLite \MACROS folder.

FYI, if you're going to get in the business of creating and editing macros frequently, you may find it convenient to set up a FILELIST so you can see all of your macros in one place in the File Manager. See the FILELIST documentation in the main Help file for more information, but basically, simply create a file named MACROS.FILELIST containing the following two lines. Store in the SPFLite data directory.

For XP it would be:

```
FilePath: C:\Documents and Settings\username\My
Documents\SPFLite\MACROS
FileMask: *.*
```

and for Win 7/8:

```
FilePath: C:\Users\username\My
Documents\SPFLite\MACROS
FileMask: *.*
```

However, SPFLite has one additional requirement when creating a macro. The first line of it must be a macro prototype (or, *header*) statement, which takes the form of a thinBasic comment line in a special format. In our example, it requires little more than the name, and

our macro now looks like this:

```
' ONLY.MACRO
SPF_Cmd("EXCLUDE ALL")
SPF_Cmd("FIND ALL XXX")
```

That wasn't so bad, was it?

Our macro is complete ... except that right now, it always does a FIND command for a literal string of "XXX". That "XXX" was just a place-holder, while we were busy trying to explain things and setting up the macro. But, we originally wanted to do a FIND operation that looked like **`FIND ALL string`**, where *string* is supplied on the primary command line - remember? Somehow, we need to create this command dynamically, so it includes **your** string, and not just XXX all the time.

To do that, we need to have the SPF_Cmd function accept a *string expression*, rather than a *string literal.* So, let's make a simple change to our macro to do just that:

```
' ONLY.MACRO
SPF_Cmd("EXCLUDE ALL")
SPF_Cmd("FIND ALL " + Get_Arg$(1))
```

Here, the string literal **`"FIND ALL "`** and the string returned by the function **`Get_Arg$(1)`** are concatenated together to form a single string value. The **`Get_Arg$(n)`** function returns the string contents of the specified argument number. The + plus sign is the string concatenation operator (you can also use & ampersand if you like; they both mean the same thing).

> Remember to put a blank after the "FIND ALL " string, like we did here. Otherwise, the ALL and the string you got back from Get_Arg$(1) would get "run together". If you did that, a macro command like ONLY ABC would cause the macro to create a command string expression like **`FIND ALLABC`** which would either be illegal, or just not what you wanted.

So now, if the command ONLY FRED were issued, **`Get_Arg$(1)`** returns the string **`"FRED"`**, and so the two commands issued would effectively become:

```
EXCLUDE ALL
FIND ALL FRED
```

Exactly what we want!

But, suppose we want to include optional FIND operands like WORD or PREFIX or LAST. This is another simple change, and our final macro will now look like:

```
' ONLY.MACRO
SPF_Cmd("EXCLUDE ALL")
SPF_Cmd("FIND ALL " + Get_Arg$(0))
```

Hmm, what's different? The **`Get_Arg$(0)`** instead of **`Get_Arg$(1)`** is a request for **all** the operands, not just the first. (When you provide two or more macro operands to ONLY, the **`Get_Arg$(0)`** function returns all of them together, so they are separated from each other by a blank.)
So, if the macro command ONLY FRED WORD were issued, the two commands issued

would be:

```
EXCLUDE ALL
FIND ALL FRED WORD
```

Not too shabby, eh?

Enhancing the ONLY macro to work in more than one context

Certainly, that was a fairly simple example, but like most macros, it grows once you start using it and think "It would be nice if ...".

So, what more could we possibly do with this ONLY macro? Well, a lot of the time the ONLY macro would be issued while you were browsing a file and you saw some word of interest within the text. Why should we have to manually type in (or even cut and paste) a word on to the command line, when it's already sitting right there in the text? (There's something to be said for constructive laziness! - RH) Let's change ONLY so that it will use the word that's being pointed to by the cursor.

This is also surprisingly easy. Our macro now becomes:

```
' ONLY.MACRO
SPF_Cmd("EXCLUDE ALL")
SPF_Cmd("FIND ALL " + Get_Curr_Word$)
```

The **`Get_Curr_Word$`** function is designed specifically to address this need. It returns the 'word' that the cursor is sitting on, when the cursor is underneath some string in the data area of the edit window. Now, we can map a keyboard key (let's say, Ctrl-O) to ONLY and then all we need to do is place the cursor anywhere within a word and hit Ctrl-O.

We now have a quite useful macro, and it is still only 3 lines long.

However, we have now lost the ability to actually type in an **`ONLY`** ***`string`*** command, since the macro now requires a word to be present under the cursor. If you are typing a command on the **command** line, there isn't any file data under the cursor, because your cursor is not **in** the data area.
But, we don't want to give up the command-line usage. What we really need is a way to determine the macro's **context** - that is, **how and where the macro is being used**. Then, we tailor the macro to operate differently depending on how it's being used, and then we can use the same macro for more than one purpose.

Managing quoted operands

Before we put this new macro together, there is a little matter of quotes to deal with. If you recall from the basics of SPFLite, you can have quoted string values enclosed in ' single quotes, " double quotes or ` accent quotes, or simple strings can be supplied unquoted.

You also need to be aware that keywords that are reserved elsewhere, like ALL and LAST, are **not** reserved as operands of a macro used as a primary command.

So, if you supply a macro argument that happens to be a keyword of some other SPFLite command, and you then use that string to dynamically create some command, it could create some invalid syntax. For example, without some "fancy footwork", if we tried to say

ONLY FIRST, one of the created commands would be FIND ALL FIRST, and that's not legal.

In your macro, you might be tempted to get around this by always putting quotes around the string you get back from a Get_ARG$ call. But, suppose, as a macro user, you actually placed quotes around a string, like ONLY "FIRST". If your macro added quotes, too, you'd have too many of them. So, if you issued a macro command like ONLY "FIRST" with quotes already around the operand, and the macro logic added quotes, too, you'd end up with a command like FIND ALL ""FIRST"" with improperly-doubled quotes - and that's not what you wanted.

You **could** check to see if the argument already had quotes on it, and tailor your expression accordingly, but that would be a real nuisance to do all the time. Fortunately, you don't have to. We have created a function called SPF_Quote$. What this does is it adds quotes to a string expression, unless it is already quoted, in which case it leaves it as is. It also takes into account whether a string contains inner quotes of any type, so that it wouldn't create a wrongly-quoted string. It does that by choosing one of the three allowable quote types, to avoid having the enclosing quotes be the same as any inner quotes which are literal data values. These actions make the function what we call a "smart quoter".

The example below uses the smart-quoter function to correctly create SPFLite command strings. Now, even if the operands you provide are already quoted or are SPFLite reserved words, the macro will still work properly.

You would need this smart quoter function whether the operand was supplied on the command line or was obtained from a function like Get_Curr_Word$, because we have no way of knowing if the word under the cursor is reserved either.

Managing argument lists

When you use the ONLY macro, and it has multiple arguments, you can "grab" them all with a function. But if you needed to quote the search string AND you had additional arguments, how could you just quote the first one and not the rest of them?

This is going to take a little tinkering ... but we promise, it won't be too hard.

First of all, you need to understand that macro parameter lists are **not** like SPFLite command parameters. For example, a FIND command could say FIND ALL ABC or it could say FIND ABC ALL, and they both mean the same thing, because you can supply operands to most SPFLite commands in any order. SPFLite can tell which operands are reserved words and which are user-supplied values, and it sorts them out based on which command you used.

Although Macros can be coded to handle operands this way, for the purposes of this discussion we will ignore that temporarily and keep things simple. If, following this discussion, you want to learn how to handle operands in that manner, review Full Parse Access.

With that in mind, we will agree to design the ONLY macro so that the *string* operand is always the first one, meaning that you'd always use **`Get_Arg$(1)`** to "grab" it. What about all the *other* operands - if there are any? How do we grab them?

First, we take advantage of the special fact that Get_Arg$ can be called with either one or two arguments of its own. If called with two arguments, you provide a "range" of operands. So, if you wanted arguments 3 through 5, you could get them with **`Get_Arg$(3,5)`**.

> You have to ask for arguments in ascending order. Calling **`Get_Arg$(5,3)`** will return nothing (an empty string). And also trigger a failing RC value and associated error message if you cared to check and retrieve them.

Second, if you read closely in the Function Details section, you'll find there is a function called **`Get_Arg_Count`**. So, assuming you wanted all arguments starting with the second one, you could say **`Get_Arg$(2,Get_Arg_Count)`**. That would work fine, but it's kind of wordy. Since it will be a common task to require "all remaining operands" this way, the function allows the second operand to be specified as **`0`**, with a call like **`Get_Arg$(2,0)`**. When you supply a **`0`**, it is treated the same as if you had used **`Get_Arg_Count`** instead. That makes this a lot shorter and easier to type.

> A call like **`Get_Arg$(2,0)`** is not considered breaking the rule about asking for arguments in ascending order; it's just a short-cut convenience so you don't have to worry so much about exactly how many arguments were present, and so you don't have to type so much.

Finally, if a call to **`Get_Arg$`** asks for more arguments than you supplied when you ran the macro, it will only provide the ones that are there. It's not illegal to ask for "too many".

> Just like the single-argument form **`Get_Arg$(0)`**, when you provide two or more macro operands and use **`Get_Arg$(2,0)`** it returns the requested macro operands so they are separated from each other by a blank.

Putting it all together

So, let's finish the job on our macro and flesh it out, adding a bit of error checking as well.

Note that **`Get_Arg$`** will put spaces *between* the operands it returns, but not before or after. So, in the **`SPF_Cmd`** expressions, we have to separate the various "pieces" of the expression with a blank so that we end up creating a valid SPFLite primary command; otherwise macro argument 1 and 2 would get "run together" and things wouldn't work right.

Notice too that we only quote the first argument, but not the remaining ones. Why not? Because we agreed that the "string" argument would always be the first one (remember our discussion from above?) which means the remaining arguments, if there are any, would only be SPFLite keywords like WORD or LAST, which would never be quoted if you wanted to use them as normal keywords and not as data. So, we don't call SPF_Quote$ for them.

```
' ONLY.MACRO
if Get_Arg$(0) <> "" then
   SPF_Cmd("EXCLUDE ALL")
   SPF_Cmd("FIND ALL " + SPF_Quote$(Get_Arg$(1) + " " +
Get_Arg$(2,0)))
else
   if Get_Curr_Word$ <> "" then
      SPF_Cmd("EXCLUDE ALL")
      SPF_Cmd("FIND ALL " + SPF_Quote$(Get_Curr_Word$))
   else
      Halt(FAIL, "There is no word under the cursor")
   end if
end if
```

Several changes have been made:

- If a command line argument is present, then **that** is what is searched for
- If no command line argument **and** the cursor is on a word, **that** word is searched for
- If no command line argument and the cursor is not on a word, an error message is issued using the **`Set_Msg`** function
- The smart-quoter function **`SPF_Quote$`** ensures that if operand 1 is quoted or is a reserved word, it won't cause an incorrect **`FIND`** command to be created.

Our final, fully fleshed-out macro is still only 12 lines, including the header. That's pretty reasonable, and hopefully not too intimidating, even if you're new to writing macros.

Simplifying the ONLY macro with NEXCLUDE

If you look closely, by providing some extra logic to take the context into consideration, you now have a capability that SPFLite doesn't have on its own - even the most current version. Remember we said the basic functionality of ONLY is already covered by the built-in command NEXCLUDE? Well, what ONLY does **now** is actually **more** powerful than NEXCLUDE. But, that doesn't prevent you from actually using NEXCLUDE here. We can eliminate two lines from the macro by replacing EXCLUDE and FIND ALL with NEXCLUDE ALL.

This is a case where having a good familiarity with all of SPFLite's array of available commands can help you write macros that are shorter and run faster. Here is how the revised ONLY macro would now look:

```
' ONLY.MACRO
'
'
if Get_Arg$(0) <> "" then
  SPF_Cmd("NEXCLUDE ALL " + SPF_Quote$(Get_Arg$(1)) + " "
+ Get_Arg$(2,0))
else
  if Get_Curr_Word$ <> "" then
     SPF_Cmd("NEXCLUDE ALL " +
SPF_Quote$(Get_Curr_Word$))
  else
     Halt(FAIL, "There is no word under the cursor")
  end if
end if
```

Extra credit - Part 1

Did you notice how we determined the 'context' of the macro - how it was being used? Calling **`Get_Arg$(0)`** returns a string with all the arguments provided to the macro. If none were provided, you'd get an empty string.

Suppose your macro is called **AX.MACRO**. Under what circumstances would you get an empty string back from **`Get_Arg$(0)`** when you used this macro?

- You enter **AX** on the command line by itself with nothing else, to run the macro as a primary command, and press Enter.
- You enter **AX** or **AXX** in the sequence number area of one or two lines, and press

Enter. (If you use **AX** as a line-command macro, only one may be specified at a time. If you use **AXX** you must specify it in pairs. Otherwise, SPFLite will detect a usage error.)

Is that the **only** way to determine the context? No, there are a few other ways, too:

- You can call **`Get_Arg_Count`** to find out how many arguments were present; if there weren't any, you get 0
- You can call **`Get_Csr_LPTR`** to find out the line-pointer where the cursor is located; if it's not in the main edit area, you'd get a 0
- You can call **`Get_LNUM(Get_Csr_LPTR))`** to find out the line-number where the cursor is located; if it's not in the main edit area **or** it's on a non-data line like BNDS or TABS, you'd get a **0** (since lines like BNDS and TABS **have** no line number). This is a little more involved test, but it's a "stronger" one.

You can also see whether your macro was called as a primary-command macro, or as a line-command macro. See Primary mode vs. Line mode macros for more information.

Based on your requirements, you'd have to decide how "strict" your test needs to be to ensure it only runs under the right circumstances.

Extra credit - Part 2

Now that we have created the ONLY macro, fine-tuned its behavior, and taken its operating context into account, what do you think:

Is ONLY *a primary-command macro or a line-command macro?*

Yes, it's a trick question - and the answer is, surprisingly, **both**.

The ONLY macro works quite well when invoked as a line-command macro. If you enter ONLY in the sequence area of a data line, then move the cursor over the word you want as the operand, and press Enter, the macro will function correctly. In fact, the ONLY macro name could be in the sequence area of one line, and the cursor could be on an entirely different line, and it would still work.

What happens? First, the Get_Arg$(0) call will return an empty string, because the macro has no arguments, per se. It then finds a value returned from the call to Get_Curr_Word$, and the macro logic proceeds from that point.

No, this isn't likely to be a convenient way to type this, and most users would not do it that way, but unless a macro is carefully written to insist on being run only one way or the other, it may very well meet all the requirements for execution as either a primary-command macro or a line-command macro - even if that isn't what you intended when you wrote it.

Most of the time, you will write a macro with the intent of it only running one way, either as a primary-command macro or a line-command macro, and you may neither know no care that it would work the 'other' way.

If you need to be sure, you can use the Is_Line_Cmd and Is_Primary_Cmd functions. See Primary mode vs. Line mode macros for more information.

SPFLite Interface

The **thinBasic** script interpreter itself has no knowledge of SPFLite whatsoever. It knows and understands Basic syntax, and nothing else.

But, when run from within SPFLite, a collection of SPFLite interface routines are automatically made available as extensions to the **thinBasic** language. These routines provide all the needed tools to support interaction with SPFLite and allow **thinBasic** to be used as a proper scripting language. These functions provide the following abilities:

- Functions which can access a variety of status information about the SPFLite edit session itself and the file being edited.
- Functions which can be used to:
  - Fetch and store data lines in the Edit session
  - Issue Primary and Line commands that affect the Edit session
  - Set macro return codes and messages for the user
  - Set cursor location to be used following the macro execution

Note: You may notice that one of the features of SPFLite that is **not** supported through macros is the invocation of keyboard primitive functions. That is a limitation in the present implementation of macro support in SPFLite. Later versions of SPFLite may or may not add this support.

Interface Structure

Interface Structure

Passing control to your macro

SPFLite determines that a command **is** a macro when a command name is not recognized as a built-in command, but a file having the name of macroname.MACRO is found the in the SPFLite macro directory. This design means that (unlike mainframe ISPF) it is not possible to replace a built-in primary command with a macro.

> It **is possible** to make a macro used as a **primary** command with the same name as built-in **line** command. For example, a primary-command macro named CC.MACRO is valid. However, that could be confusing, and for most people, it will be best if you avoid **any** built-in command names, whether line-mode or command mode. This is especially true in case you write a macro that could be used both as a primary-command macro or as a line-command macro.

When SPFLite detects a macro is being invoked, it triggers the **thinBasic** interpreter and passes it two things: the source of your macro from the macroname.MACRO file and the list of available SPFLite interface functions being made available.

The number of available interface functions is extensive (approaching 100) but don't let this put you off. Just like SPFLite's extensive list of Primary commands, Line commands, and Keyboard primitives, nobody will ever need or use all of them. We're just trying to ensure that we provide as extensive an interface as possible. Nothing is worse than developing a macro and saying "Gee, if there was only a function that would ....". We don't want that to happen to you.

thinBasic parses and analyzes your source code and begins execution of the interpreted code. At this point, the macro source code has still been provided no information whatsoever about SPFLite, or the file being edited.

All information passed between the macro and SPFLite is handled via those custom functions which SPFLite made available when invoking **thinBasic**.

> If you're a compiler geek or something, what SPFLite does is "inject" the SPFLite interface functions into the **thinBasic** namespace, so that you don't have to declare or "include" them. (If you're not a compiler geek, then it's magic - and don't nobody ask no more questions.)

Primary mode vs. Line mode macros

SPFLite allows you to write macros that can be used on the primary command line, or in the line command area. The macro has the same structure either way. You would have to determine how your macro is getting called, by making queries against some SPFLite built-in functions.

NOTE: You cannot have an SPFLite interaction in which you have macros used as line commands, **and** any command on the primary command line (whether built-in SPFLite primary command or a primary-command macro), at the same time.

When you use a macro as primary command, and you have any pending line commands at the same time, these would usually consist of an **A** or **B** and/or a **CC** block, to define the source lines and optional destination line.

When a macro is being used as a primary command

- Is_Primary_Cmd will return TRUE
- Is_Line_Cmd will return FALSE
- Get_Src1_LPTR and Get_Src2_LPTR return non-zero values if you have a line range defined using a **CC** or **MM** block, otherwise these will return zero.
- Get_Src_LCMD$ will return a **C**/**CC** or M/MM line command if present, otherwise an empty (null) string value
- Get_Dest1_LPTR, GET_Dest2_LPTR and Get_Dest_LCMD$ will be set to appropriate values if you have defined a destination line range using **A**, **AA**, **B**, **BB**, **H**, **HH**, **W**, **WW**, **O**, **OO**, **OR** or **ORR** line commands, otherwise these will be zero or null.

When a macro is being used as a line command

- Is_Line_Cmd will return TRUE
- Is_Primary_Cmd will return FALSE
- Get_Src1_LPTR and Get_Src2_LPTR will return non-zero values
- Get_Src_LCMD$ will return the contents of the line command used to invoke the macro. For example, a macro **AX** is invoked in response to a line command of **AX** or **AXX**.
- Get_Dest1_LPTR, GET_Dest2_LPTR and Get_Dest_LCMD$ will be set to appropriate values if you have defined a destination line range using **A**, **AA**, **B**, **BB**, **H**, **HH**, W, **WW**, **O**, **OO**, **OR** or **ORR** line commands in addition to the macro defined source range, otherwise these will be zero or null.

The name of a line-command macro is the name you'd use for a single line. So, **for example**, if you wanted to create a line command called **AX**, your macro would be called **AX.MACRO**. If you want to use this command as a block-mode line-command macro, you can do it any of the normal ways you'd expect with an SPFLite built-in line command:

- Define the beginning and end of the block with **AXX**
- Put **AX** on a line, and follow it with an **n**, **/** or **** modifier
- Create a **SET** name of the form **SET MACROMODE.AX = BLOCK** if you only want to use **AX** as a block-mode command

Passing information between macros and SPFLite

To accomplish its desired actions, your macro must use the provided SPFLite functions to both fetch and to store data back and forth between SPFLite's data areas and the macro's own variable areas.

There are four main types of provided functions:

Get_xxxx The **Get_xxxx** function calls will return the value of the requested SPFLite data item to the macro. These **Get_xxxx** function calls can effectively be used as if they are variables. **Get_xxxx** functions ending in a $ sign (like **Get_RECFM$**) return character strings, those without a $ sign return numeric values. (This is a standard BASIC naming convention.)

Set_xxxx The **Set_xxxx** function calls are used to alter the value of a specified SPFLite data item using the provided macro value.

Is_xxxx The IS_XXXX function calls are used to test the basic characteristics of the environment, of data items, and to test the line type of a specified line within the Edit session. SPFLite provides access to all of the different line types (including "special" ones), not just text data lines. Access is provided to NOTE lines, MASK lines, etc. The **Is_xxxx** function calls will return TRUE if the line **is** the requested type, and FALSE if it is any other type. For example, to test if a specific line is a normal text data line you would code **IF Is_DATA(line-ptr) THEN ....**

Because special lines are not data lines, they only can have a line pointer to them - but never a line number; whereas a data line can have both a line pointer and a line number.

SPF_xxxx The SPF_XXXX function calls are used to request specific SPFLite services of a more complex nature, or ones which cannot be strictly described as a GET or SET operation. **SPF_CMD** for example is used to pass a primary command to SPFLite for execution.

Each of these function calls is described in the Function Overview and Function Details sections.

Identifying and referencing lines

To inspect or modify the lines in an Edit file, you must be able to identify which line of the file you are referring to when using the supplied SPFLite interface functions. This seems simple enough - couldn't we just use the visible line number itself to reference a line? **No.**

We need to be able to access all the different line types within an edit session, not just the text data lines. An edit session may have "special" lines such as **NOTE**, **BNDS**, **COLS**, **TABS**, etc. that we might want access, and these lines don't have line numbers.

The solution is to use something called a **line pointer** as the key to accessing the data. As long as you keep in mind that a line pointer can sometimes point to non-text lines, most of the time you can think of it **as** a line number. To simplify things, most (but not all) of the custom interface functions work with line pointer parameters. We have been careful with naming conventions so you'll know what functions take what kind of arguments, and there

are ways of converting between line numbers and line pointers.

The difference between a line **number** and a line **pointer** becomes an issue when calling the SPF_CMD function, and when interfacing with the external user (yourself) since the user never sees and is not aware of the line pointer values. So, line number values are provided by the user, and you may want to pass back line number values to the user, for example, in messages. To do this, you will need the conversion functions.

Conversion between a line **number** and a line **pointer** is a simple process. There are two companion functions whose sole purpose is to convert line pointers to line numbers, and line numbers to line pointers. A side benefit of the line number to line pointer conversion function is that it incorporates automatic support for line labels as well as line numbers. In that sense, it doesn't just accept **line numbers** but more generic **line references**, of which line numbers are one type.

The two conversion functions are `Get_LNum` and `Get_LPtr.`

But the most powerful reason for using line **pointers** over line **numbers** is that line **pointers** are amenable to being adjusted arithmetically while line **numbers** are not. Why so?

Line numbers, especially if using the Quick Renum option, are not guaranteed to be consecutive. Line-Number + 1 simply cannot be guaranteed to point at the next line number. (e.g. lines may be numbered in increments of 10, or 5, or 1; there is no way of knowing)

Line pointers however, will always point at a valid line of some type. (lets ignore the Top and Bottom boundary conditions for now). So you can always do a line-pointer + 1 calculation and you will always end up with a valid line pointer. This is the main reason that the macro interface functions are set up to use line **pointers** almost exclusively. It's just easier in nearly all respects.

Understanding Line Pointers and Line Numbers

A line **number** is the numeric value of the sequence number you see displayed on a data line. So, if you see a data line with a sequence number **000123** then the line number is **123**. A valid line number will never be zero.

A line **pointer** may be thought of as a numeric index to every visible line (special or ordinary) of the entire edit file, starting with 1 for the Top of Data line, and incremented by 1 for each subsequent line (special or ordinary) until the Bottom of Data line is reached.

> That is an important fact to remember: The Top of Data line will always have the known Line Pointer value of 1. You can use that fact when you are scanning the lines in your file, and you need a Line Pointer value as a starting point.

This means that you can scan a range of lines (special or ordinary) using a FOR/NEXT loop, checking the "type" of each line with an **Is_xxxx** function, incrementing the line pointer by 1 each time, and taking appropriate action as needed. The most common such test will be checking for **Is_Data**, passing the function a line pointer argument.

Like line numbers, valid line pointers will never be zero. However, whatever **other** values might be taken on by a line pointer, you should always let SPFLite inform you, and **not** make assumptions about the value you may get. Future releases of SPFLite could change those assumptions, and if you were relying on a particular implementation, your macro might not work right in a future release. Don't try to be too 'tricky' with line pointers; just use them as

the documentation and examples show you.

> What a line pointer definitely is **not** is a true memory pointer to some dynamically-allocated storage area within **thinBasic**. Don't even **think** of trying to use it that way.

When you have the Fast Renumber General option (within SPFLite Global Options) enabled, it may affect how the conversion functions handle converting a line pointer to a line number. Again, use the functions for what they are, and don't try to be too tricky with them or make assumptions about what the value is going to be.

The SPF_CMD function and Line Numbers vs. Line Pointers

The SPF_CMD function is used to issue SPFLite primary commands. The command string you pass to the function looks just like one you would type in yourself on the primary command line. There are a few minor differences:

- SPF_CMD only takes one command. You cannot use a command separator character (like a **;** semicolon) to perform multiple commands.
- The special ! notation that erases the primary command area prior to executing the command is not supported. That feature is primarily in place to assist key-mapped commands to function properly in case there were any "left over" commands or operands in the primary command area, an issue that doesn't apply to a macro.
- Presently, you cannot issue a macro call from a macro. So, in **AX.MACRO**, you cannot issue a call to **SPF_CMD("BX")** for instance.

When you create a command string for SPF_CMD using string constants and expressions, and the command contains a line reference, you must determine if the reference is a line **number** or a line **pointer**. Most macros will use line **pointers** almost exclusively.

If you have a Line Number value, you create the command string so that it appears the same as you would type it in on the primary command line. Recall that SPFLite allows what are called "temporary line labels" or "line number pseudo-labels", which are line numbers with a preceding dot.

For example, if you wanted to change ABCD to WXYZ on line 4 of the edit file, you could issue

```
SPF_CMD ("C ABCD WXZY .4 ALL")
```

However, if you have a Line Pointer value, you can't (directly) use such notation. For example, the line with ABCD might have a line **number** of 4, but a line **pointer** of 7. You may have gotten this value of 7 from a function call like GET_FIND_LPTR. Now, if you created a command like this:

```
SPF_CMD ("C ABCD WXZY ." + TSTR$(GET_FIND_LPTR) + " ALL")
```

it would evaluate to this:

```
SPF_CMD ("C ABCD WXZY .7 ALL")
```

but **that won't work**, because it's referring to the wrong line. Since most SPFLite macro interface functions that deal with lines use line **pointers**, somehow we have to make SPF_CMD understand what we mean.

There's two ways to do this. First, you can use the conversion function GET_LNUM. In our example, that would convert the 7 into a 4:

```
SPF_CMD ("C ABCD WXZY ." + TSTR$(GET_LNUM
(GET_FIND_LPTR)) + " ALL")
```

Second, SPF_CMD allows an alternative notation. If you use an **!** exclamation point **instead of a dot**, SPF_CMD will understand the value to be for a line **pointer** rather than a line **number**. When you do this, SPF_CMD will internally convert the line pointer for you into a line number.

In our example, you would do this:

```
SPF_CMD ("C ABCD WXZY !" + TSTR$(GET_FIND_LPTR) + " ALL")
```

and it would evaluate to this:

```
SPF_CMD ("C ABCD WXZY !7 ALL")
```

The SPF_CMD function would then see the exclamation point, take the number 7, internally call GET_LNUM itself to obtain the line number 4, and finally treat the command as if you issued:

```
SPF_CMD ("C ABCD WXZY .4 ALL")
```

Some points to keep in mind if you use the exclamation point notation:

- This notation **only** works for the **SPF_CMD** function, and no where else.
- This notation only applies to macros. You cannot issue this syntax on the regular primary command line.
- The exclamation point must be preceded by a blank, and the number after it must be followed by a blank or the end of the string
- Exclamation points are only recognized if not in SPFLite quotes.

So, a function call of `SPF_CMD ("C ABCD '!7' ALL")` would have nothing to do with this notation; the `!7` is just an ordinary string.

Line Pointers or Line Numbers: Which is Better?

Since there are two possible ways to refer to a line, which way is "better" ? To (probably) no one's surprise, the answer is, "it depends".

A more appropriate question is, better for **what**? It depends on what you're doing in your macro.

> This question is similar to asking which is better, Celsius or Fahrenheit? It depends where you live. (Since George lives in Canada, and I live in the States, we each have our own preferences. Basically, whether it's 100 C or 212 F, you know you're in hot water ! - RH)

There are several factors to consider when making this decision:

- The SPFLite macro interface functions take line **pointers**. So, this is the form you will use most often, and if line pointers are what you have, this will be the most efficient

method. For the most common cases, you will get a line number (or label) as a macro argument, convert it to a line pointer using Get_LPTR, and use that value from then on.

- The SPF_CMD function will accept either notation (as discussed above).
- If you have one form, but need another, you can call a conversion function to get the desired value.
- Line pointers may be somewhat transient, depending on what you're doing. For example, suppose you have several non-data lines in your file. These could be =PROF> lines, =NOTE> lines, =TABS> lines, etc. You may also have excluded-line placeholders present. If you obtain a line **pointer**, and then these special lines disappear as a result of issuing a RESET command or by direct deletion, the line pointer may no longer be valid. A line **number** will remain valid across a RESET command.
- It is important to understand that when a line **pointer** becomes "invalid" it does **not** mean you can't use it. For example, a line pointer might contain 5, meaning it's the fifth line from the beginning, counting the Top of Data as line-pointer line 1, but it would be the fourth data line. Now, if you inserted a =NOTE> or =TABS> line somewhere between the Top of Data and data line 4, or removed one already there, a line pointer of 5 won't point to data line number 4 any more. **But that's won't stop you from using it.** You can still pass a value of 5 to any of the macro interface functions that take line pointer arguments - the function won't stop you or claim the number 5 was "invalid". However, if you do that, the 5 wouldn't refer to data line 4 any more, but to some **other** line. That would almost certainly **not** be what you wanted.

The point is, line pointers and line numbers **both** have their place. You need to know what you're doing, and use them for what they are good for.

For line pointers, you must understand the rules that affect the validity of a line pointer reference, and stay within those rules. The main rule is that you must not "rock the boat" by doing anything that will change the relative line-pointer positions of data lines. Things that cause that are inserting and deleting (other) data lines, inserting or deleting special lines, and excluding or unexcluding lines. Special and excluded lines are affected by the **RESET** command. Provided you don't do anything to invalidate a pointer, you can continue to use it without problems. (They are not "fragile" and won't "break" by themselves.)

If you are forced to "rock the boat" so to speak, then you may have to recalculate new line pointers. Suppose you needed to do a **RESET** command that would impact the line pointer to a data line. You could "save" the pointer by (a) converting it to a line number, (b) issuing the **RESET** command, and finally (c) converting the line number back to a line pointer. You could also put a label on the line, and retrieve the label's line pointer later on.

As long as you keep these rules in mind and live within them, everything will work just fine.

Understanding Source and Destination Line Ranges

A macro will often need to refer to a line range. The way in which a macro obtains line-range information depends on whether is was launched as a line-command macro or as a primary-command macro.

Primary command macros may include line-range information as operands, such as line

labels (including pseudo-label line numbers) and line tags. However, SPFLite will **not** interpret such operands. It merely passes them along to the macro, which must retrieve them with the Get_Arg$ function and then interpret them as it sees fit. For line label operands, you would use the Get_LPTR function to convert them to a Line Pointer value, which is what most line-referencing macros use.

This means that if you create a macro called **ABC.MACRO**, you **can** issue a primary macro command like **ABC .AA.BB**, but **you** would have to write **thinBasic** code to interpret arguments 1 and 2 so that you could utilize the provided labels.

> This may seem like an inconvenience (and to a certain extent, it is) but this design allows you complete control over how you would use these labels. For example, you might wish to write a macro that took 3 or even 4 labels. Standard SPFLite primary command processing would treat that as a syntax error, but your macro code could do something to address your unique requirements. If SPFLite recognized a standard line-range operand on macros (and nothing else), it would simplify some tasks, but would hold you back from doing other things. For now, the design seems like a good compromise.

While you don't have direct support for line labels on primary macro commands, you **do** have the ability to query other line commands that may be pending when the macro is launched. These are **source line ranges** and **destination line ranges**. Specifying source and destination line ranges is optional when launching a macro.

> Technically, a line-command macro always defines its own source line range by its very presence in the line command area, so when you use a line-command macro, you always have an implicit source line range; it isn't optional.

For a primary-command macro, a **source** line range can be a **CC** or **MM** block. Such blocks can be single lines, a range of lines defined by a C or M followed by **n** or a **/** or **** modifier, or a pair of **CC** or **MM** commands to define a block.

For a line-command macro, the macro name itself appears on one or two data lines to define the **source** block. As discussed elsewhere, SPFLite automatically detects the single-line form vs. the block-mode form of a line command. For example, if you write a macro called AX, then you could do any of the following to define a source block: **AX**, **AX*n***, **AX/**, **AX**, **AXX** or **AXAX**.

For both primary-command and line-command macros, you can specify a **destination line range** using any of the following line commands:

> A, **AA**, B, **BB**, **H**, **HH**, **W**, **WW**, **O**, **OO**, **OR** and **ORR**.

The non-block forms **A**, **B**, **H**, **W**, **O** and **OR** can take the usual **n**, **/** and **** modifiers. You can also specify a trailing **+** or **-** modifier.

SPFLite does not act upon source or destination line command operands when used by a macro. That means that **CC** will **not** copy lines, MM will **not** move lines, **OO** will **not** overlay lines, **AA** will **not** intersperse copied data every few lines, HH will **not** replace lines, and so on.

But if that's true, what **does** it do? What **good** is it to use any of these things? These source and destination line ranges pass along information to the macro, which you can query and act on yourself. **You** can decide which of any of these line commands you want to recognize in your macro, and decide what they will do if used. What SPFLite does is (a)

ensures that only these line commands can coexist with your macro's execution, and no other ones, (b) makes sure that they are in the proper format, and properly matched if used in pairs, and (c) will extract any addition information like **n** values and **+** or **-** and K modifiers that may be present, so you can look at them and act accordingly.

For the source line range, you have the following functions available:

Get_Src_LCmd$	Get a string containing the name of the source line command. For line-command macros, this is the macro name (possibly a plural/block form) of the macro name. For primary-command macros, this may be a **C**/**CC** or **M**/**MM** command, or may be an empty (null) string if no **C**/**CC**/**M**/**MM** was used with a primary-command macro.
Get_Src1_Lptr	Get the line pointer of the first (or only) line in the source line range, or 0 if not applicable.
Get_Src2_Lptr	Get the line pointer of the last (or only) line in the source line range, or 0 if not applicable.
Get_Src_Op	Get optional **n** value on the source line range. For example, if you have a source line range of **C5**, **Get_Src_Op** will return the number **5**. The function returns 0 when no explicit command operand number is specified. For instance, if you define a source block such as **CC** which happens to be 5 lines long, the function will still return 0, because the number 5 was not actually specified on the CC block.
Get_Src_LMOD$	Gets the **+** or **-** (post-unexclude or post-exclude) modifier and **&** (Retain) modifier if one is present on the source line range. The returned value is always two characters long. Position 1 will contain **+** or **-** or blank, and position 2 will contain a **R** or blank. The **/** and **** line command modifiers are processed, if present, and this gets reflected in first and last LPTR values for the line range, but the **/** and **** codes themselves are not returned as LMOD values.

For the destination line range, you have the following functions available:

Get_Dest_LCmd$	Get a string containing the name of the destination line command. This may be an **A**/**AA**, **B**/**BB**, **H**/**HH**, **W**/**WW**, **O**/**OO**, or **OR**/**ORR** command, or may be an empty (null) string if no destination line range was specified.
Get_Dest1_Lptr	Get the line pointer of the first (or only) line in the destination line range, or 0 if not applicable.
Get_Dest2_Lptr	Get the line pointer of the last (or only) line in the destination line range, or 0 if not applicable.
Get_Dest_Op	Get optional **n** value on the destination line range. For example, if you have a destination line range of

H5, **Get_Dest_Op** will return the number **5**. The function returns 0 when no explicit command operand number is specified. For instance, if you define a destination block such as **HH** which happens to be 5 lines long, the function will still return 0, because the number 5 was not actually specified on the **HH** block.

Get_Dest_LMOD$ Gets the **+** or **-** (post-unexclude or post-exclude) modifier and **&** (Retain) modifier if one is present on the destination line range. The returned value is always two characters long. Position 1 will contain **+** or **-** or blank, and position 2 will contain a **R** or blank.

The **/** and **** line command modifiers are processed, if present, and this gets reflected in first and last LPTR values for the line range, but the **/** and **** codes themselves are not returned as LMOD values.

Note that when you use line macros in block form, or use destinations such as OR, you can define a 'block' by repeating the last character. (This is the same rule that ISPF uses, and we follow the ISPF rules on this.) So, the block form of a macro **AX** is **AXX**. When you ask for the line command name of the source or destination using Get_Src_LCmd$ or Get_Dest_Lcmd$, you get the actual command used, which in this case is **AXX**.

Function Overview

This section provides an overview of the various interface functions and their usage. For full details of syntax, allowed parameters, return values etc. click on the Function Name on the left to switch to the detailed description for the function.

Understanding SPFLite status values

A macro you run from your edit session, by placing a macro name on the primary command line, or on the sequence area of a data line, will return a **status** when it completes. A **status** consists of two values: a numeric **Return Code** value (often referred to as **RC**), and a **Message** string. This status is displayed on the upper-right portion of the edit screen, in the form of the Message string, and an optional Warning or Error message prefix in case a non-zero Return Code has been set.

When you call an SPFLite-supplied interface function (such as **Get_Line$**), that function will **also** return a status. You may retrieve this status using the functions **Get_RC** and **Get_Msg$**.

There are thus **two** sets of status information. The status returned to **you** (as an edit user that executed a macro command) that is visible on the edit screen as a message is called the **top-level status.** The status returned to a user-defined macro by a call to an SPFLite interface function is called the **function-level status.**

It it important to understand that the top-level status and the function-level status are **completely different and separate values.**

This means that if you issue a call to **Set_Msg(my_RC, my_Message)** it will change the top-level status values for the RC and Message, but will **not** change the function-level values that are returned by **Get_RC** and **Get_Msg$**. Another way to look at this is that the function-level status is "lower" than the top-level status, and that status values can go "up" (via Set_Msg) but cannot go "down".

Example:

```
DIM rc as NUMBER
DIM msg as STRING
' ... some macro code
Set_Msg (8, "user defined error occurred")
rc = Get_RC
msg = Get_Msg$
' at this point, rc = 0, and msg = "", not the values
set by Set_Msg
```

The other key point to keep in mind is that SPFLite interface functions do not "communicate" the function-level RC and Message values between themselves. So, if one SPFLite function set an RC value of 8, the next SPFLite function you call will know nothing about that; it won't "look" at the RC=8 value set by the prior function call. Each function "starts over from scratch" in terms of setting the function-level status values, by assuming each time that the RC will be 0 and the Message will be an empty (null) string, unless conditions require them to

be changed.

Return codes and returned messages from SPF-supplied functions

All SPF-supplied functions set the function-level Return Code (RC) variable and the standard Message text appropriately based on the success of the function. **This is done in addition to providing the return value from the function directly.** In a few cases, the value returned by the function will be the RC value itself, but most functions will return values other than the RC.

> Functions built into **thinBasic** do **not** set the RC and message values. That only applies to functions we provide as part of the SPFLite macro interface. So, if you call a **thinBasic** function like **MID$** to get a substring of a value, that does **not** change the RC or message.

The function-level status values can be obtained with the **`Get_RC`** and **`Get_Msg$`** functions. A successful return will usually result in `RC=0` and a message text of `""` (null). However, some successful functions may result in a RC=0 but with a completed message text string. For example, invoking an **`SPF_Cmd("QUERY EOL")`** function may result in a RC=0, but a **`Get_Msg$`** return string of **`"EOL set to CRLF"`**.

When a function returns a value which is not important to you, you can call it directly without assigning its returned value to a variable. For example, you can call

```
RC = SPF_Exec("MyProgram")
```

but if the return code isn't important, you can shorten this to:

```
SPF_Exec("MyProgram")
```

Return codes and returned messages from Set_Msg to the edit user

The function **`Set_Msg`** is used to pass back a top-level return code and message to the edit user (you) that is displayed in the upper-right corner of the edit screen.

It is important to understand that function-level RC and message values passed back by other functions (like **Get_Line$**, for instance) are **not** automatically passed through to the edit user, even when the RC value is non-zero, signifying a warning or error condition was detected. You must explicitly call **`Set_Msg`** yourself if you wish a top-level status message to be displayed back to the user once the macro completes execution.

The values set by **`Set_Msg`** will be overridden by any other subsequent call to **`Set_Msg`**. If you call this function more than once, only the RC and Message of the last call will be used as the top-level status, and displayed on the edit screen when the macro completes.

If you execute a macro that never calls **`Set_Msg`**, it is treated the same as if **`Set_Msg(0,"")`** were called; that is, no message will appear on the edit screen when the macro completes.

Supposing you wanted to "pass along" the function-level status values up to the top-level status after an SPFLite function call, it would require you to call the **Set_Msg** function. If you did this, the call would take the form **`Set_Msg(Get_RC, Get_Msg$)`**.

Example:

```
SPF_Cmd ("LINE 'R5' .1234")
Set_Msg (Get_RC, Get_Msg$)
' at this point, the visible top-level RC and Message
are
' set to the status returned by the SPF_Cmd function
call
```

List of available macro functions

The functions here are divided by categories of interest, for ease of lookup.

Functions that manage macro invocation and arguments

Get_Arg$(n) **Get_Arg$(f,n)**	Returns a specified macro operand number **n,** or a range of operand numbers separated by single blanks.
Get_Arg_Count	Returns the number of supplied macro arguments.
Get_MacName$	Returns the name of the current macro
SPF_Parse	Performs full operand parsing of command operands
Get_Arg_KW	Returns True/False based on a Keywords presence in the macro command line
Get_Arg_KWGroup$	Returns the entered keyword fro a list of mutually exclusive items
Get_Arg_LRef$(n)	Returns the requested relative Line Reference from the macro command line
Get_Arg_Tag$(n)	Returns the requested relative Tag Operand from the macro command line
Get_Arg_TextLit$(n)	Returns the requested relative Text Literal from the macro command line
Get_Arg_NumLit$(n)	Returns the requested relative Numeric Literal from the macro command line
SPF_Quote$	Returns a string value properly enclosed in quotes
SPF_UnQuote$	Returns a string value with any existing paired outer quotes removed.

Get_FullPath$(file-name)	Returns a fully qualified path for a filename.
Is_Line_Cmd	Returns TRUE if macro launched as a line-command macro, otherwise FALSE
Is_Primary_Cmd	Returns TRUE if macro launched as a primary-command macro, otherwise FALSE
Set_Msg([*rc-num,*] *msg-str*)	Sets the visible (top-level) return code and message that is returned to the user
Halt([rc-num,] msg-str)	Sets the visible (top-level) return code and message that is returned to the user and then immediately terminates the macro

Functions that manage environment and SET variables

Get_EnvVar$(*env-var-str*)	Returns the specified Windows Environment Variable data.
Get_SETVar$(*set-var-str*)	Returns the specified variable value from the SPFLite SET variable pool.
Set_SETVar(*set-var-str, value-str*)	Sets the specified variable value in the SPFLite SET variable pool.

Functions that manage the Edit file

Get_FilePath$	Returns the current file path for the current file.
Get_FileBase$	Returns the base filename of the current edit file.
Get_FileName$	Returns the filename of the current edit file.
Get_FileExt$	Returns the current file extension.
Get_FileDate$	Returns the last modified date of the current file.
Get_FileTime$	Returns the last modified time of the current file.
Get_First_LPtr	Returns a Line Pointer to the first line of the Edit session
Get_Last_LPtr	Returns a Line Pointer to the last line of the Edit session
Get_Select_First_LPtr	Returns a Line Pointer to the first line of a selected text block

Get_Select_Last_LPtr	Returns a Line Pointer to the last line of a selected text block
Get_Select_Col	Returns a number indicating the leftmost selected column
Get_Select_Len	Returns a number indicating the length of the selected data
Get_Profile$	Return a PROFILE setting as a string.

Functions that manage the Edit session

Get_RC	Contains the Return Code from the last executed SPFLite function call.
Get_Msg$	Contains the text of the message issued by the last executed SPFLite function call
Set_Csr(*line-ptr,col-num [, len-num]*)	Set the desired location for the cursor following macro execution.
Get_Csr_Col	Returns the column number of the current cursor line
Get_Csr_LPtr	Returns the line pointer of the line containing the cursor
Get_TopScrn_LPtr	Returns the line pointer of the line which is currently at the top of screen.
Set_TopScrn_LPtr	Sets the line pointer which is to appear at the top of screen on macro exit.
Get_Curr_Line$	Returns the contents of the data line where the cursor was located **at the start** of macro processing
Get_Curr_Word$	Returns the contents of the current 'word' where the cursor was located **at the start** of macro processing
Get_Curr_Path$	Returns the full path of the Windows current working directory
Get_EXE_Path$	Returns the full path of the folder where SPFLite.EXE resides.
Get_INI_Path$	Returns the full path of the SPFLite data

	storage folder.
Get_LBound	Returns the current Left bounds value.
Get_RBound	Returns the current Right bounds value
Get_Modified	Returns the current file modified status as TRUE or FALSE.
Get_Modified_Filename	Returns the modified status for an individual file within an MEdit session
Get_Session_Type$	Returns a string containing the 'type' of the current edit tab.
Get_Uniq_ID	Returns a unique ID number that identifies this particular edit session.

Functions that manage Edit text data

Get_Line$(*line-ptr*)	Returns the current Edit text data for the specified line-ptr
Get_Clr_Line$(line-ptr)	Returns the current color control line for the specified line-ptr
Get_Line_Len(line-ptr)	Returns the length of the current Edit text data for the specified line-ptr
Get_Line_Type$(*line-ptr*)	Returns a text string indicating the particular line type
Get_XLines(*line-ptr*)	Returns the number of Excluded lines in an Exclude block
Get_XStatus$(*line-ptr*)	Returns an X/NX string value indicating the Exclude status of a data line
IS_xxxx(*line-ptr*)	Tests if a specific line (line-ptr) is a particular line type
Set_Line(*line-ptr, data-str*)	Will replace the current Edit text data for the specified line-ptr with the new string specified by data-str
Set_Clr_Line(*line-ptr, clr-str*)	Will replace the current color control string for the specified line-ptr with the new string specified by clr-str
SPF_INS(*line-ptr,col-num,data-str*)	Will insert *StrVar* into the text of *line-ptr* at the indicated column *col-num*

SPF_REP(*line-ptr,col-num,data-str*)	Will replace text in *line-ptr* with *StrVar* starting at the indicated column *col-num*
SPF_OVR(line-ptr,col-num,StrVar)	Will overlay text in *line-ptr* with *StrVar* starting at the indicated column *col-num*
SPF_OVR_REP(*line-ptr,col-num,data-str*)	Will overlay replace text in *line-ptr* with *StrVar* starting at the indicated column *col-num*
Get_Next_LPtr(line-ptr, adjust-amount, line-type)	Will adjust a line-pointer by moving it either forward or backward a specified number of lines of a specified line type.
Get_LPtr(*line-ref*)	Convert an external line reference to a line pointer
Get_LNum(*line-ptr*)	Convert a line pointer to an external line number
Get_Uniq_ID	Return a unique ID for this particular edit session.
SPF_Exempt_File(file-ext)	Test if a file extension is in the Text-Exempt list

Functions that manage FIND and LOCATE commands

Get_LOC_LPtr	Returns the line pointer of the last line found by the LOCATE command
Get_FIND_LPtr	Returns the line pointer of the last line found by a FIND / CHANGE command
Get_FIND_Col	Returns the column number where the last FIND / CHANGE was located
Get_FIND_Len	Returns the length of the last FIND / CHANGE search argument

Functions that manage line command operands

Label and Tag functions:

Get_Label$	Returns the .LABEL for a given data line
Get_Tag$	Returns the :TAG for a given data line
Release_Label$	Releases a label obtained via the Request_Label$ function

Request_Label$	Returns a label for a data line, creating a temporary one if needed

Source line-range functions:

Get_Src_LCmd$	Returns a string containing the name of the source line command.
Get_Src1_Lptr	Returns the line pointer of the last line in the source line range, or 0 if not applicable.
Get_Src2_Lptr	Returns the line pointer of the last line in the source line range, or 0 if not applicable.
Get_Src_Op	Returns the optional **n** value on the source line range.
Get_Src_LMOD$	Returns the source line command modifiers.

Destination line-range functions:

Get_Dest_LCmd$	Returns a string containing the name of the source line command.
Get_Dest1_Lptr	Returns the line pointer of the first line in the destination line range, or 0 if not applicable.
Get_Dest2_Lptr	Returns the line pointer of the last line in the destination line range, or 0 if not applicable.
Get_Dest_Op	Returns the optional **n** value on the destination line range command.
Get_Dest_LMOD$	Returns the destination line command modifiers.

Functions that manage globally-stored values

Set_Gbl_Num([TblNum,] ***key-str, value-num*****)**	Stores the numeric *value* under the associated string *key*.
Set_Gbl_Str([TblNum,] ***key-str, value-str*****)**	Stores the string *value* under the associated string *key*.
Get_Gbl_Num([TblNum,] ***key-str*****)**	Retrieves the numeric value associated with *key.*
Get_Gbl_Str$([TblNum,] ***key-str*****)**	Retrieves the string value associated with *key.*

Get_Gbl_Num_Count	Returns the current number of variables stored in the Numeric pool.
Get_Gbl_Str_Count	Returns the current number of variables stored in the String pool.
Get_Gbl_Num_Name$(*index-num*)	Returns the key name of the specified item in the Numeric pool.
Get_Gbl_Num_TableName$(*index-num*)	Returns the Table-number and key name of the specified item in the Numeric pool.
Get_Gbl_Str_Name$(*index-num*)	Returns the key name of the specified item in the String pool.
Get_Gbl_Str_TableName$(*index-num*)	Returns the Table-number and key name of the specified item in the String pool.
Reset_Gbl_Num(*[TblNum]*)	Deletes the entire contents of the Gbl_Num storage area; all Keys and Values are deleted
Reset_Gbl_Str(*[TblNum]*)	Deletes the entire contents of the Gbl_Str storage area; all Keys and Values are deleted

Functions that manage external command execution

SPF_Cmd(*cmd-str*)	Execute the provided SPFLite primary command.
SPF_Shell(*cmd-str*)	Execute the provided command under the System's CMD.EXE command processor.
SPF_Exec(*cmd-str*)	Execute the provided command directly, not under the CMD.EXE command shell.

Functions that manage macro debugging

SPF_Debug(*msg-str*)	Display a message on the debugging console
SPF_Loop_Check(*option-num*)	Fetch and Set the SPFLite Macro loop monitoring function ON or OFF.
SPF_Trace(*mode-num*)	Set SPFLite trace mode to ON, OFF or ERROR

Globally Stored Data

Introduction to globally-stored data

The SPFLite macro facility provides functions which allow you to save numeric and/or string data from your macro. This saved data can be retrieved later in the same macro, in a subsequent new execution of the same macro, or by any other macro executing later.

See Function Details for information on calling these functions.

Globally-stored data can enable the creation of a set of related macros and provide a means for them to communicate with each other. It could also be used to store "state-like" information for a single macro that is executed repeatedly over the course of an edit session.

The stored values are maintained **only** for the duration of the current SPFLite session. You may do multiple Edit/Save functions on different files, and the global data will be maintained. Globally-stored data will be discarded when SPFLite is terminated.

> Globally-stored data is not "persistent" in the way that certain Profile and edit settings are when you have **STATE ON** in effect. Globally-stored data, and **STATE** information, are two completely different concepts.
>
> If you really needed globally-stored data to persist across SPFLite executions, it is possible you could write out the data to a file and then read it back in later, using file-management services provided by thinBasic. We haven't experimented with this ourselves, but you're welcome to try. (If you do, let us know how it works out for you.)
>
> Or you could use the SPFLite SET variable support to retain data using the Get_SetVar$ and Set_SetVar functions.
>
> Also be aware that globally-stored data is only "global" within a given SPFLite execution. If you have multiple instances of SPFLite running in Windows, each one has its **own** copy of globally-stored data for its own execution instance. The global data is **not** shared across multiple SPFLite instances.

The data is held in "storage pools" as a series of key/data pairs. (Some people refer to that as a "map" or "associative storage", if that helps.) The keys may be any strings of your choosing. There are two global storage pools. One is for storing string data, and the other is for storing numeric data.

> Like global variables in any other language, globally-stored values can be a good thing **if** handled carefully, but could cause problems if misused in a poorly designed macro, for reasons well-known to experienced software developers. Persistent data sometimes causes unexpected side effects, which you need to be aware of, and plan for. Generally, you should only use globally-stored data if conventional local variables are not enough. They should be used sparingly, and only as a last resort. (As they say, be careful what you wish for.)

Sub-Tables

Since there are just two Global pools (String and Numeric), you may have multiple requirements for storage of a single type. This is handled by using an Optional sub-table

number to create unique groupings. You may use any number you like to identify a table, and the same key-names may occur in different subtables. The Table-number is entered as an optional first parameter to the Set_Gbl, Get_Gbl and Reset_Gbl functions. If the operand is omitted, a value of zero is assumed.

Examples:

SET_GBL_NUM("KEY1", 12) will store 12 as a value for the key KEY1 in the default Table zero

SET_GBL_NUM(4, "KEY1", 15) will store 15 as a value for the key KEY1 in Table 4

GET_GBL_NUM("KEY1") will return 12 as a value for the key KEY1 from the default Table zero

GET_GBL_NUM(2, "KEY1") will return 0 as a value since it asked for a value from Table 2 which was never used.

GET_GBL_NUM(4, "KEY1") will return 15 as a value for the key KEY1 from Table 4

Undefined keys and deletion of global storage

If you request the value of an undefined key, the functions will return default values of **0** (for Get_Gbl_Num) and a null string (for Get_Gbl_Str$). Thus, it is not necessary to "initialize" global storage if these default values are sufficient.

You can also use the fact that an undefined global number returns a zero value as a "flag" showing that some (other) global value needed to be initialized. Recalling that FALSE is the same thing as zero, here is an example of how to do that:

```
IF GET_GBL_NUM("INITIALIZED") = FALSE THEN
     SET_GBL_STR ("MY-STR-VAR", "some string value")
     SET_GBL_NUM ("MY-NUM-VAR", some-numeric-value)
     SET_GBL_NUM ("INITIALIZED", TRUE)
END IF
```

At present, there is no way to delete an individual key/value pair from a storage pool, once it's defined. To achieve the effect of a deletion, you can store a zero or null value. If it's necessary to determine whether a zero or null is an actually-stored value or just the default value returned when querying for an undefined key, you can check the return code. Querying for a defined key will produce a return code of **0**, and querying for an undefined key will produce a return code of **8**. You can delete **every** global number or string using the functions using Reset_Gbl_Num and Reset_Gbl_Str. These two functions also enable you to delete only a specific sub-table. See the Function details for further assistance.

Example

If you wanted to store the last modified date for the current edit file, you could do so by issuing:

```
Set_Gbl_Str(Get_FileName$, Get_FileDate$)
```

This would create a global variable with a key equal to the filename, and a value equal to the file's modification date. At some future time this macro, or some other macro, could retrieve this particular date for the file by issuing:

```
Date-var = Get_Gbl_Str$("the file name")
```

If still had the same data file actively being edited, you could do this:

```
Date-var = Get_Gbl_Str$(Get_FileName$)
```

There is no limit to the number of global variables, or the size of the saved strings, other than available system memory.

Available global functions

See the relevant section in Function Details for additional information on these functions.

Set_Gbl_Num([TblNum,] *key-str, value-num*)	Stores the ***value-num*** under the key specified by ***key-str***. If a key matching the contents of ***key-str*** already exists, the numeric value it was previously associated with is replaced with the new ***value-num***. If a key matching the contents of the ***key-str*** does not already exist, it is created.
Set_Gbl_Str([TblNum,] *key-str, value-str*)	Stores the ***value-str*** under the key specified by ***key-str***. If a key matching the contents of ***key-str*** already exists, the string value it was previously associated with is replaced with the new ***value-str***. If a key matching the contents of the ***key-str*** does not already exist, it is created.
***num-var* = Get_Gbl_Num([TblNum,] *key-str*)**	Returns the numeric value associated with ***key-str***. If the key does not exist, zero is returned, and an error is stored in the RC and Msg$ variables.
***str-var* = Get_Gbl_Str$([TblNum,] *key-str*)**	Returns the string value associated with ***key-str***. If the key does not exist, a null (empty) string is returned, and an error is stored in the RC and Msg$ variables.
***num-var* = Get_Gbl_Str_Count**	Returns the current number of variables stored in the String pool.
***num-var* = Get_Gbl_Num_Count**	Returns the current number of variables stored in the Numeric pool.
***str-var* = Get_Gbl_Str_Name$ (*index-num*)**	Returns the key name of the specified item in the String pool. ***index-num*** must be a number in the range of 1 through the the value of **Get_Gbl_Str_Count**. The keys stored in the String pool are not stored in any predicable order. To find the location of a given key in the pool, you must do a linear search from beginning to end until

it is found. Pool management is done using software outside the control of SPFLite. Adding new entries to a pool may change the index of previously stored entries. If this happens, you may need to perform a new search to redetermine the location of a previously-located key.

str-var = **Get_Gbl_Str_TableName$** **(*index-num*)**

Returns the Table-number and key name of the specified item in the String pool.
index-num must be a number in the range of 1 through the the value of **Get_Gbl_Str_Count**.

The str-var returned contains the table-number preceding the key name, separated by a comma. e.g. a value like **4,KEYWORD.**

The table number is easily accessed via **VAL(str-var)** The keyname can be accessed via **REMAIN$(str-var, ",")**

See also the note above re: order of entries.

str-var = **Get_Gbl_Num_Name$** **(*index-num*)**

Returns the key name of the specified item in the Numeric pool.
index-num must be a number in the range of 1 through the the value of **Get_Gbl_Num_Count**.

The keys stored in the Numeric pool are not stored in any predicable order. To find the location of a given key in the pool, you must do a linear search from beginning to end until it is found. Pool management is done using software outside the control of SPFLite. Adding new entries to a pool may change the index of previously stored entries. If this happens, you may need to perform a new search to redetermine the location of a previously-located key.

str-var = **Get_Gbl_Num_TableName$** **(*index-num*)**

Returns the Table-number and key name of the specified item in the String pool.
index-num must be a number in the range of 1 through the the value of **Get_Gbl_Str_Count**.

The str-var returned contains the table-number preceding the key name, separated by a comma. e.g. a value like **4,KEYWORD.**

The table number is easily accessed via **VAL(str-var)** The keyname can be accessed via **REMAIN$(str-var, ",")**

See also the note above re: order of entries.

num-var = **Reset_Gbl_Num**

Clears the entire Gbl_Num storage area. Returns True is successful, or False if the Clear fails.

num-var **=** **Reset Gbl Str**	Clears the entire Gbl_Str storage area. Returns True is successful, or False if the Clear fails.

Working with Line Colors

Overview of SPFLite color support

SPFLite provides primary commands and keyboard primitive functions to support what are called Virtual Highlighting Pens.

A highlighting pen can color a portion of text in one of fifteen colors:

The available keyboard primitive functions are:

(Pen/*colorname*)
(Pen/Std)

You can specify a color on **FIND** and **CHANGE** commands. You can also use **LOCATE** to find lines having a particular color, and can use **RESET** to clear the color from all lines.

See the main SPFLite Help documentation for more details on these commands and primitive functions.

Macro functions for color support

The color interface for macros provides a means to query the current color state of a line, and a way to alter that state.

> What could you do with this capability? You may want to use it to analyze a block of text, and add or remove highlighting colors based on some particular condition or data value you are interested in. You could look or, or change, the text coloring in ways that are beyond the ability of native SPFLite FIND and CHANGE commands.

Recall that text highlighting pens come in up to 15 colors: The current color of a character position in a given line is represented by a unique single character. These letters and their color name are:

B	Blue
G	Green
Y	Yellow
R	Red
K	Black
N	Navy
T	Teal
V	Violet
O	Orange
A	Gray
L	Lime
C	Cyan
P	Pink
M	Magenta
W	White

Character positions having the standard text color are presented by a blank.

Suppose you have a line of data colored like this. (The colors are shown using one possible color palette selection. You can use the Screen tab of the Global Options dialog to choose any desired text foreground and background for each of the four possible colors.)

```
000001 LINE ONE OF COLOR-HIGHLIGHTED DATA
```

To "get" the color of this line, you ask for a string that represents the color of each column of data on that line. Assume that **currLptr** is the line pointer to that line, and **Clr_Line** is a string variable:

Clr_Line = Get_Clr_Line$ (**currLptr**)

This would produce an "array" of color codes, one character for each character in the line of data, as a string value. Here's what the function would return:

```
data line  = "LINE ONE OF COLOR-HIGHLIGHTED DATA"
color line = "     RRR                      BBBB"
```

Because the data line has a length of 34, the string returned in the color-line variable is also of length 34.

Now, suppose you wanted to set the text value "HIGHLIGHTED" to Yellow, and make the word "ONE" the standard color. You would pass a "new" color array to the function **Set_Clr_Line**. This function **completely replaces** all existing colors for that line. So, the typical way you would do this is:

- Call **Get_Clr_Line$** to find the current colors for the line
- Use **thinBasic** statements to modify the color-string as needed.
- In all likelihood, color changes will involve the use of the **MID$** function and/or the **MID$** statement. See the **thinBasic** documentation for more details on these features.
- Call **Set_Clr_Line** to replace the colors.
- Be aware that the word COLOR is a reserved word in **thinBasic**; you can't use it as a variable name.

Assume that you have the line of data shown above, and the variable **currLptr** contains a Line Pointer value to that line. To remove the red color from the word "ONE", and make "HIGHLIGHTED" display as Yellow, you'd write statements like the following:

```
DIM Clr_Line AS STRING
Clr_Line = Get_Clr_Line$ (currLptr)
MID$(Clr_Line, 6, 3) = "   "              ' remove color
from ONE
MID$(Clr_Line, 19, 11) = "YYYYYYYYYYY"    ' make
HIGHLIGHTED display as Yellow
Set_Clr_Line (currLptr, Clr_Line)
```

where

```
data line  = "LINE ONE OF COLOR HIGHLIGHTED DATA"
color line = "                  YYYYYYYYYYY BBBB"
```

Note that "DATA" was Blue before, and we are setting to Blue "again". That's OK. We also changed "ONE" from Red to "standard" by making its color-code positions blank, and we

changed the word "HIGHLIGHTED" from standard to Yellow. Observe that each time you call **Set_Clr_Line**, you are performing a complete replacement of the colors of every character on the data line.

Rules:

- Only data lines have a color interface, not special lines (like TABS, BNDS and NOTE lines). If the LPTR is not for a valid, existing data line, RC is set to 8, and no other action is taken. You cannot change the color of special lines.

- **Get_Clr_Line$** always returns a string of length identical to the data line, even if that length is zero.

- You are not allowed to attempt to change the color of non-existent character positions of a data line. The string passed to **Set_Clr_Line** cannot be longer than the data line you are trying to update, which is another way of saying that it cannot be longer than the string returned by **Get_Clr_Line$** for that same data line. If **Set_Clr_Line** does get a string longer than the data line, RC is set to 8 and the colors on the data line are not changed. For this reason, it is best to always fetch the current color state of a line into a string using **Get_Clr_Line$**, and then modify that string, being careful not to extend its length. The function Get_Line_Len may be useful in this regard.

  You should normally complete any macro modifications to the data line before proceeding with changes to the associated color line. This will ensure the line length problem will not occur.

- If the color string passed to **Set_Clr_Line** is shorter than the data line, the color string is effectively treated as if padded with blanks to make it as long as the data line.

- If any color codes other than the valid ones (or blank) are passed to **Set_Clr_Line**, RC is set to 8, and no other action is taken.

- It is not necessary for **STATE ON** to be in effect if **Set_Clr_Line** is used. However, unless **STATE** is **ON**, any changes to the text colors will not persist after the edit file is closed.

- For **Get_Clr_Line$**, the color code letters are returned in upper case. For **Set_Clr_Line**, you may specify these codes in either upper or lower case.

Working with the Clipboard

thinBasic allows you to access the Windows clipboard. You can use this fact to gain access to data placed into the clipboard during editing when a macro is executing.

To get the contents of the clipboard:

```
DIM clip AS STRING
clip = ClipBoard_GetText
```

To change the contents of the clipboard:

```
ClipBoard_SetText (stringExpression)
```

Keep in mind that when you copy text into the clipboard, it will normally be stored with a trailing CR/LF pair. SPFLite handles this automatically when you are editing, but if you directly access the clipboard in a **thinBasic** macro, you are going to see a CR/LF pair for every line in the clipboard - even when you only copied a single word from a single line.

If this an issue, there are few things you can do to deal with this:

- You can KEYMAP a key or mouse button to (CopyPasteRaw), which copies data to the clipboard without the CR/LF line terminator.
- You can scan the returned string, and trim off the CR/LF

Example:

```
DIM clip AS STRING
clip = ClipBoard_GetText
if Right$(clip,2) = $CRLF then
    clip = Remove$(clip, $CRLF)
end if
```

This example would remove ALL instances of CR/LF, even ones embedded in the middle if the clipboard had more than one line.

If you are going to handle multi-line clipboards, you will probably need more involved macro coding than this, but for simple strings this should be enough to get by.

Function Details

Introduction

For the descriptions below, each function is shown as it would be called to return a value to a local variable within the script.

All functions which return a string contain a trailing $ and are shown returning a value to **str-var**, a string variable.
All functions which return a numeric value do not contain a trailing $ and are shown returning a value to **num-var**, a numeric variable.

Note that in the macro language, there is no requirement that the data returned by a function be assigned to a local script variable before using it. For example, to check if the current Edit file has been modified, it is perfectly correct to code:

```
if Get_Modified = TRUE then ...
```

It does not have to be coded as:

```
LocalVar = Get_Modified
if LocalVar = TRUE then ...
```

In fact, because the **Get_Modified** function returns a TRUE or FALSE value already, you don't even have to compare it explicitly to TRUE or FALSE. You can use the value returned by the function directly, like this:

```
if Get_Modified then ...
```

This is different - and much simpler - than the IBM Rexx / ISREDIT interface, in which this technique was not available.

To illustrate, here is a simple macro that emulates a built-in command that existed in Tritus SPF:

```
' QMOD.MACRO
IF GET_MODIFIED THEN
   SET_MSG (OK, "DATA MODIFIED")
ELSE
   SET_MSG (OK, "DATA NOT MODIFIED")
END IF
```

With the * on the status line, and color-coded edit tabs, a macro to show the modified state isn't really necessary in SPFLite. It's shown here just as an example.

Detailed description of functions

Note that in the descriptions below, names like ***line-ptr*** are just symbolic for an actual name or expression you might use. In real code, you can't literally have variable names with minus signs in them (underscores are allowed, though).

Function operands may be either variables or expressions, providing they are of the correct type; you are not limited to just using variables. For a given function, consult the function description as to whether a particular operand should be numeric-valued or string-valued.

Operands whose descriptions end in ***-num*** or ***-ptr*** are numeric. Operands whose descriptions end in ***-str*** are strings.

When the results of a function are not important to you, it is not necessary to assign the results to a variable; you can just call this function as a "procedure call" statement, by simply specifying the function name and its parameters on a single line, with no variable name and = equal sign preceding it.

str-var **= Get_Arg$(*arg-num*)** ***str-var*** **= Get_Arg$(*first-arg-num*,*last-arg-num*)**		
Operands:	***arg-num***	the number of the only macro operand
	first-arg-num	the number of the first macro operand
	last-arg-num	the number of the last macro operand

Returns:		For (***arg-num***), it returns the specified macro operand number ***arg-num***, where the operands are numbered left to right starting at 1.
		
		For (***first-arg-num,last-arg-num***) it returns a range of operands from operand number ***first-arg-num*** through operand number ***last-arg-num***, where the operands are numbered left to right starting at 1.
		
		Unless ***last-arg-num*** is **0**, then ***first-arg-num*** must be <= ***last-arg-num***.
		
		Macro arguments are treated as simple space delimited operands. If an undefined / illegal argument number is specified, a "" (null) will be returned.
		
		If a valid argument number, RC will be 0 and Msg$ will be "" (Null)
		If an invalid argument number, RC will be 8, and Msg$ will contain an error message

Special Notes:		If **Get_Arg$(0)** is requested, the complete macro arguments are returned as one string, where the individual arguments are separated from each other by blanks.

num-var **= Get_Arg_Count**

Operands:	none	
Returns:		Returns the number of supplied macro arguments. If no arguments are specified, 0 will be returned. When the macro header contains one or more default values, the number of default values supplied will be the minimum value returned by **`Get_Arg_Count`**. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:		

num-var **`= Get_Arg_KW("keyword")`**		
Operands:	keyword	The literal value of the keyword to be tested.
Returns:		Returns True of False depending on whether the specified Keyword has been entered as an operand. Note: this function is only valid following successful completion of an SPF_Parse function call to parse the command line operands. RC will always be 0 and Msg$ will be "" (Null) on successful completion. RC will be 8 and Msg$ set to an error message on any error. e.g. an invalid 'n' value
Special Notes:		A full description of using this function will be found in How do I access the parsed out Operands?

str-var **`= Get_Arg_KWGroup$("list-name")`**		
Operands:	list-name	The literal value of the Keyword list-name for which the value of the entered keyword is to be returned.
Returns:		Returns the actual keyword which was entered on the command operand line. If no keyword in the list of valid keywords was entered, this function will return "" (Null) Note: this function is only valid following successful completion of an SPF_Parse function call to parse the command line operands. RC will always be 0 and Msg$ will be "" (Null) on successful completion. RC will be 8 and Msg$ set to an error message on any error. e.g. an invalid 'n' value

Special Notes:	A full description of using this function will be found in How do I access the parsed out Operands?

***str-var* = Get_Arg_LRef$(n)**		
Operands:	n	The relative Line Reference value desired, counting from left to right.
Returns:		Returns the specified Line Reference value. Note: this function is only valid following successful completion of an SPF_Parse function call to parse the command line operands. RC will always be 0 and Msg$ will be "" (Null) on successful completion. RC will be 8 and Msg$ set to an error message on any error. e.g. an invalid 'n' value
Special Notes:		A full description of using this function will be found in How do I access the parsed out Operands?

***str-var* = Get_Arg_NumLit$(n)**		
Operands:	n	The relative Numeric Literal value desired, counting from left to right.
Returns:		Returns the specified Numeric Literal value. Note: this function is only valid following successful completion of an SPF_Parse function call to parse the command line operands. RC will always be 0 and Msg$ will be "" (Null) on successful completion. RC will be 8 and Msg$ set to an error message on any error. e.g. an invalid 'n' value
Special Notes:		A full description of using this function will be found in How do I access the parsed out Operands?

***str-var* = Get_Arg_Tag$(n)**		
Operands:	n	The relative Tag Operand value desired, counting from left to right.

Returns:	Returns the specified Tag Operand value. Note: this function is only valid following successful completion of an SPF_Parse function call to parse the command line operands. RC will always be 0 and Msg$ will be "" (Null) on successful completion. RC will be 8 and Msg$ set to an error message on any error. e.g. an invalid 'n' value
Special Notes:	A full description of using this function will be found in How do I access the parsed out Operands?

***str-var* = Get_Arg_TextLit$(n)**		
Operands:	n	The relative Text Literal value desired, counting from left to right.
Returns:		Returns the specified Text Literal value. Note: this function is only valid following successful completion of an SPF_Parse function call to parse the command line operands. RC will always be 0 and Msg$ will be "" (Null) on successful completion. RC will be 8 and Msg$ set to an error message on any error. e.g. an invalid 'n' value
Special Notes:		A full description of using this function will be found in How do I access the parsed out Operands?

***str-var* = Get_Clr_Line$(*line-ptr*)**		
Operands:	***line-ptr***	the line pointer for the desired color control line to be fetched.
Returns:		Returns the current color control line data for the specified ***line-ptr. Line-ptr*** must be a line pointer for a data line. str-var will be "" (zero-length string) if the ***line-ptr*** value is invalid. RC will be 0 and Msg$ will be "" (Null) for successful completion RC will be 8 and Msg$ set to an error message on failure (e.g. invalid ***line-ptr***)

Special Notes:

***num-var* = Get_Csr_Col**	
Operands:	none
Returns:	Returns the column number of the current cursor line. If the cursor is not on a valid line in the text area, then 0 (zero) will be returned. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***num-var* = Get_Csr_LPtr**	
Operands:	none
Returns:	Returns the line pointer of the line containing the cursor. If the cursor is not on a valid line in the text area, then 0 (zero) will be returned. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***str-var* = Get_Curr_Line$**	
Operands:	none
Returns:	Returns the contents of the edit data line where the cursor was located **at the start** of macro processing. If the cursor is not located on a valid data line, the value returned will be "" (a zero-length string). RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***str-var* = Get_Curr_Path$**	
Operands:	none
Returns:	Returns the full path of the Windows 'current' working directory. This is the current directory in effect when SPFLite is started up from a command prompt, or is the Start directory property when SPFLite is run from an icon, such as from the Windows Desktop. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	None

***str-var* = Get_Curr_Word$**	
Operands:	none
Returns:	Returns the contents of the current 'word' where the cursor was located **at the start** of macro processing. If the cursor is not located on a valid data line, or located on white space or delimiters within the line, the value returned will be "" (a zero-length string). Note that 'word' means the same here as it does when used with a FIND 'xxx' WORD command. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	An **entire** 'word' will be returned regardless of where within the word the cursor is located. It doesn't matter whether the cursor is on the first, last, or any other character of the word. For purposes of determining what a 'word' is, the same rules are used as for a FIND/CHANGE command using the WORD operand. i.e. based on the contents of the Profile WORD character setting.

***str-var* = Get_Dest_LCmd$**	
Operands:	none
Returns:	The string containing the destination line command if present. This may be an empty (null) string if no

	destination line range was specified. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	Get a string containing the name of the destination line command. This may be an A/AA, B/BB, H/HH, W/WW, O/OO, or OR/ORR command, or may be an empty (null) string if no destination line range was specified.

***num-var* = Get_Dest_Op**	
Operands:	none
Returns:	Returns the optional numeric line operand, if it is entered, for the destination line command if one is used. If none is entered, this will return 0 (zero). RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	For example, if you have a destination line range of **H5**, **Get_Dest_Op** will return the number **5**. The function returns 0 when no explicit command operand number is specified. For instance, if you define a destination block such as HH which happens to be 5 lines long, the function will still return 0, because the number 5 was not actually specified on the HH block.

***str-var* = Get_Dest_LMOD$**	
Operands:	none
Returns:	Gets the **+** or **-** (post-unexclude or post-exclude) modifier and K (Keep) modifier if one is present on the destination line range. The returned value is always two characters long. Position 1 containing **+** or **-** or blank, and position 2 will contain a K or blank. The **/** and **** line command modifiers are processed, if present, and this gets reflected in first and last LPTR values for the line range, but the **/** and **** codes themselves are not returned as LMOD values. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

num-var **= Get_Dest1_LPtr**	
Operands:	none
Returns:	Get the line pointer of the first (or only) line in the destination line range, or 0 if not applicable. The destination line range may be defined by A/AA, B/BB, H/HH, W/WW, O/OO, or OR/ORR command(s), if present. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	Note that for A/AA and B/BB, the value returned is **not** normalized to be treated as an A line command. You will have to look at Get_Dest_LCmd to determine the line command used, and act accordingly.

num-var **= Get_Dest2_LPtr**	
Operands:	none
Returns:	Get the line pointer of the last (or only) line in the destination line range, or 0 if not applicable. The destination line range may be defined by A/AA, B/BB, H/HH, W/WW, O/OO, or OR/ORR command(s), if present. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	Note that for A/AA and B/BB, the value returned is **not** normalized to be treated as an A line command. You will have to look at Get_Dest_LCmd to determine the line command used, and act accordingly.

str-var **= Get_ENVVAR$(*env-var-str*)**		
Operands:	***env-var-str***	the name of the environment variable for which the data is to be returned.

Returns:	Returns the string value for the specified var-name. If an unknown variable name is specified, a "" (a zero-length string) will be returned. If a valid ***env-var-str***, RC will be 0 and Msg$ will be "" (Null) If an invalid ***env-var-str***, RC will be 8, and Msg$ will contain an error message
Special Notes:	For example, **`Get_ENVVAR$("COMSPEC")`** would return the name of the system's command processor (Typically **`C:\Windows\System32\Cmd.EXE`**)

str-var **`= Get_EXE_Path$`**	
Operands:	none
Returns:	Returns the full path of the folder where SPFLite.EXE is run. This will usually be C:\Program Files\SPFLite. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

str-var **`= Get_FileBase$`**	
Operands:	none
Returns:	Returns the base filename of the current edit file. e.g. for MYFILE.TXT it would return MYFILE RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

str-var **`= Get_FileDate$`**	
Operands:	none
Returns:	Returns the last modified date of the current file in the format YYYY-MM-DD.

	RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***str-var* = Get_FileName$**	
Operands:	none
Returns:	Returns the filename of the current edit file. e.g. MYFILE.TXT RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***str-var* = Get_FileExt$**	
Operands:	none
Returns:	Returns the current file extension, including the leading period. e.g. for MYFILE.TXT it would return .TXT RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***str-var* = Get_FilePath$**	
Operands:	none
Returns:	Returns the current file path for the current file. e.g. C:\Users\Me\Documents RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***str-var* = Get_FileTime$**

Operands:	none
Returns:	Returns the last modified time of the current file in the format hh:mm:ss RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***num-var* = Get_FIND_Col**	
Operands:	none
Returns:	Returns the column number where the last FIND / CHANGE was located. Until the first FIND / CHANGE command is issued for the current Edit file, this will return 0 (zero). RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***num-var* = Get_FIND_Len**	
Operands:	none
Returns:	Returns the length of the last FIND / CHANGE search argument. Until the first FIND / CHANGE command is issued for the current Edit file, this will return 0 (zero). RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***num-var* = Get_FIND_LPtr**	
Operands:	none
Returns:	Returns the line pointer of the last line found by a FIND / CHANGE command. Until the first FIND / CHANGE command is issued for the current Edit file, this will return

	0 (zero). RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***num-var* = Get_First_LPtr**	
Operands:	none
Returns:	Returns a Line Pointer to the first line of the Edit session. If there are no lines in the dataset, **Get_FIRST_LPTR** returns 0 (zero). RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***str-var* = Get_FullPath$(file-name)**		
Operands:	**file-name**	The file name to be fully qualified.
Returns:		Returns a fully qualified filename including `Drive:\path\filename` If the macro call is issued from a temporary session (CLIP, Set-Edit, (Empty), etc.) the path assumed will be the last displayed path in File Manager. If the call is issued from a normal Edit/Browse session, the path assumed will be that of the currently loaded file.Line Pointer to the first line of the Edit session. If there are no lines in the dataset, If the passed *`file-name`* is Null, or already appears to be qualified, it is returned untouched and the RC will be 8, and Msg$ will contain an appropriate error message. Otherwise the RC will always be 0 and Msg$ will be "" (Null)
Special Notes:		

num-var **= Get_Gbl_Num([TblNum,]*key-str*)**		
Operands:	***TblNum***	Optional. The sub-table number to be searched. If omitted, zero is assumed.
	key-str	the desired key for which the associated value is desired.
Returns:		Returns the requested value to num-var. If the ***key-str*** is invalid / not found, zero (0) is returned. A successful retrieval will set RC to 0 and Msg$ will be "" (Null) An unsuccessful retrieval, will set RC to 8 and Msg$ to an error message.
Special Notes:		

num-var **= Get_Gbl_Num_Count**		
Operands:	none	
Returns:		Returns the current number of stored numeric key/value pairs. RC is always set to 0 and Msg$ will be "" (Null)
Special Notes:		

str-var **= Get_Gbl_Num_Name$(*index-num*)**		
Operands:	***index-num***	the index number of the desired numeric entry.
Returns:		Returns the current key name associated with this entry to str-var. If successful, RC is always set to 0 and Msg$ will be "" (Null) If unsuccessful (invalid ***index-num***), RC will be 8, and Msg$ will contain an error message.
Special Notes:		

***str-var* = Get_Gbl_Num_TableName$(*index-num*)**		
Operands:	***index-num***	the index number of the desired numeric entry.
Returns:		Returns the current Table-Number and key name associated with this entry to str-var. The Table-number precedes the key name separated by a comma. e.g. **4,VARNAME** The table number is easily accessed via **VAL(str-var)** The keyname may be accessed via **REMAIN$(strvar, ",")** If successful, RC is always set to 0 and Msg$ will be "" (Null) If unsuccessful (invalid ***index-num***), RC will be 8, and Msg$ will contain an error message.
Special Notes:		

***str-var* = Get_Gbl_Str$([TblNum,]*key-str*)**		
Operands	***TblNum***	Optional. The sub-table number to be searched. If omitted, zero is assumed.
	key-str	the desired key for which the associated value is desired.
Returns:		Returns the requested value to str-var. If the ***key-str*** is invalid / not found, "" (null) is returned. A successful retrieval will set RC to 0 and Msg$ will be "" (Null) An unsuccessful retrieval, will set RC to 8 and Msg$ to an error message.
Special Notes:		

***num-var* = Get_Gbl_Str_Count**	
Operands:	none
Returns:	Returns the current number of stored numeric key/value pairs.

	RC is always set to 0 and Msg$ will be "" (Null)
Special Notes:	

***str-var* = Get_Gbl_Str_Name$(*index-num*)**		
Operands:	***index-num***	the index number of the desired numeric entry.
Returns:		Returns the current key name associated with this entry to str-var. If successful, RC is always set to 0 and Msg$ will be "" (Null) If unsuccessful (invalid ***index-num***), RC will be 8, and Msg$ will contain an error message.
Special Notes:		

***str-var* = Get_Gbl_Str_TableName$(*index-num*)**		
Operands:	***index-num***	the index number of the desired numeric entry.
Returns:		Returns the current Table-Number and key name associated with this entry to str-var. The Table-number precedes the key name separated by a comma. e.g. **4,VARNAME** The table number is easily accessed via **VAL(str-var)** The keyname may be accessed via **REMAIN$(strvar, ",")** If successful, RC is always set to 0 and Msg$ will be "" (Null) If unsuccessful (invalid ***index-num***), RC will be 8, and Msg$ will contain an error message.
Special Notes:		

***str-var* = Get_INI_Path$**	
Operands:	none

Returns:	Returns the full path of the SPFLite data storage folder. i.e. the folder where SPFLite data files are stored (INI files, Profiles, Macros, etc.) RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***str-var* = Get_Label$(*line-ptr*)**

Operands:	***line-ptr***	The line pointer of the data line whose line pointer is to be obtained.
Returns:		If the ***line-ptr*** argument is a valid line pointer to a data line, the function will return the label associated with that data line, if one exists; otherwise it will return a null (empty) string; the RC value will be set to 0 when the ***line-ptr*** is associated with a data line. The string value of the returned label will be just as it appears in the edit session, with a leading . dot and one to five letters in upper case. If ***line-ptr*** is an invalid line pointer value, or is the line pointer of a non-data line, the function will return a null (empty) string and set RC to 8.
Special Notes:		

***num-var* = Get_LBound**

Operands:	none
Returns:	Returns the current Left bounds value. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***num-var* = Get_Last_LPtr**

Operands:	none

Returns:	Returns a Line Pointer to the last line of the Edit session. If there are no lines in the dataset, **`Get_LAST_LPTR`** returns 0 (zero). RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

str-var **`= Get_Line$(`** ***`line-ptr`*** **`)`**		
Operands:	***line-ptr***	the line pointer for the desired line to be fetched.
Returns:		Returns the current Edit text data for the specified ***line-ptr. Line-ptr*** can be a line pointer for a data line or a special line. str-var will be "" (zero-length string) if the ***line-ptr*** value is invalid. RC will be 0 and Msg$ will be "" (Null) for successful completion RC will be 8 and Msg$ set to an error message on failure (e.g. invalid ***line-ptr***)
Special Notes:		

num-var **`= Get_Line_Len(`** ***`line-ptr`*** **`)`**		
Operands:	***line-ptr***	the line pointer of the desired line.
Returns:		Returns the length of the current Edit data line for the specified ***line-ptr. Line-ptr*** must be a line pointer for a data line, not for a special line. num-var will be 0 if the ***line-ptr*** value is invalid or is not the line pointer of a data line. RC will be 0 and Msg$ will be "" (Null) for successful completion RC will be 8 and Msg$ set to an error message on failure (e.g. invalid ***line-ptr***)
Special Notes:		

str-var **= Get_Line_Type$(*line-ptr*)**		
Operands:	***line-ptr***	the line pointer for which the line's type is to be returned
Returns:		Returns a string containing one of the following values: `DATA, EXCL, TOP, BOT, TABS, BNDS, COLS, WORD, MARK, MASK, PROF, FILE or NOTE.` If the extended NOTE types are used, the returned string will be xNOTE (e.g. ANOTE, BNOTE, XNOTE etc.) If ***line-ptr*** is invalid, RC will be set to 8. Otherwise, RC is set to 0 and Msg$ will be "" (Null)
Special Notes:		

num-var **= Get_LNum(*line-ptr*)**		
Operands:	***line-ptr***	the specific line-ptr for which the line number is desired.
Returns:		If the line-ptr is valid, the associated external line number is returned; RC is set to 0 and Msg$ set to "" (null) If the line-ptr is invalid, "" (null) is returned, RC is set to 8, and Msg$ is set to an error message.
Special Notes:		Only data lines have line numbers, so if you attempt to use Get_LNum$ using a line-ptr value for which a call to IS_DATA(line-ptr) would have returned False, the call to GET_LNUM$(line-ptr) will with RC=8 and a returned value of 0. Assuming the edit session is not operating in Fast Renumber mode, the line number of the last line of the file, and the number of lines in the file, are the same number; call this number N. If you call Get_LNUM with an LPTR value <= 1 or > N+2, the Line Pointer parameter is out of range, and Get_LNUM will return 0.

num-var **= Get_LOC_LPtr**	
Operands:	none

Returns:	Returns the line pointer of the last line found by the LOCATE command. If the last LOCATE was for a special line type (FILE, SPECIAL, etc.) which would be a non-data line, this will return zero (0). Until the first LOCATE command is issued for the current Edit file, this will return 0 (zero). RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***num-var* = Get_LPtr(*line-ref*)**		
Operands:	***line-ref***	a line reference. This could be a line number as it is displayed on the screen, or a line label (.HERE, .THERE, etc.) If the line reference is a simple line number, the ***line-ref*** value may be specified either as a string expression or as a numeric expression.
Returns:		If the line reference is valid, the associated line pointer is returned; RC is set to 0 and Msg$ set to "" (null) If the line reference is invalid, 0 (zero) is returned, RC is set to 8, and Msg$ is set to an error message.
Special Notes:		

***str-var* = Get_MacName$**	
Operands:	none
Returns:	Returns the name of the currently running macro. For a macro called ABC.MACRO, the value of the string returned will be "ABC". RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***str-var* = Get_MASKLine$**	
Operands:	none
Returns:	Returns the current MASK line data. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***num-var* = Get_Modified**	
Operands:	none
Returns:	Returns TRUE or FALSE. True is returned if the current Edit file has been modified since loaded or the last SAVE command; otherwise false if the file is unmodified. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	Get_Modified would return True under the same conditions that would produce an * asterisk on the Edit status line or would change the color of the Edit tab (assuming the color palettes for modified and unmodified tab colors have been properly set). In the case of an MEDIT session, Get_Modified will return True if any file in the MEDIT session has been modified.

***num-var* = Get_Modified_FileName**	
Operands:	filename The name of the file within a MEdit session for which the modified status is required.
Returns:	Returns TRUE or FALSE. True is returned if the specified file has been modified since loaded or the last SAVE command; otherwise false if the file is unmodified. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	If the function is used in a non-MEdit session, it will act exactly like the basic Get_Modified function.

***str-var* = Get_Msg$**		
Operands:	none	
Returns:	Contains the text of the last message issued by an SPFLite interface function. If a non-successful Return code had been issued by a function, the message associated with that error can be fetched with this function call. **Get_Msg$** and **Get_RC** do **not** themselves modify the function-level RC or Message status values, but simply report them. That is, **Get_Msg$** and **Get_RC** do not have, or set, function-level status values of their own, but only return the last RC or Message value set by some **other** SPFLite interface function.	
Special Notes:		

***num-var* = Get_Next_LPtr(line-ptr, adjust-amount, line-type)**			
Operands:	***line-ptr***	The starting line-pointer to be adjusted and returned.	
	col-num	The number of lines by which the line-ptr is to be adjusted. Positive values indicate adjustment **downward** in the file (towards the last line) and negative values indicate adjustment **upward** in the file (towards the top line).	
	line-type	This is a string operand which may be one of:	
		DATA	Adjustment is made where *adjust-amount* is measured by counting only DATA line types
		SPECIAL	Adjustment is made where *adjust-amount* is measured by counting only non-DATA (i.e. SPECIAL) line types
		:tagname	Adjustment is made where *adjust-amount* is measured by counting only lines which have a matching line tag.
Returns:	The adjusted line-ptr as requested. If it is impossible to satisfy the request (e.g. movement has reached either the top or bottom of the file, depending on direction) then 0 (zero) will be returned and RC and Msg$ set with appropriate error indications.		
Special	The Get_Next_LPtr function will typically be used in line		

Notes:	scanning loops to move from a currently processed line to the next/previous line of a particular type.

***str-var* = Get_Profile$(*option-str*)**

Operands: ***option-str*** A string containing the name of a PROFILE option. The permitted options are:

Option name	Return value
	profile-name [USING(*alt-prof-name*)]
NAME	LOCKED | UNLOCKED
AUTOBK	ON | OFF
UP	ON | OFF PROMPT | NOPROMPT
AUTOSA	ON | OFF | AUTO:on | AUTO:off
VE	C | T
CAPS	DS | CS
CASE	ON | OFF
CHANGE	CRLF | LF | CR | NL | X'xx'
	ON | OFF
COLS	ON | OFF
EOL	OFF | FIND ON | OFF AUTO ON | OFF
HEX	nnn
HIDE	nnn
HILITE	ON | OFF
IMPORTT	the current MASK string
ABS	nnn
LRECL	NONE | EDIT | ALL
MARK	ON | OFF [+/-nn]
MASK	ON | OFF | C
MINLEN	F | V | U
NOTIFY	CSR | PAGE | HALF | ... ETC.
PAGE	nnn
PRESER	ANSI | UTF8 | UTF16 | UTF16BE |
VE	EBCDIC
RECFM	PRIOR | LABEL | FIRST | LAST | NEW
	ON | OFF
SCROLL	OFF | *string*
	ON | OFF
SETUND	A string containing all characters which
O	are considered valid Word characters
SOURCE	for the current Profile.
START	
STATE	
SUBARG	
TABS	
WORD	

Returns: The string value of the requested option is returned.

	RC will be 8 if the ***option-str*** is not a valid string PROFILE option name; otherwise 0.

Special Notes:	There is no corresponding general function to set a PROFILE option. To accomplish that, issue an SPF_CMD for the desired setting. For example: SPF_CMD("EOL AUTO").
	
	For numeric settings like LRECL, if you need to work with the returned value in numeric form, you can enclose the request in a VAL() function, or assign the result to a numeric variable first.
	
	Example:
	`DIM MyLRECL, myLRECL1 as NUMBER`
	`myLRECL = Get_Profile$("LRECL")`
	`myLRECL1 =`
	`val(Get_Profile$("LRECL")) + 1`

num-var **= Get_RBound**	
Operands:	none
Returns:	Returns the current Right bounds value. If the right bounds is MAX, 0 (zero) is returned. RC will always be 0
Special Notes:	Note that a return value of 0 here means that there is **no** maximum (the right boundary has no arbitrary limit), rather than implying that the right boundary were "column 0".

num-var **= Get_RC**	
Operands:	none
Returns:	The Return Code from the last executed SPFLite function call. The return codes will generally be 0, 4 or 8, where 0 = success, 4 = warning and 8 = failure. The specifics of the error can be obtained via Get_MSG$. In many cases, depending on the particular function, the RC may be returned directly from the function call as well. See the specifics for each function call.

	Get_Msg$ and **Get_RC** do **not** themselves modify the function-level RC or Message status values, but simply report them. That is, **Get_Msg$** and **Get_RC** do not have, or set, function-level status values of their own, but only return the last RC or Message value set by some **other** SPFLite interface function.
Special Notes:	

***num-var* = Get_Select_First_LPtr**		
Operands:	***index-num***	None
Returns:		Returns the line-ptr to the first line of a selected text block. If no block is currently selected, 0 (zero) will be returned. RC is always set to 0 and Msg$ will be "" (Null)
Special Notes:		

***num-var* = Get_Select_Last_LPtr**		
Operands:	***index-num***	None
Returns:		Returns the line-ptr to the last line of a selected text block. If no block is currently selected, 0 (zero) will be returned. RC is always set to 0 and Msg$ will be "" (Null)
Special Notes:		

***num-var* = Get_Select_Col**		
Operands:	***index-num***	None
Returns:		Returns the left-hand column number of the selected block. If no block is currently selected, 0 (zero) will be returned. RC is always set to 0 and Msg$ will be "" (Null)
Special Notes:		

***num-var* = Get_Select_Len**		
Operands:	***index-num***	None
Returns:		Returns the length of the current selected area. If no block is currently selected, 0 (zero) will be returned. RC is always set to 0 and Msg$ will be "" (Null)
Special Notes:		

***str-var* = Get_Session_Type$**	
Operands:	none
Returns:	Returns a string containing the 'type' of the current edit tab. This will be one of: BROWSE EDIT CLIP-EDIT SET-EDIT MEDIT RDONLY RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	This value could be saved and used, for example, to return the screen display back to it's original position using the Set_Csr_Pos if macro processing had caused it to be altered.

***str-var* = Get_SETVAR$(*set-var-str*)**		
Operands:	***set-var-str***	the name of the SPFLite SET variable for which the data is to be returned.
Returns:		Returns the string value for the specified SET variable named by ***set-var-str***. If an unknown variable name is specified, a "" (a zero-length string) will be returned. If a valid ***set-var-str***, RC will be 0 and Msg$ will be "" (Null) If an invalid ***set-var-str***, RC will be 8, and Msg$ will contain an error message

Special Notes:	For example, **`Get_SETVAR$("ABC")`** would return the value assigned to the SPFLite SET variable ABC.

`str-var` **`= Get_Src_LCmd$`**

Operands:	none
Returns:	Contains the line command used to mark the **source** line range for use by the macro. e.g. C/CC, M/MM or the macro name itself if this is a line command macro. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	For line-command macros, this is the macro name (possibly a plural/block form) of the macro name. For primary-command macros, this may be a C/CC or M/MM command, or may be an empty (null) string if no C/CC/M/MM was used with a primary-command macro.

`num-var` **`= Get_Src_Op`**

Operands:	none
Returns:	Contains the optional numeric line operand, if it is entered, for the source line range if used. If none is entered, this will return 0 (zero). For example, if the line command entered were C12, the value returned would be 12. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	For example, if you have a source line range of **C5**, **Get_Src_Op** will return the number **5**. The function returns 0 when no explicit command operand number is specified. For instance, if you define a source block such as CC which happens to be 5 lines long, the function will still return 0, because the number 5 was not actually specified on the CC block.

`str-var` **`= Get_Src_LMOD$`**

Operands: none	
Returns:	The line command modifiers when present. The **/** and **** line command modifiers are processed, if present, and this gets reflected in first and last LPTR values for the line range, but the **/** and **** codes themselves are not returned as LMOD values. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	Gets the **+** or **-** (post-unexclude or post-exclude) modifier and K (Keep) modifier if one is present on the source line range. The returned value is always two characters long. Position 1 will contain **+** or **-** or blank, and position 2 will contain a K or blank. The **/** and **** line command modifiers are processed, if present, and this gets reflected in first and last LPTR values for the line range, but the **/** and **** codes themselves are not returned as LMOD values.

***num-var* = Get_Src1_LPtr**	
Operands: none	
Returns:	Contains the line pointer of the first line in the selected line range marked for use by the macro. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***num-var* = Get_Src2_LPtr**	
Operands: none	
Returns:	Contains the line pointer of the last line in the selected line range marked for use by the macro. Note: Src1 and src2 may be the same for a single-line selection. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

str-var **= Get_Tag$(*line-ptr*)**		
Operands:	***line-ptr***	The line pointer of the data line whose line tag is to be obtained.
Returns:		If the ***line-ptr*** argument is a valid line pointer to a data line, the function will return the tag associated with that data line, if one exists; otherwise it will return a null (empty) string; the RC value will be set to 0 when the line-ptr is associated with a valid data line. The string value of the returned tag will be just as it appears in the edit session, with a leading : colon and one to five letters in upper case. If ***line-ptr*** is an invalid line pointer value, or is the line pointer of a non-data line, the function will return a null (empty) string and set RC to 8.
Special Notes:		

num-var **= Get_TopScrn_LPtr**	
Operands:	none
Returns:	Returns the line pointer of the line which is currently at the top of screen position. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	This value could be saved and used, for example, to return the screen display back to it's original position using the Set_Csr_Pos if macro processing had caused it to be altered.

num-var **= Get_Uniq_ID()**		
Operands:	***none***	There are no operands.
Returns:		Returns a unique ID number for this edit session. The number will be unique to the particular instance of SPFLite. No other Edit/Browse tab will ever duplicate the number. This can be useful for sets of related macros that want to store state type data and ensure the data is

	identified as belonging to a particular edit session.
Special Notes:	

***str-var* = Get_WORDCHAR$**	
Operands:	none
Returns:	Returns a string containing all the characters which are considered valid Word characters for the current Profile. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

***num-var* = Get_XLINES(line-ptr)**		
Operands:	line-ptr	The line pointer of the exclude marker for which the size of the exclude block is desired.
Returns:		If line-ptr is not a valid line, or not an Exclude line marker, then 0 **(zero)** will be returned, RC will be 8, and Msg$ set to an appropriate text message. If line-ptr is valid exclude marker, the function will return the number of excluded lines in the group, RC will always be 0 and Msg$ will be "" (Null)
Special Notes:		

***str-var* = Get_XSTATUS$(line-ptr)**		
Operands:	line-ptr	The line pointer of the data line whose exclude status is to be obtained.
Returns:		Returns a string containing either **X** or **NX** to indicate the line's exclude status. If line-ptr is not a valid line, then **"" (null)** will be returned, RC will be 8, and Msg$ set to an appropriate text message. If line-ptr is valid, then RC will always be 0 and Msg$ will be "" (Null)

Special Notes:

num-var **= Halt([*rc-num,*] *msg-str [,msg-str]* ...)**

Operands:	***[rc-num]***	this optional operand provides thee numeric return code you wish SPFLite to use for the success of this macro invocation. If the operand is omitted, then 0 (zero) is assumed. The conventions for these return codes are: RC=0 for success RC=4 for a warning condition RC=8 when a failure occurs There are predefined symbols for these values. OK is the same as 0; WARN is the same as 4, and FAIL is the same as 8. To avoid issues with future versions of SPFLite, only the values 0, 4 and 8 (or their symbolic equivalents of OK, WARN and FAIL) should be used as return code values.
	msg-str	a string containing the text of the message to be issued. When multiple strings are provided as operands, they will be concatenated together with a single blank between the operands.
Returns:		Nothing is returned. The macro is terminated by this function and control returned to normal SPFLite operation.

Special Notes:		If a message is issued with ***rc-num***=4, SPFLite will prefix it with "Macro warning: "
		If a message is issued with ***rc-num***=8, SPFLite will prefix it with "Macro failed: "
		
		Because the RC operand is optional, simple success messages can be issued with only the text string operand. For example:
		`Halt("Normal completion")`
		is functionally equivalent to:
		`Halt(OK, "Normal completion")`

num-var **= Is_Line_Cmd**

Operands:	none

Returns:	Returns TRUE (1) if the macro was launched as a line-command macro, by placing its name into the line sequence-area of one line (if used on a single line, or with an **n** or **/** or **** modifier) or two lines (if used in the block-mode form). Otherwise, FALSE (0) is returned. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:	

num-var **= Is_Primary_Cmd**

Operands:	none	
Returns:		Returns TRUE (1) if the macro was launched as a line-command macro, by placing its name into the line sequence-area of one line (if used on a single line, or with an **n** or **/** or **** modifier) or two lines (if used in the block-mode form). Otherwise, FALSE (0) is returned. RC will always be 0 and Msg$ will be "" (Null)
Special Notes:		

num-var **= IS_xxxx(*line-ptr*)**

Operands:	***line-ptr***	the line pointer of the line which is to be tested.
Returns:		Returns TRUE if the line at ***line-ptr*** is the indicated type, or FALSE if it is not or if the ***line-ptr*** is invalid. The IS_xxxx function exists in the following forms: `Is_Bnds(line-ptr)` - a BNDS line `Is_Bottom(line-ptr)` - the Bottom of Data line `Is_Cols(line-ptr)` - a COLS line `Is_Data(line-ptr)` - a normal text data line `Is_File(line-ptr)` - a File line `Is_Mark(line-ptr)` - a MARK line `Is_Mask(line-ptr)` - a MASK line `Is_Note(line-ptr)` - a NOTE line `Is_Prof(line-ptr)` - a =PROF>

```
                    line
                    Is_Tabs(line-ptr)       - a TABS line
                    Is_Top(line-ptr)        - the Top of
                    Data line
                    Is_ULine(line-ptr)      - a User line
                    (having a | mark)
                    Is_XLine(line-ptr)      - an excluded
                    line (usually, but
                                              not always,
                    a data line)
                    Is_XMarker(line-ptr)    - an excluded
                    line marker (the
                                              dashed
                    "placeholder" line)
                    Is_Word(line-ptr)       - a WORD line
```

RC will always be 0 and Msg$ will be "" (Null)

Special Notes:

***str-var* = Release_Label$(Label-ref | "ALL")**

Operands:	***label-ref***	the label reference which is to be released. Normally, this will be the label reference returned to the macro by the Request_Label$ function. If "ALL" is passed as the label-ref, it is a request to release all temporary labels.
Returns:		If *label-ref* is a normal temporary label returned by Request_Label$, then that label is removed from the line and the function returns a null string and RC = 0 If *label-ref* is a permanent label, that label is left alone on the line and is returned by the function along with a RC=0 If *label-ref* is a not defined, then the function returns a null string and RC = 8 If *label-ref* is the string "ALL", then all known temporary labels are removed. The function returns a null string and RC = 0

Special Notes:

***str-var* = Request_Label$(Line-ptr)**

Operands:	***line-ref***	the line pointer of the line for which a label is desired.

	This can be a string in the form `".123"` or `"!456"`
Returns:	If *line-ptr* points at a line which already has a label, then **that** label is returned by the function. RC=0 is set. If the line has no current label, a temporary label will be generated and returned. RC=0 will be set. Whether an existing label is returned, or a new temporary label is returned, you should only remove labels when finished by using the Release_Label$ function. By doing so, the macro need never care of know whether the label is temporary or not, and will be assured of never accidentally removing a permanent label.
Special Notes:	

num-var **= Reset_Gbl_Num([TblNum])**

Operands:	**TblNum**	Optional. The sub-table number to be cleared. If **NO** TblNum operand is entered, **ALL** sub-tables are cleared. Entering a **0** (Zero) is **NOT** the same as omitting the operand, it will simply clear sub-table 0.
Returns:		Clears the Gbl_Num storage area and returns True if successful, or False if the function fails.
Special Notes:		

num-var **= Reset_Gbl_Str([TblNum])**

Operands:	**TblNum**	Optional. The sub-table number to be cleared. If **NO** TblNum operand is entered, **ALL** sub-tables are cleared. Entering a **0** (Zero) is **NOT** the same as omitting the operand, it will simply clear sub-table 0.
Returns:		Clears the Gbl_Str storage area and returns True if successful, or False if the function fails.
Special Notes:		

num-var **= Set_Clr_Line(*line-ptr, clr-str*)**

Operands:	***line-ptr***	the line pointer for the data line to be replaced
	clr-str	the new color control string to be stored in the line referenced by ***line-ptr***.
Returns:		Both ***num-var*** and RC will be 0 and Msg$ will be "" (Null) for successful completion Both ***num-var*** and RC will be 8 and Msg$ set to an error message on failure (e.g. invalid line pointer)
Special Notes:		

num-var **= Set_Csr(*line-ptr, col-num, [len-num]*)**		
Operands:	***line-ptr***	the line pointer of the line where the cursor is to be placed.
	col-num	the column number where the cursor should be placed. If ***col-num*** is zero, the cursor is placed in the line number field of the given line.
	len-num	If a non-zero ***len-num*** is provided, then the data starting at ***col-num***, for ***len-num*** characters will be highlighted. When ***len-num*** is omitted, it is treated as 0; that is, the cursor will be positioned, but no highlighting will be performed.
Returns:		If ***line-ptr*** is valid, RC will be 0 and Msg$ will be "" (null) If ***line-ptr*** is invalid, RC will be 8 and Msg$ will contain an error message.
Special Notes:		Requests SPFLite to move the cursor to the specified line and column number.

num-var **= Set_Gbl_Num([TblNum,]*key-str, value-num*)**		
Operands:	***TblNum***	Optional. The sub-table number in which to store this value. If omitted, a default value of zero is assumed.
	key-str	a string value under which to store the numeric value
	value-num	the numeric value to be associated with ***key-str***.
Returns:		Always returns 0 to num-var and RC. Msg$ will be "" (Null)

Special Notes:		If a key with the value of ***key-str*** already exists, the prior value is replaced with the new ***value-num***. If the key does not already exist, it is created.

num-var **= Set_Gbl_Str([TblNum,]*key-str, value-str*)**		
Operands:	***TblNum***	Optional. The sub-table number in which to store this value. If omitted, a default value of zero is assumed.
	key-str	a string value under which to store the numeric value
	value-str	the string value to be associated with ***key-str***.
Returns:		Always returns 0 to num-var and RC. Msg$ will be "" (Null)
Special Notes:		If a key with the value of ***key-str*** already exists, the prior value is replaced with the new ***value-str***. If the key does not already exist, it is created.

num-var **= Set_Line(*line-ptr, data-str*)**		
Operands:	***line-ptr***	the line pointer for the data line to be replaced
	data-str	the new data value to be stored in the line referenced by ***line-ptr***.
Returns:		Both ***num-var*** and RC will be 0 and Msg$ will be "" (Null) for successful completion Both ***num-var*** and RC will be 8 and Msg$ set to an error message on failure (e.g. invalid line pointer)
Special Notes:		

num-var **= Set_Msg([*rc-num,] msg-str [,msg-str] ...*)**		
Operands:	***[rc-num]***	this optional operand provides thee numeric return code you wish SPFLite to use for the success of this macro invocation. If the operand is omitted, then 0 (zero) is assumed.

	The conventions for these return codes are: RC=0 for success RC=4 for a warning condition RC=8 when a failure occurs There are predefined symbols for these values. OK is the same as 0; WARN is the same as 4, and FAIL is the same as 8. To avoid issues with future versions of SPFLite, only the values 0, 4 and 8 (or their symbolic equivalents of OK, WARN and FAIL) should be used as return code values. ***msg-str*** a string containing the text of the message to be issued. When multiple strings are provided as operands, they will be concatenated together with a single blank between the operands.
Returns:	The ***rc-num*** value is returned. The value of ***msg-str*** is what will be returned by the next Get_Msg$ call.

Special Notes:	If a message is issued with ***rc-num***=4, SPFLite will prefix it with "Macro warning: "
	If a message is issued with ***rc-num***=8, SPFLite will prefix it with "Macro failed: "
	
	Because the RC operand is optional, simple success messages can be issued with only the text string operand. For example:
	`Set_Msg("Normal completion")`
	is functionally equivalent to:
	`Set_Msg(OK, "Normal completion")`
	
	It is important to understand that RC and message values passed back by other functions (like **Get_Line$**, for instance) are **not** automatically passed through to the edit user, even when the RC value is non-zero, signifying a warning or error condition was detected. You must explicitly call **`Set_Msg`** yourself if you wish a message to be displayed back to the user once the macro completes execution.
	
	If **`Set_Msg`** is followed by any other SPFLite-provided function which itself sets the RC and message values, the values set by **`Set_Msg`** will be overridden. If it is your intent to use **`Set_Msg`** to pass back a message to the edit user (yourself) that appears on the edit screen, the call to **`Set_Msg`** should be the last function you call before issuing a HALT statement. (Or use the HALT statement itself to issue the RC and message string)

num-var **= Set_SETVAR(*set-var-str, value-str*)**		
Operands:	***set-var-str***	a string containing the name of the SPFLite SET variable for which the data is to be changed.
	value-str	the new value to be assigned to the variable named by ***set-var-str***
Returns:		If successful, RC will be 0 and Msg$ will be "" (Null) If a failure, RC will be 8, and Msg$ will contain an error message
Special Notes:		For example, S**et_SETVAR("ABC", "New value")** would set the variable ABC to the string "New value".

num-var **= Set_TopScrn_LPtr(*line-ptr*)**		
Operands:	***line-ptr***	the line pointer for the line to be positioned at the top of screen when the macro exits.
Returns:		Both ***num-var*** and RC will be 0 and Msg$ will be "" (Null) for successful completion Both ***num-var*** and RC will be 8 and Msg$ set to an error message on failure (e.g. invalid line pointer)
Special Notes:		If your macro uses both Set_TopScrn_LPtr **and** Set_Csr, the positioning of Set_Csr will **override** that of the Set_TopScrn_LPtr **if** the desired cursor position is not within the visible screen as set by the Set_TopScrn_LPtr function. i.e. Set_Csr 'wins'.

num-var **= SPF_Cmd(*cmd-str [,cmd-str] ...*)**		
Operands:	***cmd-str***	the SPFLite primary command you wish executed. When multiple strings are provided as operands, they will be concatenated together with a single blank between the operands.
Returns:		RC will be set to the return code issued by the command. This value will also be returned in ***num-var***. Msg$ will be set to whatever message is issued by the command.
Special		The ***cmd-str*** expression is passed as a primary

Notes: command to SPFLite

Within the ***cmd-str***, all line references should be entered using the **dotted prefix**. e.g. the numeric values should be normal line pointers (**in string format**), preceded by a dot. Line 12 should be specified as **.12** and not just **12**.

The dotted notation is referred to as "temporary line labels" or "line number pseudo-labels", since this notation is an extension to standard ISPF label syntax.

See the section, "*Line Labels: Temporary Line Labels, also known as Line-Number Pseudo Labels*" in *Working with Line Labels* in the main Help documentation, for more information.

Because SPFLite primary commands do not deal with line pointers, any command expression string you create must contain line **numbers** and not line **pointers**. If you are working with a line pointer, you must ensure that the line pointer is both valid and a pointer to a data line, and then convert that line pointer to a line number; otherwise the command you create will not work. You can use the Is_Data function to confirm that a line pointer is valid and points to a data line, and Get_LNUM is used for the conversion.

For example, if you had a variable ***myLptr*** containing a valid line pointer value that points to a data line, and you wanted to build a DELETE command to delete that line, an SPF_CMD call to do that would look like this:

```
SPF_CMD("DELETE ." +
TSTR$(Get_LNUM(myLptr)))
```

If you wish to use a Line Pointer directly, rather than convert it to a Line Number, you would use the **!** exclamation-point notation instead of the **.** dot notation. The same command above could be specified as

```
SPF_CMD("DELETE !" + TSTR$(myLptr))
```

to achieve the same thing. This **!** exclamation-point notation works only inside of macros, and only for the SPF_CMD function. When you use this technique, you must be certain that the line pointer value used is for a data line and not for a special line. Otherwise, the command passed to SPF_CMD will not be executed, and the function will return RC = 8.

Note: Be aware of the difference between concatenating strings via macro syntax and concatenating strings using the multiple operand support in SPF_Cmd.

Example:

	`SPF_Cmd("DELETE !" + TSTR(myLptr))`
	would generate a command of:
	`DELETE !5`
	
	While:
	`SPF_Cmd("DELETE !", TSTR(myLptr))`
	would generate a command of:
	`DELETE ! 5`
	
	Note the 2nd version has added a blank separating the SPF_Cmd operands, which is **NOT** what is desired.

***num-var* = SPF_Debug(*msg-str [,msg-str] ...*)**	
Operands:	***msg-str*** the text string you wish displayed. When multiple strings are provided as operands, they will be concatenated together with a single blank between the operands.
Returns:	Always RC = 0 and Msg$ = "" (null)

Special Notes:	During macro development, the SPF_Debug statement can be used to display information related to the macro execution. This can be simple trace messages, or displays of internal variables.
	
	Examples:
	
	`SPF_DEBUG ("Starting line loop")`
	`SPF_DEBUG ("Var_b=" & TSTR$(Var_b))`
	
	The SPF_DEBUG statement will open a console window in which to display the information.

***num-var* = SPF_Exec([{SYNC|ASYNC},][{HIDDEN|NORMAL},] *cmd-str [,cmd-str] ...*)**		
Operands:	***SYNC* | *ASYNC***	This optional operand indicates how you want the command to run. If SYNC is coded, control will not be returned to the macro until the command is complete. If ASYNC is coded, the command will be issued and control immediately returned to the macro.
	HIDDEN* | *NORMAL	This optional operand indicates whether you want the command to run in a visible window or not. If HIDDEN is coded, no window will be seen. (And if errors occur, they will not be visible either) If NORMAL is coded, the command will run in a normally visible command window.

	cmd-str the command you wish executed. When multiple strings are provided as operands, they will be concatenated together with a single blank between the operands.
Returns:	RC will be set to the return code from the executed function. This value will also be returned in ***num-var***. Msg$ will be set to "" (null).
Special Notes:	Both `SYNC|ASYNC` and `HIDDEN|NORMAL` are optional operands, and if omitted, `SYNC, NORMAL` will be assumed. **NOTE:** If you want to specify the `HIDDEN|NORMAL` operand, you **MUST** specify the `SYNC|ASYNC` operand as well, it can **not** be omitted. The ***cmd-str*** string expression is passed as a program to be executed. This operates similar to the SPF_Shell function, except that the command does not run under the command interpreter (CMD.EXE) but is executed directly. The usual issues about locating the program using the current working directory and the system PATH variable apply to calls to SPF_EXEC. Invoking programs to run under CMD.EXE can require complicated quoting and double quoting of operands when things like filenames with embedded blanks are required. With SPF_Exec only simple quoting is required, making the building of the string expression for the command much simpler. The function SPF_Quote$ may be of assistance in creating properly quoted operand values. Because such commands can perform powerful functions (such a deleting files), care should be taken to issue these commands correctly.

num-var **= SPF_Exempt_File(file-ext)**	
Operands:	***file-ext*** This specifies a single file extension. It may contain the leading period or no.
Returns:	RC=0 will be set if the extension is **NOT** in the 'exempt file' list, RC=8 will be set if the extension **IS** in the 'exempt file' list.
Special Notes:	SPFLite maintains a list of file extension which are to be considered as **non-text** files. For example this exempts these files from being searched by the FF (Find in Files) command. The list may be altered in the Options - File Manager options.

	This function allows you to determine if a particular file is contained in this list.

***num-var* = SPF_INS(*line-ptr, col-num, data-str*)**		
Operands:	***line-ptr***	the line pointer for the data line to be modified
	col-num	the column number at which the ***str-var*** contents are to be inserted.
	data-str	the text string to be inserted
Returns:		Both ***num-var*** and RC will be 0 and Msg$ will be "" (Null) for successful completion Both ***num-var*** and RC will be 8 and Msg$ set to an error message on failure (e.g. invalid line pointer)
Special Notes:		Will insert s*tr-var* into the text of ***line-ptr*** at the indicated column ***col-num***. Current text from ***col-num*** to the end of line will be shifted right to allow insertion of ***data-str***. If needed, the line's text will be extended with blanks to ***col-num - 1*** before ***data-str*** is copied in.

***num-var* = SPF_Loop_Check(*num-var*)**		
Operands:	***option-num***	A numeric value that sets the state of the loop-checking option. This should be TRUE or FALSE depending on setting desired. You may also specify ON or OFF, or 1 or 0.
Returns:		The function will return the previous setting for Loop Check prior to establishing the new value. RC is set to 0 and Msg$ = is set to "" (null)
Special Notes:		If your macro is going to perform some long running function, such as interfacing with the user by prompting, or any other long-running function, you should call this function to disable the normal SPFLite monitoring of macro execution. This will prevent SPFLite from popping up and informing you that the macro might possibly have gone into a loop. Disabling the loop check should not normally be done during macro development and testing, but only after you are confident your macro is running properly. If a macro **does** go into a loop after loop-checking is disabled, the only recourse is to cancel the entire SPFLite session.

When you issue any **thinBasic** function that causes a wait condition, such as the **MsgBox** function, SPFLite will detect a loop condition in progress, when in reality nothing is wrong. To avoid such problems, you would disable loop checking before issuing a function call like **MsgBox**, and re-enable loop checking once **MsgBox** has completed.

***num-var* = SPF_OVR(*line-ptr, col-num, data-str*)**		
Operands:	***line-ptr***	the line pointer for the data line to be overlaid.
	col-num	the column number at which the ***data-str*** contents are to be overlaid.
	data-str	the new contents of the data line to be overlaid
Returns:		Both ***num-var*** and RC will be 0 and Msg$ will be "" (Null) for successful completion Both ***num-var*** and RC will be 8 and Msg$ set to an error message on failure (e.g. invalid line pointer)
Special Notes:		Will overlay text in ***line-ptr*** with ***data-str*** starting at the indicated column ***col-num***. Other text in the line is unaffected. If needed, the line's text will be extended with blanks to ***col-num - 1*** before ***data-str*** is copied in. The overlay operation consists of replacing characters from ***data-str*** into the same relative location in the line's text **only when** there are blanks in the original data. Compare the functionality of SPF_OVR to the O/OO edit line commands.

***num-var* = SPF_OVR_REP(*line-ptr, col-num, data-str*)**		
Operands:	***line-ptr***	the line pointer for the data line to be overlaid. Line-ptr is a normal Internal line pointer.
	col-num	the column number at which the ***data-str*** contents are to be overlaid.
	data-str	the new contents of data line to be overlaid
Returns:		Both num-var and RC will be 0 and Msg$ will be "" (Null) for successful completion Both num-var and RC will be 8 and Msg$ set to an error

	message on failure (e.g. invalid line pointer)
Special Notes:	Will overlay replace text in ***line-ptr*** with ***data-str*** starting at the indicated column ***col-num***. Other text in the line is unaffected. If needed, the line's text will be extended with blanks to ***col-num - 1*** before ***data-str*** is copied in. Overlay replace consists or replacing characters from ***data-str*** into the same relative location in the lines text **only when** the characters in ***data-str*** are non-blank. i.e. similar to the previous SPF_OVR except here non-blanks in ***data-str*** take precedent over the original text. In SPF_OVR, non-blanks in the original text take precedent. Compare the functionality of SPF_OVR_REP to the OR/ORR edit line commands.

num-var **= SPF_PARSE(** ***TextLit-num, NumLit-Num, LinRef-num, TagOp-Num,***
[***keyword-set,*** **]**
[***keyword-set,*** **]**
...)

Operands:	***TextLit-Num***	Specifies the number of Text Literals allowed along with optional validation flags (see Special Notes below). If no special options are provided, the value specifies the fixed number of this operand which must be entered.
	NumLit-Num	Specifies the number of Numeric Literals allowed along with optional validation flags (see Special Notes below). If no special options are provided, the value specifies the fixed number of this operand which must be entered.
	LinRef-Num	Specifies the number of Line References allowed along with optional validation flags (see Special Notes below). If no special options are provided, the value specifies the fixed number of this operand which must be entered.
	TagOp-Num	Specifies the number of Tag Operands allowed along with optional validation flags (see Special Notes below). If no special options are provided, the value specifies the fixed number of this operand which must be entered.
	Keyword-set	Defines a set of allowable keywords. The format of a Keyword set is: `"[list-name:]keyword,keyword,(kwalias,kwalias),keyword,..."`

	list-name — a name to refer to this set of keywords, to be used in the SPF_Arg_KWGroup function. *keyword* — an allowable keyword. If only one keyword is specified, it is treated as a 'present' or 'not present' keyword type. *keyword list* — If multiple keywords are specified, they are treated as a set of mutually exclusive keywords. *alias list* — if a keyword entry is a bracketed list of keywords, the bracketed list are treated as aliases of each other. When retrieving information about these keywords, the left-most one in the bracketed list is treated as the preferred, normalized answer desired.
Returns:	Both num-var and RC will be 0 and Msg$ will be "" (Null) for successful parse where no errors are detected. Both num-var and RC will be 8 and Msg$ set to an error message on failure (e.g. invalid operand, missing operand, etc.)
Special Notes:	A full discussion of using SPF_Parse will be found in Full Parse Access **Validation Flags** Validation flags, when used, are entered by **adding** then to the specified numeric value. e.g. If 0 to 3 entries are desired, it would be entered as **3 + ARG_VAR** if these were tag entries which should be validated, it would be **3 + ARG_VAR + ARG_DEF** When none of the following flags are used, the number entered becomes the required number of that particular operand. ARG_VAR — When this is specified, Parse will allow from 0 to the specified number to be entered. ARG_OPT — When this is specified, Parse will allow either 0 or the exact specified number of

	operands ARG_DEF When this is specified, the Line Reference and Tag Operands are validated to ensure they are valid, defined references.

***str-var* = SPF_Quote$(*data-str*)**	
Operands:	***data-str*** a string value to be enclosed in quotes
Returns:	the value of ***data-str*** enclosed in quotes RC will be 0 and Msg$ will be "" (Null).
Special Notes:	The contents of ***data-str*** are examined. If ***data-str*** is already in the format of a string enclosed in properly-matched, valid SPFLite quote characters of any type (either " double quotes, ' single quotes, or ` accent quotes), and ***data-str*** does not contain inner quote characters as data values that are the same as the enclosing quote characters, the contents of the ***data-str*** expression are returned as-is without modification. If ***data-str*** does not contain any " double quote characters as data characters, the function returns the value of ***data-str*** preceded and followed by " double quotes. If ***data-str*** already contains " double quote characters as data characters, but does not contain any ' single quotes as data characters, the function returns the value of ***data-str*** preceded and followed by ' single quotes. If ***data-str*** already contains " double quote and ' single quote characters as data characters, but does not contain any ` accent quotes as data characters, the function returns the value of ***data-str*** preceded and followed by ` accent quotes.

***num-var* = SPF_REP(*line-ptr, col-num, data-str*)**		
Operands:	***line-ptr***	the line pointer of the data line to be modified.
	col-num	the column number at which the ***data-str*** contents are to be replaced.

	data-str	the replacement text string
Returns:		Both num-var and RC will be 0 and Msg$ will be "" (Null) for successful completion Both num-var and RC will be 8 and Msg$ set to an error message on failure (e.g. invalid line pointer)
Special Notes:		Will replace text in ***line-ptr*** with ***data-str*** starting at the indicated column ***col-num***. Other text in the line is unaffected. If needed, the line's text will be extended with blanks to ***col-num - 1*** before ***data-str*** is copied in. The replacement operation consists of replacing characters from ***data-str*** into the same relative location in the line's original text.

num-var **= SPF_Shell([{SYNC|ASYNC},][{HIDDEN|NORMAL},]** ***cmd-str [,cmd-str] ...*****)**

Operands:	***SYNC*** **|** ***ASYNC***	This optional operand indicates how you want the command to run. If `SYNC` is coded, control will not be returned to the macro until the command is complete. If `ASYNC` is coded, the command will be issued and control immediately returned to the macro.
	HIDDEN **|** ***NORMAL***	This optional operand indicates whether you want the command to run in a visible window or not. If `HIDDEN` is coded, no window will be seen. (And if errors occur, they will not be visible either) If `NORMAL` is coded, the command will run in a normally visible command window.
	cmd-str	the command you wish executed by the CMD.EXE shell. When multiple strings are provided as operands, they will be concatenated together with a single blank between the operands.
Returns:		RC will be set to the return code from the Shell function. This value will also be returned in ***num-var***. Msg$ will be set to "" (null).
Special Notes:		Both `SYNC|ASYNC` and `HIDDEN|NORMAL` are optional operands, and if omitted, `SYNC, NORMAL` will be assumed. **NOTE:** If you want to specify the `HIDDEN|NORMAL` operand, you **MUST** specify the `SYNC|ASYNC` operand as well, it can **not** be omitted. The cmd-string expression is passed as a Windows system command to the Command Shell. The command

	will be performed, and control returned. RC will be set to the Errorlevel / Exit code issued by the specified command. The function SPF_Quote$ may be of assistance in creating properly quoted operand values. SPF_Shell can be used for entering Windows shell commands like DEL, MKDIR, RMDIR, etc. Because such commands can be powerful (including commands that may delete files), care should be taken to issue these commands correctly.

***num-var* = SPF_Trace(*mode-num*)**		
Operands:	***mode-num***	Mode may be ON, OFF or ERROR. When ON, a debugging window will appear that shows a trace record for every SPFLite function called, including the parameters it was passed, its RC and any message text generated. When ERROR, the debugging information will be recorded only when an SPFLite function is called that does not return an RC of 0.
Returns:		RC will be set 8 if you supply an invalid ***mode-num*** operand, otherwise RC will be 0.
Special Notes:		The **SPF_Trace** call is itself always traced, regardless of the old or new trace mode in effect.

***str-var* = SPF_UnQuote$(source-str)**		
Operands:	***source-str***	The string to be 'un-quoted'. UnQuote$ will remove the outer quotes from a string, if it is indeed quoted. Unquoted source strings are returned unchanged. The outer quotes may be any of the standard SPFLite supported quotes. i.e. Single quotes ' - Double quotes " or Back quotes `
Returns:		RC will always be 0 and Msg$ = "" (Null)
Special Notes:		

Debugging your Macro

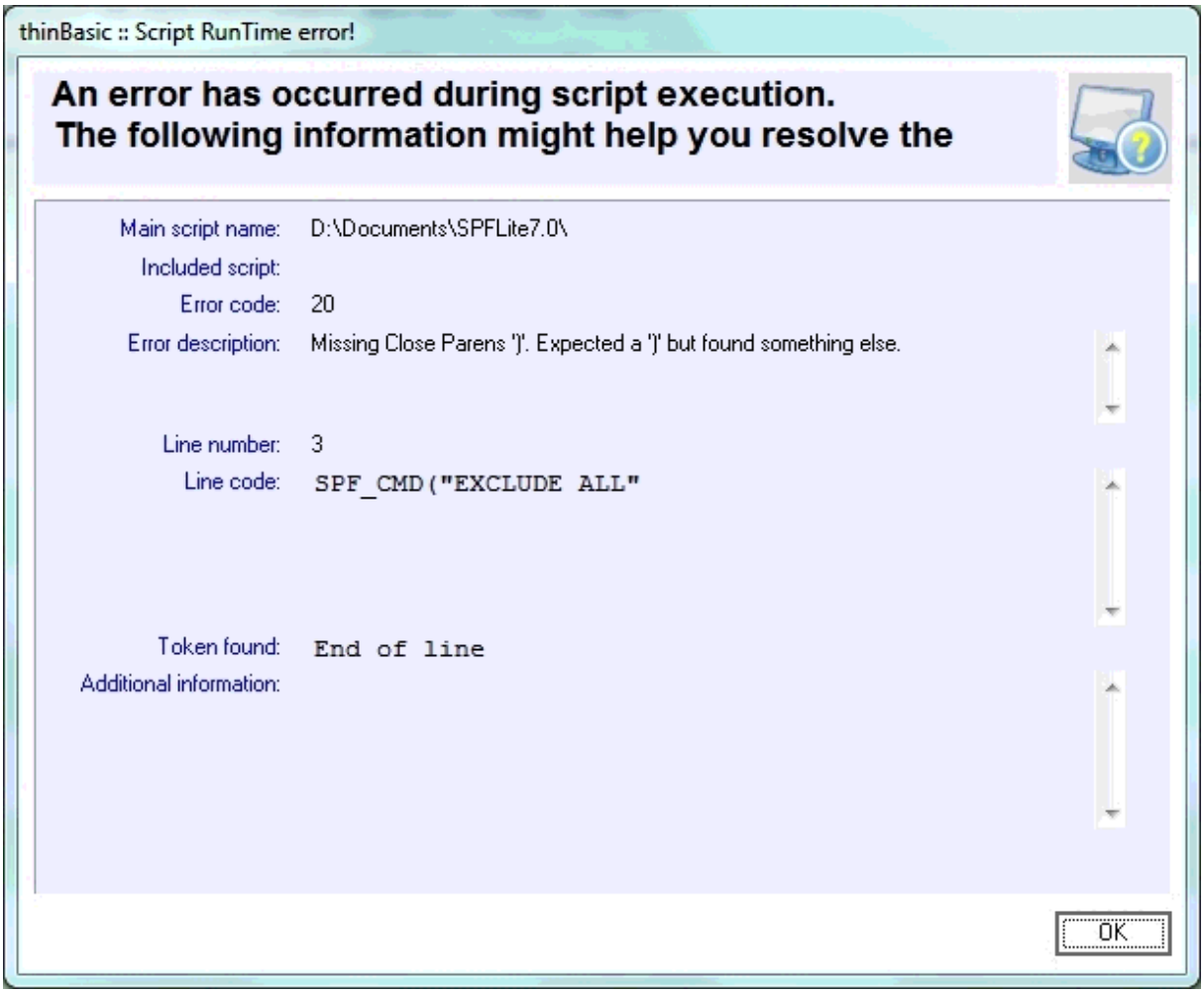
Contents of Article

Introduction

Just as with any programming exercise, your macro may not to do just what you want on the first attempt.

So, what types of errors are you likely to see? **Simple syntax errors** are usually the simplest to correct.

thinBasic provides you with a detailed error display like this:



This message tells you what **thinBasic** expected to find - a closing parenthesis - and what it found instead - an End of line.
Once you add a closing) to line 3 this problem is fixed.

However, there are some errors that thinBasic detects that result in misleading and confusing messages:

- Duplicating reserved words as variable names can trigger strange errors. If the error is reported immediately following a DIM statement that declared variables, review the variable names you have used.
- Delimiter parsing errors (expecting a closing parenthesis and not finding it) will sometimes describe the 'found' tag as some value from the previous valid statement.
- Other missing syntax, like a missing close-quote, will cause similar confusing messages

These can be frustrating, but the line number flagged as in error is usually where the problem line is, or is close to it.

The other error most commonly made is misspelling of function names, and a quick browse of the Help file should provide the correct spelling. An easy mistake to make is failing to put a **$** for a string-based function like **Mid$**. If you do that, you'll get an error message about code that "seems" to be correct but isn't. Similar to misspelled names is failing to define a variable before you used it.

Tracking your macro's flow of control

A common debugging problem is code that does not follow the path you think it should. The macro may just end prematurely, or seems to be in an area of code that it should not be, and you have to figure out: How did the macro end up **there?**

The function `SPF_Debug` is useful in a variety of ways in solving such problems. It allows you to write messages to a debug log . These messages can be anything from simple 'breadcrumb' messages, like:

```
SPF_Debug("Starting line loop")
SPF_Debug("Parameters validated")
```

to displays of macro variables, which can be done like this:

```
SPF_Debug("Operand1=" + Operand1)
```

Functions and Return Codes

All of the SPFLite support functions will (in addition to the defined values returned by the functions) set a Return Code value and a Return Message value. You can get these values using the functions Get_RC and Get_Msg$.

Good programming practice suggests that after calling any function that indicates success or failure with a return code, the return code should be tested and acted on. In a perfect world that's a good idea, but in practice we usually only check a few of the results of function calls when it seems important. And, many of the SPFLite interface calls are simple information

retrieval calls where the functions never (or hardly ever) fail, and checking a return code would be unnecessary.

In addition, using nested function calls, like `Get_LPtr(Get_Arg$(1))` would be very burdensome if we had to break down each function call into single steps and add RC tests. But, when something is not working, and you suspect a failing function call, what option do we have, other than breaking things down into single-step code?

SPFLite Function Trace

The answer is to activate function tracing using `SPF_Trace`, which is called as follows:

```
SPF_Trace( [ ON | OFF | ERROR ] )
```

If you insert an `SPF_Trace(ON)` call at the beginning of your macro code (after the macro prototype line) the macro processor will create a debug log window, and write a trace line for every SPFLite interface function call. It will show the function call and its calling parameters, along with the resultant Return code and Error message text generated.

Lets look at an example. Given the following macro:

```
' Try.MACRO aaa bbb ccc ddd eee fff ggg hhh
SPF_Debug("Args: " + Get_Arg$(4, 2))
halt
```

The macro writer tried to display some of the calling arguments using an `SPF_Debug` display function. But when it was run, all that displays is "`Args:`" and nothing else. Obviously, it seems like `Get_Arg$` is not working, but why? And how can we be be sure?

We could track this down the down the "hard way" with a brute-force approach, and break the macro down to individual statements, adding code to fetch and display error messages and return codes at every point - but that would be a lot of work. Instead, let's try using `SPF_Trace` and see how it can help. We add the `SPF_Trace` activation call, and the macro now looks like this:

```
' Try.MACRO aaa bbb ccc ddd eee fff ggg hhh
SPF_Trace(ON)
SPF_Debug("Args: " + Get_Arg$(4, 2))
halt
```

When run, the trace output looks like:

```
Get_Arg$(4, 2) RC=8 Invalid ARG indexes
Args:
SPF_Debug(Args: ) RC=0
```

There in the first line is our problem. The Get_Arg$ is failing. Now we need to determine why. (By the way, the actual log won't highlight the display in yellow like this. We just added the color in the documentation here to make it stand out.) Reviewing the documentation for the Get_Arg$ function reveals the problem. The two arguments are reversed; we needed to say (2, 4) and not the other way around.

> Moral of the story: You have to read the manual.

What are the other two lines?

> The `Args:` line is the output from the `SPF_Debug()` function, Debug and Trace share the same debugging window.
> The last line is the trace output for the **execution** of the `SPF_Debug()` showing that **it** worked correctly (RC=0).

The Trace function can be turned ON and OFF selectively throughout your macro, which is handy when you can isolate the problem area to one section. If you don't need to trace an entire macro's execution, it will make the debugging output smaller, which makes it easier to follow.

The ERROR option requests that trace records need only be displayed when a function's return code is non-zero. This can reduce the volume of output down considerably, by turning the debugging output into "exception reporting", since it's usually more important to know when functions failed than when they succeeded.

Between `SPF_Trace` and `SPF_Debug`, it usually becomes simple to nail down the cause of most bugs.

thinBasic Output Statements

If you wish, you can use **thinBasic**'s output facilities to write debugging logs or any other information you wish. To do this, you must have the CONSOLE **thinBasic** module available at runtime.

You should place a statement of the form,

```
USES "CONSOLE"
```

prior to writing to the **thinBasic** console screen.

The "print" statements are slightly different than other versions of Basic you may know. They do not use semi-colons to influence the formatting of the output.

The **`PRINT`** statement will output one or more (comma-separated) items, but will **not** write an end-of-line termination (no Carriage Return and Line Feed).

The **`PRINTL`** statement ("print line") will output one or more (comma-separated) items, and **will** write an end-of-line termination.

You may also issue a **`CLS`** statement to clear the screen.

There are additional Console capabilities which you may find useful. See the **thinBasic** documentation under **thinBasic Modules - Console** for more information.

The thinBasic Trace Module

If you read the **thinBasic** documentation, you will notice in the Modules section they mention a Trace module, which for stand-alone **thinBasic** use provides a single-step debugging mode.

Unfortunately, the Trace module cannot be used presently in SPFLite Macro mode. If you

attempt to use this, crashes will occur. When this problem is resolved and **thinBasic** Trace functions correctly, we will announce its availability.

Working with The Interface

There are many common coding requirements which will be needed by many macros. Below are small common coding techniques showing how these various requirements can be handled. These are **not** full, complete macros but only small code blocks showing one particular aspect.

Fragments include:

Accessing macro operands as Line Range operands

Assume a macro which accepts two operands, a starting and an ending line range.

```
dim fromlptr, tolptr as number
fromlptr = Get_LPtr(Get_Arg$(1))           ' Fetch from
operand
tolptr = Get_LPtr(Get_Arg$(2))             ' Fetch to
operand
if fromlptr = 0 or _                       ' Validate
what we got
   tolptr   = 0 or _                       '
   fromlptr > tolptr then                  '
   Halt(FAIL, "From / To line pointers missing or
invalid")
end if
```

Note: The use of `Get_LPtr` to convert the arguments to internal line pointer format. After this code the `fromlptr` and `tolptr` variables are all ready to be used in further processing code. The **if** statement uses line continuation characters _ to format the statement as one test per line. (Without the underscore, each **thinBasic** statement ends on the line it starts on. This is a standard BASIC coding convention.)

Macro support also allows a more structured form which automatically fetches command operands. Here's the above sample done in that structured format:

```
' MyMacro.MACRO
SUB MainLine(FromArg as string, ToArg as string)
dim fromlptr, tolptr as number
fromlptr = Get_LPtr(FromArg)                ' Convert the
from operand
tolptr = Get_LPtr(ToArg)                    ' Convert the
to operand
if fromlptr = 0 or _                        ' Validate
what we got
   tolptr   = 0 or _                        '
   fromlptr > tolptr then                   '
   Halt(FAIL, "From / To line pointers missing or
invalid")
end if                                      '
END SUB                                     ' End of
MainLine SUB
```

And a third method is also available, particularly useful with macros that may have many operands available. See Full Parse Access for details.

Accessing all macro operands as a list

Assume a macro which accepts any number of operands, and processes each one at a time.

```
dim OpIndex as number
dim Operand as string

if Get_Arg_Count = 0 then halt(FAIL, "No operands are
present")

for OpIndex = 1 to Get_Arg_Count            ' Loop
through operands
   Operand = Get_Arg$(OpIndex)              ' Fetch the
operand
   ... process each Operand value ... ' Process it
next
```

Note: This uses a simple FOR / NEXT loop to fetch and process each operand.

Modifying data in a text line

Assuming the macro has a valid line pointer to a specific data line, there are a variety of methods to change the text.

Replace the entire line

This is done with the **Set_Line** function.

Set_Line(*line-ptr, replacement-text*)

Which will completely replace any existing text in the line with the *replacement-*

text value

Replace a substring of the text

This is done with the **SPF_REP** function

SPF_REP(*line-ptr, column, replacement-text*)

Which will completely replace a portion of the existing text, starting at *column* with the *replacement-text*. If the existing text is less than *column* characters long, it will be first extended with spaces.

Insert a text string into the existing text

This is done with the **SPF_INS** function

SPF_INS(*line-ptr, column, insert-text*)

Which will insert a string (*insert-text*) into existing text, following the *column* position. If the existing text is less than *column* characters long, it will be first extended with spaces.

Overlay a text string into the existing text

This is done with the **SPF_OVR** function

SPF_OVR(*line-ptr, column, overlay-text*)

The SPF_OVR function will overlay a character string within a data line starting at a specified column. Overlay is done by comparing relative characters within the overlay area and then copying characters from the provided string to the data line only if the same relative character in the data line is a blank. This is identical in function to the SPFLite line commands **CC** / **OO**.

Overlay Replace a text string into the existing text

This is done with the **SPF_OVR_REP** function

SPF_OVR_REP(*line-ptr, column, overlay-text*)

The SPF_OVR_REP function will overlay a character string within a data line starting at a specified column. Overlay replace is similar to Overlay (above) except that Overlay Replace gives priority to the **new** character string. The Overlay is done by comparing relative characters within the overlay area and then copying characters from the provided new string **which are non-blank** to the data line regardless of the contents of the data line. Blanks in the new character string are not copied, thus leaving the original data line characters untouched. This is identical in function to the SPFLite line commands **CC** / **ORR**.

Setting a final Return Code and Message from a Macro

When a macro completes, it is often useful to have SPFLite issue a status message to indicate the relative success or failure of the macro processing. For some macros, this message might be the only desired output (for example, from a word-counting macro). This can be done by either of two very similar functions. `Set_Msg` and `Halt`. The difference is that `Set_Msg` establishes the RC and message values and macro processing continues. `Halt` establishes the values and terminates the macro immediately.

```
Set_Msg("This macro ran successfully")

Set_Msg(0, "The number of words in the file is: " +
TSTR$(wordctr))

Halt(4, "There were no strings found")

Halt(FAIL, "A required operand is missing")
```

The first operand of `Set_Msg/Halt` is optional, but if provided, and the value is numeric, it is assumed to be the desired return code for the macro, and the second is the message string.

If the first operand is a string operand, it is assumed to be the message text, and a default Return Code value of 0 (zero) is assumed.

Note the first example above uses the default RC format, the other three examples provide specific RC values.

A return code of **0** means the macro was successful; **4** means you wish to issue a warning, and **8** means that the macro has "failed", based on your definition of what you mean for the macro to fail.

Rather than hard-coding these values, you can specify them symbolically, using **OK** for 0, **WARN** for 4 and **FAIL** for 8.

For non-Zero RC values, SPFLite will format the value and prefix the message text with **RC=nn:** e.g. `RC=8: Missing search string`

For compatibility with future releases of SPFLite, the return code operands you use should only contain 0, 4 or 8, or the symbolic names for them.

Setting a new cursor location to be used when the macro ends

You may optionally set a new cursor location to be used on macro exit. If you do not, the cursor will remain where it was when the macro started **or** where some other SPFLite command positioned it, if SPFLite command(s) were invoked by the macro.

The Set_Csr function provides the line pointer, the column number, and an optional length value.

If the column number provided is zero, the cursor will be placed in the line number area of the line.

If the optional length parameter is zero, no text will be highlighted.

If the optional length parameter **is** provided, then the text at the cursor location, for the specified length, will be highlighted.

```
Set_Csr(line-ptr, column, 0)    ' Cursor to line-ptr,
column, with no highlighting

Set_Csr(line-ptr, 0)            ' Cursor to the line
number area of line-ptr

Set_Csr(line-ptr, column, 5)    ' Cursor to line-ptr,
column, with a 5 chars highlighted
```

Perform a simple action on all lines in a file

Assume a portion of macro which wants to perform an action on all lines in the file.

```
dim i as number

for i = Get_First_Lptr to Get_Last_Lptr       ' Loop
through the file
   if Is_Data(i) then                         ' Doing
only data lines
      Set_Line(i, ucase$(Get_Line$(i)))       ' SAMPLE
ACTION: uppercase the text
   end if                                     '
next                                          '
```

Note: This uses a simple FOR / NEXT loop to process each line. An `Is_Data` test ensures we only process text data lines rather than on special lines. The action performed here is just a simple uppercase function on each line's data.

An alternate method using the `Get_Next_LPtr` function could be:

```
dim i as number value 1

i = Get_Next_LPtr(i, 1, "DATA")           ' Adjust i to
the next DATA line
do while istrue i                         ' If not end
of file
   Set_Line(i, ucase$(Get_Line$(i)))      ' SAMPLE
ACTION: uppercase the text
   i = Get_Next_LPtr(i, 1, "DATA")        ' Adjust i to
the next DATA line
loop                                      ' loop-de-loop
```

Starting in SPFLite version 8.4, you can write code to access all lines in a file using a simplified syntax. The **WHEN** option allows the check for the correct type of line, right on the **FOR** statement. It operates just like the code above, without requiring to have an **IF** statement nested inside the **FOR**:

```
dim i as number

for i = Get_First_Lptr to Get_Last_Lptr WHEN Is_Data(i)
   Set_Line(i, ucase$(Get_Line$(i)))       ' SAMPLE
ACTION: uppercase the text
next
```

Note: Thanks to Eros Olmi of *thinBasic* for adding this support to his Basic engine for us.

Use the FIND command to locate text lines to process

To locate specific lines to process using the FIND command use the following code. Assume the string to search for is already set up in the variable `lookfor`.

```
SPF_Cmd("FIND FIRST " & lookfor)                    '
Issue 1st FIND cmd
do while Get_RC = 0                                 '
while found
   IF other tests needed to 'qualify' this record for
processing
   ... process the found record - - -               '
Process it
       ... The line's text can be obtained with
       ... Get_Line$(Get_Find_LPtr))
       ... The found string will be at the column
provided by
       ... Get_Find_Col
   end if                                           '
   SPF_Cmd("RFIND")                                 '
Look some more
loop                                                '
```

Note: The ...process the found record ... code can use the data from `Get_Find_LPtr`, `Get_Find_Col` and `Get_Find_Len` functions since the code is only executed following a successful FIND command (Get_RC = 0).

Locate text lines to process using non-FIND criteria

To locate specific lines to process using some other criteria than what the FIND command can provide, use the following code.

```
dim i as number
for i = 1 to Get_Last_LPtr                          '
Let's search
   if is_Data(i) then                               '
Only data lines
      if ... special criteria test ... then         '
Examine the line
         ... process the line - - -                 '
      end if                                        '
   end if                                           '
next i                                              '
```

Note: The ...process the found record ... code can use the data from `Get_Find_LPtr`, `Get_Find_Col` and `Get_Find_Len` functions since the code is only executed following a successful FIND command (Get_RC = 0).

Insert a New Text line into the File

This sample assumes the macro has already determined the line-pointer after which a line is to be inserted. We will assume this is in the variable `curr_Lptr`.

```
        SPF_CMD("LINE N1 !" & TSTR$(curr_Lptr)) ' Insert a
line after curr one

        Set_Line(curr_Lptr + 1, "New text for the new line")

        INCR curr_Lptr                          ' Adjust
curr_Lptr for the inserted line
```

Note: The new text line is inserted using the primary command **LINE**. The Set_Line function adds the new text, and uses **Curr_Lptr + 1** for the line number, since that will be the new line pointer for the inserted line. The third line, which increments the **Curr_Lptr** value, is what would probably be needed in most macros to adjust the line pointer value for the inserted line. That's because **Curr_Lptr** points to the "current" line, and the line inserted *after* the current line would have a line pointer one greater than the current one. Whether this would actually be needed is of course dependent on the design of the logic flow in the specific macro.

If you wish to use the Line **Number** value directly (rather than a **pointer**), the first line of this example can be specified using the . (period) notation like this:

```
        SPF_CMD("LINE N1 ." & TSTR$(Get_LNUM(curr_Lptr)))
```

Search for and process all lines with a specific Tag ID

This sample locates all lines with a Tag of **:ABC** and processes them.

```
dim i as number value 1
dim tagname as string value ":ABC"

i = Get_Next_LPtr(i, 1, tagname)          ' Adjust i to
the next :ABC tagged line
do while istrue i                         ' If not end
of file
   Set_Line(i, ucase$(Get_Line$(i))       ' SAMPLE
ACTION: uppercase the text
   i = Get_Next_LPtr(i, 1, tagname)       ' Adjust i to
the next DATA line
loop                                      ' loop-de-loop
```

Locate 'Problem' lines and add a NOTE line to mark them

This sample will locate lines which match some unique criteria, and insert a NOTE line following the line to flag the problems.

```
dim cLPtr as number value 1


while cLPtr < Get_Last_LPtr                                    '
Loop through all lines
   if Is_Data(cLPtr) then                                      '
Only do data lines
      if ...special criteria one ... then                      ' A
problem line?
         SPF_CMD("LINE NOTE !" + TSTR$(cLPtr))                 '
Insert a NOTE line after this one
         Set_Line(cLPtr + 1, "Check this line - special
criteria one")
         incr cLPtr                                            '
Adjust cLPtr for the inserted line
      elseif ...special criteria two ... then                  ' A
different problem line?
         SPF_CMD("LINE NOTE !" + TSTR$(cLPtr))                 '
Insert a NOTE line after this one
         Set_Line(cLPtr + 1, "Check this line - special
criteria two")
         incr cLPtr                                            '
Adjust cLPtr for the inserted line
      end if                                                   '
End of special tests
   end if                                                      '
End of Is_Data tests
   incr cLPtr                                                  '
Bump to next line
wend
```

Using Global storage to communicate between Macros

In this sample, Macro A saves the location of the cursor when it is invoked in Global storage. Later, another macro Macro B wishes to return the cursor to the original location stored by Macro A.

Macro A

```
'--- Save where we are for Macro B, use key names
MacALine and MacACol for global storage pool
Set_Gbl_Num("MacALine", Get_LNum(Get_Csr_LPtr))      '
Save where the cursor is
Set_Gbl_Num("MacACol", Get_Csr_Col)                  '
```

Macro B

```
dim i, j as number
```

```
   i = Get_Gbl_Num("MacALine")                                    '
   Get saved line number
   j = Get_Gbl_Num("MacACol")                                     '
   Get saved column
   if i = 0 then halt(FAIL, "No location saved by Macro
   A")
   Set_Csr(Get_LPtr(i), j, 0)                                     '
   Put cursor back where it was
```

Create a new Line Command

To create a new line command, do the following:

- Choose your command name. It must be short and cannot conflict with any existing line or primary commands.
- Line-command macro names by nature must be short; otherwise they cannot be entered in the sequence area of the edit screen.
- Line-command macros cannot contain digits in the name.
- It is possible to have a line-command macro with a single-letter name. Bear in mind that most of the single letter commands are already taken by SPFLite. The letters remaining available for use as single-letter macro names are **E K P Q U V Y** and **Z**. The macro files such macros are stored in would use the normal naming conventions. So, an **E** line-command macro would be stored in the file **E.MACRO**.
- For our example, we'll use **CT** (for Center Text). SPFLite will also automatically recognize **CTT** as the block form of this command, and will process any modifiers that imply a block, such as **n** or a **/** or **** modifier.
- You can also force a line macro of any format to be a Block or single line macro; see "Macro Format and Structure" for details.
- Create the macro name as the "singular form" of **CT.MACRO** (not **CTT.MACRO**)
- There is **no other** setup required to activate this as a line command. The presence of **CT.MACRO** in the \MACROS folder is sufficient.

Example:

```
' CT.MACRO
dim tt, tt2 as string
dim lno1, lno2, width, i as number
   ' Ensure we're called correctly
   if Is_Line_Cmd = FALSE then Halt(fail, "CT/CTT macro
was not issued as a line command")

   '----- Get the line number range
   lno1 = Get_Src1_Lptr                                           '
From line
   lno2 = Get_Src2_Lptr                                           '
To line
   width = Get_Line_Op                                            '
Get the centering width
   if width = 0 then width = Get_RBound                           '

   for i = lno1 to lno2                                           '
```

```
Loop through the line range
      if Is_Data(i) then                                  '
Just Data lines
         tt = trim$(Get_Line$(i))                         '
Get the trimmed text
         if width = 0 then width = Get_Line_Len(i)        '
Use line length if no other.
         if len(tt) > width then                          '
Center possible?
            Halt(warn, "One or more lines exceed Center
length")
         else                                             '
            tt2 =  repeat$((width - len(tt)) / 2, " ") +
tt ' Center it
            Set_Line(i, tt2)                              '
Stuff it back
         end if                                           '
      end if                                              '
   next                                                   '
   halt                                                   '
Done
```

Note: The check of **Is_Line_Cmd** to ensure the macro is invoked as a line command. The line range to be processed is obtained from **Get_Src1_LPtr** and **Get_Src2_LPtr**. For a single line selection, these two values would be identical. A **FOR** / **NEXT** loop is used to process each line. An **Is_Data** test ensures we only process text data lines and not special lines.

Sample Macros

```
' CT.MACRO
' Center text on a line. Uses RBound (if active) or specified length
' Syntax:  CTnn or CTTnn / CTTnn as line commands     nn is the width to center on
'
' Author: George Deluca
'
dim tt, tt2 as string
dim lno1, lno2, width, i as number
   if is_primary_Cmd then halt(fail, "CT macro was not invoked as a line command")

   '----- Get the line numbers
   lno1 = Get_Src1_Lptr                                ' From line
   lno2 = Get_Src2_Lptr                                ' To line
   width = Get_Src_Op                                  ' Get the centering width
   if width = 0 then width = Get_RBound                '

   for i = lno1 to lno2                                ' Loop through the line range
      if Is_Data(i) then                               ' Just Data lines
         tt = trim$(Get_Line$(i))                      ' Get the trimmed text
         if width = 0 then width = Get_Line_Len(i)     ' Use line length if no other
         if len(tt) > width then                       ' Center possible?
            Set_Msg(8, "One or more lines exceed Center length")
         else                                          '
            tt2 =  repeat$((width - len(tt)) / 2, " ") + tt ' Center it
            Set_Line(i, tt2)                           ' Stuff it back
         end if                                        '
      end if                                           '
   next                                                '
   halt                                                ' Done

-------------------------------------------------------------------------------
-

' ISRBox.MACRO
' ISRBOX - Draw box with its U/L corner at the cursor
' Syntax:  ISRBOX                                    With cursor sitting on the
'                                                    desired location. Normally the
'                                                    command would be assigned to a
'                                                    command key for convenience.
' Author: George Deluca
' Original in Rexx in the IBM macros documentation
'
dim i, j as number
if Get_Csr_LPtr = 0 then halt(FAIL, "Cursor is not within the text area")'

j = Get_Csr_LPtr                                     ' Point at 1st line
for i = 1 to 5                                       ' Set # lines to do
   if j = Get_Last_LPtr then exit for                ' Just in case we hit bottom
   if Is_Data(j) then                                ' Only data lines
      if i = 1 or i = 5 then                         ' If Top or bottom of box
         SPF_Ovr(j, Get_Csr_Col, "+--------------------+")  ' Overlay the box chars
      else                                           ' Else middle lines
         SPF_Ovr(j, Get_Csr_Col, "|                    |")  ' Overlay the box chars
      end if                                         '
      if i = 3 then Set_Csr(j, Get_Csr_Col + 2, 0)   ' Put cursor on middle line
```

```
   else                                                 '
      Decr i                                            ' Don't move box row number
   end if                                               '
   Incr j                                               ' On to next edit line
next                                                    '
halt                                                    ' Done

-------------------------------------------------------------------------------------
-

' ISRCount.MACRO
'
'  ISRCOUNT counts the number of occurrences of a string, and
'           issues a message.
'
' Syntax: ISRCOUNT string
'
' Author: George Deluca
' Original by IBM in the Macros documentation

dim msg as string
if Get_Arg$(0) = "" then halt(fail, "Missing search argument") ' Better have an operand

SPF_Cmd("FIND ALL " + SPF_Quote$(Get_Arg$(0)))        ' Issue a FIND ALL command
if Get_RC <> 0 then  halt(fail, "No occurrences of: " + Get_Arg$(1)) ' Tell of error

msg = Get_Msg$                                        ' Get the SPFLite message text
' Issue our own format message
halt("ISRCount found", Get_Arg$(0), parse$(msg, " ", 4), "times")

-------------------------------------------------------------------------------------
-

' ISRMask.MACRO
'
'  ISRMASK - Overlay a line with data from the mask line.
'  Use either line command O/OO or OR/ORR to specify
'  which lines to overlay. O/OO causes nondestructive
'  overlay, and OR/ORR causes a destructive overlay.
'
' Author: George Deluca
' Original by IBM in the Macros Documentation
'
dim mask as string
dim i as number
spf_debug(Get_Dest_LCmd$)
if left$(Get_Dest_LCmd$, 1) <> "O" then halt(fail, "No overlay range has been selected")

mask = Get_Profile$("MASK")                           ' Fetch the MASK data

for i = Get_Dest1_LPtr to Get_Dest2_LPtr              ' Loop through line range
   if Is_Data(i) then                                 ' Only Data lines
      if Get_Dest_LCmd$ = "O" or Get_Dest_LCmd$ = "OO" then  ' Normal O/OO type overlay?
         SPF_Ovr(i, 1, mask)                          ' Overlay the mask
      else                                            ' Must be OR/ORR type
         SPF_Ovr_Rep(i, 1, mask)                      ' Force Overlay the mask
      end if                                          '
   end if                                             ' End IsData
next                                                  '
halt                                                  ' Done

-------------------------------------------------------------------------------------
-
```

```
' PB.MACRO
' Run a batch compile of the current program. If the compiler reports an error,
' read the LOG file to locate the error, and move the cursor to the line in error
' and display the compiler's error message. This demonstrates how to read and
' process an external file using the thinBasic FILE module.
'
' Syntax: PB
'
' Author: George Deluca
'
dim cmd, errmsg, tt as string
dim errlin, errcol, i as number
dim fHandle as DWORD
uses "FILE"                                          ' Attach the FILE module

   '----- Save unmodified files
   SPF_Cmd("SAVEALL COND")

   '----- Build command line to run the compiler
   cmd = $DQ + "D:\Google Drive\Misc Data\PBWin10.bat"  + $DQ + " "
   cmd += $DQ + mid$(Get_FilePath$, 3) + $DQ + " "
   cmd += $DQ + Get_FileName$ + $DQ

   '----- Do the compile and get the RC result
   SPF_EXEC(cmd)
   if Get_RC = 0 then halt("PB10 compile successful")

   '----- Handle the compiler error, get the LOG file
   fHandle = FILE_OPEN(Get_FilePath$ + "\" + Get_FileBase$ +  ".LOG" , "INPUT")
   if fHandle = 0 then halt(fail, "PB10 compile failed, can't open LOG file")
   for i = 1 to 6                                    ' Read line 6 of the LOG
      errmsg = FILE_LineInput(fHandle)               '
   next i                                            '
   i = FILE_Close(fHandle)                           '

   '----- Extract error line number, column and error message text
   '----- Sample error line:  Error 442 in C:\Documents\Source\try.bas(12:016):  THEN
expected

   tt = parse$(errmsg, any "()", 2)                  ' Get the (nnn:nnn) value
   errlin = val(tt)                                  ' Line number preceeds the :
   tt = mid$(tt, instr(tt, ":") + 1)                 '
   errcol = val(tt)                                  ' Col number follows the :
   errmsg = parse$(errmsg, "):", 2)                  ' Message follows the ):

   '----- Issue an error message and set the cursor to the error line
   Set_Msg(fail,"Line: " + format$(errlin) + " Col: " + format$(errcol) + " " + errmsg)
   tt = Get_Line$(errlin + 1)                        ' Get the error line
   if errcol > len(tt) then                          ' Long enough to hi-lite?
      tt = lset$(tt, errcol)                         ' Lengthen it
      Set_Line(errlin + 1, tt)                       ' Stuff it back
   end if                                            '
   Set_Csr(errlin + 1, errcol, 1)                    ' Hi-light the compiler's error
location
   halt                                              ' Done

-------------------------------------------------------------------------------------------
-

' RM.MACRO
' Find rightmost occurrence of a string
' Syntax: RM string
'
' Author: George Deluca
```

```
'
dim hicol, hiline as number value 0

if Get_Arg$(0) = "" then halt(fail, "Missing search argument")

SPF_Cmd("FIND FIRST " & Get_Arg$(0))                  ' Issue 1st FIND cmd
do while Get_RC = 0                                   ' while found
   if Get_Find_Col > hicol then                       ' Save hi-water mark
      hicol = Get_Find_Col                            '
      hiline = Get_Find_LPtr                          '
   end if                                             '
   SPF_Cmd("RFIND")                                   ' Look some more
loop                                                  '

if hiline = 0 then                                    ' Find anything?
   halt(fail, "String not found")                     ' No, tell of error
else                                                  ' Yes
   Set_Csr(hiline, hicol, Get_Find_Len)               ' Set cursor to it
   halt("Found", Get_Arg$(0), "rightmost in col:", format$(hicol)) ' Issue good message
end if                                                '

-------------------------------------------------------------------------------------
-
```


Converting Rexx to thinBasic

Contents of Article

Introduction

If you come from an IBM ISPF background, you may have used or written TSO/Rexx scripts to interface with ISPF. While our **thinBasic** scripts and the capabilities of SPFLite are not exactly the same, there are enough similarities that you may be able to convert **some** existing Rexx scripts to **thinBasic** and have them perform useful work in the SPFLite environment.

You will not be able to convert ISPF-specific features, like access to the Dialog Manager, Table Services, File Tailoring, etc. but for scripts that simply interface with an edit session, there is enough functional commonality that a workable conversion may be possible.

We can't give you every possible conversion strategy and technique, or cover every syntax requirement, but the following may be a good starting point to help you. Because this topic is large and complicated, what is presented here should be considered as the first draft of a work in progress. We invite all users with ISPF/Rexx experience to test out these ideas, and

we welcome any and all suggestions for improving on these conversion strategies.

Keep in mind, in addition to ISPF vs. SPFLite differences, that the **thinBasic** functions and features are **comparable** but may not be **identical** to Rexx. Be sure to consult the **thinBasic** documentation for any features you are not familiar with, and to ensure that function parameters and results are what you expected.

Miscellaneous Considerations

- A major difference between Rexx and **thinBasic** is that Rexx treats all values as strings, which you don't need to declare, whereas **thinBasic** is a more "strongly typed" language, and does require you to predeclare string and numeric variables before you use them.

- If you have Rexx code that is checking return codes from ISREDIT and ISPEXEC calls, be aware the SPFLite return codes are different. Both treat a return code of 0 as "normal" and non-zero is usually treated as a "problem occurred". Beyond that, there is no similarity.

- Rexx scripts return messages back to ISPF using a call to ISPEXEC SETMSG MSG(msgID). In many cases, this call will reference two Dialog variables, **ZerrSm** and **ZerrLm**, the "short error message" and "long error message". ISPF will first display a "short" message, and if you press F1, it will first display the longer message, then launch a Help panel. SPFLite does not implement this two-stage messaging; there is only a single message displayed by SET_MSG. If you are converting ISPF Rexx scripts with ISPEXEC SETMSG calls, you will have to pick the message you like the best (either short or long) as the one and only message displayed. Because the SPFLite screen can handle fairly long messages, you may want to choose the **ZerrLm** value for SET_MSG. If you really need both the short and long messages, you may wish to call MSGBOX instead of SET_MSG. (You might use the "long" message as the MSGBOX message, and the "short" message as the MSGBOX title.) Some message ID codes for ISPEXEC SETMSG refer to fixed messages in the ISPF message library, and do not use the contents of Dialog variables **ZerrSm** and **ZerrLm**. You would need to know what those messages were, and/or have access to a copy of that library, to be able to convert such messages to SPFLite.

- ISREDIT commands that deal with lines use visible line number values. SPFLite uses the Line Pointer concept. In most cases, you can treat Line Pointers as if they were line numbers, for purposes of converting the Rexx script. You will need to add code to ensure that IS_DATA(linePointer) is TRUE, so you don't attempt to process non-data lines in your edit file.

- Any Rexx statements involving LINE functions such as LINEIN and LINEOUT that you wish to convert to **thinBasic** will require a **USES "FILE"** statement in your macro. Be aware that if you attempt to perform file I/O on files that SPFLite already has opened, it could result in runtime conflicts. If you attempt to modify SPFLite control files, such as Profile INI files or the SPFLite configuration file, it could cause SPFLite to malfunction. We won't say you cannot attempt such things, but you need to be careful if you do. We cannot guarantee that such techniques will work. If you try this, be sure to back up the SPFLite control files first so you have a recovery strategy.

- Rexx has a command called INTERPRET, which will take a string and treat it as a Rexx statement, which is immediately parsed and executed. **thinBasic** does not

have an INTERPRET capability per se, but it does have a series of functions that begin with "Eval" which are used to evaluate string and math expressions. If you are converting a Rexx script with INTERPRET, one of the Eval functions may work for you, if the Rexx INTERPRET operation is limited to dynamic expression evaluation. Consult the **thinBasic** documentation for more information on the Eval functions.

- Rexx scripts tend to frequently use the PARSE facility. Often this is simply to grab arguments passed to a subroutine or to read values from a file or queue. However, you may run into cases where PARSE is being used in more complex ways. **thinBasic** has a PARSE function, which operates differently than Rexx, but it may be enough for simpler cases. Otherwise, you may have a bit of work to do converting this.

- Rexx has a novel way of storing string values, in which you can define a fixed "base" variable name and append a dynamic "stem" to it. Rexx uses this stem approach to substitute for arrays, which is doesn't support. If you have a Rexx script that uses stemmed variables, you may have considerable work to convert them. Often, Rexx stem variables are simply used as arrays, and **thinBasic** supports these natively. If your Rexx script does anything fancier than that with stems, you might be able to use SPFLite Globally Stored Data as a substitute for stem variables.

- ISPF has a way to store variable values in the "shared pool" or the "profile pool" using ISPEXEC VGET and ISPEXEC VPUT. If you have a script that does this, you will have to rewrite it. Here are some strategies for going about this:

    - You can get edit Profile values using Get_Profile$, and can set Profile values using a SPF_CMD call, like SET_CMD("EOL CRLF")
    - Other than standard Profile values, you cannot add user-defined values to an SPFLite edit profile
    - For "shared variables", Globally Stored Data may be sufficient, as long as you understand it is not persistent across SPFLite instances. Once you close down SPFLite and restart it, the values will be gone.
    - If you want to store persistent values, you can use Set_SETVAR and Get_SETVAR$ to access the SPFLite SET variable pool. If you do this, be careful not to inadvertently write over any existing values. You can use SET names that have dots in them; this allows you to form "qualified SET names". Adopt a consistent naming convention for the SET symbols you define this way.
    - Because SPFLite SET names are persistent, you need to evaluate whether the script you are converting intended its variables to be persistent or not. If not, using SET names may not be not the right approach, and you may wish to use Globally Stored Data instead.
    - For example, if you are converting a script that stored name ABC in the shared pool, and XYZ in the profile pool, you could use those pool names as SET name qualifiers. Keep in mind that if you call Get_SETVAR for an undefined name, the function will return an empty (null) string. Example:

      Rexx

      ```
      "ISPEXEC VGET ABC SHARED"
      "ISPEXEC VGET XYZ PROFILE"

      "ISPEXEC VPUT ABC SHARED"
      "ISPEXEC VPUT XYZ PROFILE"
      ```

SPFLite

```
DIM ABC,XYZ AS STRING

ABC = Get_SETVAR ("SHARED.ABC")
XYZ = Get_SETVAR ("PROFILE.XYZ")

Set_SETVAR ("SHARED.ABC", ABC)
Set_SETVAR ("PROFILE.XYZ", XYZ)
```

General Statements

Rexx	***thinBasic***
ADDRESS	*not needed*
ARG, PARSE ARG	**Get_Arg$** **UCASE$(Get_Arg())**
CALL routine	**routine** *[CALL keyword not used]*
DO	*see Program Control Structures below*
DROP	**REDIM** *may apply, but probably not needed*
EXIT	**HALT**
EXIT returnCode	**HALT (returnCode, "message")**
IF	*see Program Control Structures below*
INTERPRET	*thinBasic* **Eval** *functions may apply. See thinBasic help for more information.*
ITERATE	**ITERATE FOR|WHILE|DO** *thinBasic* **ITERATE** *cannot reference an outer nested block, but only the closest one*
LEAVE	**EXIT FOR|WHILE|DO** *thinBasic* **EXIT** *has a "number of times" operand to exit from multiple FOR/NEXT blocks*
LINEIN	**USES "FILE"** *various I/O statements; see thinBasic Help*

LINEOUT	**USES "FILE"** *various I/O statements; see thinBasic Help*
NOP (used for "do nothing" branches of IF statements)	**NOP** -- thinBasic does not natively have a NOP statement, but SPFLite has added one. It is essentially a SUB named NOP that does nothing. --
NUMERIC	*not supported*
PARSE	*thinBasic has a number of parsing functions, including* **PARSE, PARSE$, PARSECOUNT** *and* **PARSESET$,** *as well as several other string functions. See the thinBasic documentation for more information.*
PROCEDURE	**SUB** *and* **FUNCTION**
PULL	*not supported*
PUSH	*not supported*
QUEUE	*not supported*
RAISE	*not supported. In Rexx,* **RAISE** *is sometimes used to simulate a* **GOTO** *statement, which thinBasic does not support either. It is possible some use of* **EXIT** *may be applied.*
RETURN ***[****expression****]***	**RETURN** ***[****expression****]***
SAY **SAY** ***string***	**SPF_DEBUG** **SET_MSG** **MSGBOX** **USES "CONSOLE"** **PRINTL** ***string*** **' displays on a console window** **' PRINT instead of PRINTL will output the** **' *string* without a line terminator.** **' You can use CLS to clear the screen** **' See CONSOLE module documentation for more** **' details.**

SIGNAL	*not supported. In Rexx,* **SIGNAL** *is sometimes used to simulate a* **GOTO** *statement, which thinBasic does not support either. It is possible some use of* **EXIT** *may be applied.*
TRACE	**SPF_TRACE** *functionality is different*

Program Control Structures

The main difference here is that Rexx uses a common DO ... END structure for all blocks of code (like PL/1 which it is modeled after) while **thinBasic** has unique delimiters for each type of block. You will also notice that in Rexx you will often see an END prior to an ELSE, ELSE IF or WHEN/OTHERWISE clauses, whereas in **thinBasic** for ELSE, ELSEIF and CASE, that is not necessary.

Note that in **thinBasic**:

- ELSEIF is all one word
- END IF is two words, where the IF part is required
- END SELECT is two words, where the SELECT part is required
- In FOR/NEXT loops, NEXT does not require an index name afterwards like NEXT N. It's just NEXT. Specifying the N value **is** supported and is probably a good idea for documenting purposes.
- SELECT CASE defines a common expression, which is tested by each CASE clause, whereas a Rexx SELECT defines a test at each WHEN clause, and there is no common expression on the Rexx SELECT clause. This means there is more to do than merely changing keywords; may will have to restructure your SELECT statements. If your WHEN clauses compare against a common value, you can move that value into the SELECT CASE expression. Otherwise, you can specify this as **SELECT CASE OF**, and ensure that each WHEN clause is rewritten to be a CASE clause with an expression that produces a "truth" value of -1 or 0. You can ensure that by enclosing each expression in the **thinBasic** function **IsTrue**.
- The **OF** keyword is a bit of "magic" supported by SPFLite; just code it as is when converting Rexx SELECT statements.
- If you are converting a Rexx WHEN clause that is already in the form of a comparison like COUNT = 5, the expression will automatically produce a **thinBasic** "truth value" of -1 or 0, and so enclosing it in a **thinBasic** function **IsTrue** would not be required.

Rexx	***thinBasic***
statement1 ; statement2	***statement1 : statement2***
if ***cond*** **then** ***statement***	**if** ***cond*** **then statement**

if ***cond1*** **then do**
**   *statement1***
end
else if ***cond2*** **then**
**   *statement2***
end
	if ***cond1*** **then**
	** *statement1***
	elseif ***cond2*** **then**
	** *statement2***
	else
	** *statement3***

else do *statement3* end	end if

do i = 1 to 10
  if *cond1* then leave
    statement
    if *cond2* then iterate
end
 	
do i = 1 to 10 by -1
  if *cond1* then leave
    statement
    if *cond2* then iterate
end
	for i = 1 to 10
	    if *cond1* then exit for
	    statement
	    if *cond2* then iterate
	for
	next
	
	for i = 1 to 10 step -1
	    if *cond1* then exit for
	    statement
	    if *cond2* then iterate
	for
	next

do while *cond*
    if *cond1* then leave
    statement
    if *cond2* then iterate
end
	do while *cond*
	    if *cond1* then exit do
	    statement
	    if *cond2* then iterate
	do
	loop

do forever
    statement
end
	do
	    statement
	loop

select
    when *cond1*
    do
        statement1
    end
    when *var = value*
    do
        statement2
    end
    when *var1 = value1, var2 = value2*
    do
        statement3
    end
/* ... */
    otherwise do
        statement3
    end
end
 	
' NOTE 1: In Rexx, a comma-separated expr
' list on WHEN implies an AND test
	select case of
	    case IsTrue(*cond1*)
	
	       statement1
	
	    case var = value
	
	       statement2
	
	    case (var1 = value1) and (var2 = value2)
	
	        statement3
	
	    case else
	        statement3
	
	end select
	
	' NOTE 1: the "of" after SELECT CASE is a
	' keyword supported only within SPFLite.
	' It won't work in stand-alone thinBasic.

	' NOTE 2: In thinBasic, a comma-separated expr ' list on CASE implies an OR test
, (comma as line continuation)	**_** (underscore as line continuation)
/* line comment */ /* block comment */	**/* line comment */** (or) **' line comment /* block comment */ ' NOTE: Using /* */ as block comments is ' non-standard for BASIC, but thinBasic ' allows it.**

Operators

Operators that are the same are not mentioned

Rexx	***thinBasic***
&	**AND**
|	**OR**
|| (concatenate)	**&** **+**
/= \= ¬= <> ><	**<>**
<< <<= >> >>= ==	**< <= > >= =**
¬ ****	**NOT**
one two (concatenation with blank) **(one)(two)** (abuttal)	**one & " " & two** **one&two**
/ (real divide)	**/**
% (integer divide)	****
****** (power)	**^**
&&	**XOR**

Macro Headers

Rexx	*thinBasic*
/* REXX other comments */ "ISREDIT MACRO (arg1,arg2) PROCESS"	' macname.MACRO DIM arg1,arg2 AS STRING arg1 = Get_Arg$(1) arg2 = Get_Arg$(2)

/* REXX other comments */
"ISREDIT MACRO (arg1,arg2)
NOPROCESS"
"ISREDIT PROCESS RANGE C"
	' macname.MACRO
	DIM arg1,arg2 AS STRING
	DIM _ZFRANGE,_ZLRANGE AS
	NUMBER = 0
	arg1 = Get_Arg$(1)
	arg2 = Get_Arg$(2)
	
	if is_line_cmd and _
	(get_src_lcmd$="C" or
	get_src_lcmd$="CC") then
	    _ZFRANGE =
	get_src1_lptr
	    _ZLRANGE =
	get_src2_lptr
	end if

ISREDIT calls

Rexx	*thinBasic*
"ISREDIT AUTOSAVE ON|OFF PROMPT|NOPROMPT"	SPF_CMD("AUTOSAVE ON|OFF PROMPT|NOPROMPT")
"ISREDIT (var) = BLKSIZE"	var = GET_PROFILE$("LRECL")
"ISREDIT BOUNDS left right"	SPF_CMD("BOUNDS left right")
"ISREDIT (left,right) = BOUNDS"	left = Get_Lbound right = Get_Rbound
"ISREDIT (var) = CAPS"	var = GET_PROFILE$("CAPS")
"ISREDIT CAPS = ON|OFF"	SPF_CMD("CAPS ON|OFF")
"ISREDIT CHANGE operands"	SPF_CMD("C operands")
"ISREDIT (times) = CHANGE_COUNTS" "ISREDIT (times,lines) = CHANGE_COUNTS"	DIM msg, times, lines AS STRING SPF_CMD("CHANGE operands") msg = GET_MSG$ times = PARSE$(msg, " ", 4) lines = PARSE$(msg, " ", 7) if lines = "" then lines = times
"ISREDIT COPY operands"	SPF_CMD("COPY operands")

"ISREDIT (line,col) = CURSOR"	line = Get_LNUM(Get_CSR_LPTR)) col = Get_CSR_COL
"ISREDIT CURSOR = line col"	SET_CSR(line,col,0)
"ISREDIT CUT operands"	SPF_CMD("CUT operands")
"ISREDIT (var) = DATA_CHANGED"	var = IIF$(GET_MODIFIED,"YES","NO")
"ISREDIT DELETE operands"	SPF_CMD("DELETE operands")
"ISREDIT EXCLUDE operands"	SPF_CMD("EXCLUDE operands") SPF_CMD("X operands")
"ISREDIT FIND operands"	SPF_CMD("FIND operands") SPF_CMD("F operands")
"ISREDIT (times) = FIND_COUNTS" "ISREDIT (times,lines) = FIND_COUNTS"	DIM msg, times, lines AS STRING SPF_CMD("FIND operands") msg = GET_MSG$ times = PARSE$(msg, " ", 4) lines = PARSE$(msg, " ", 7) if lines = "" then lines = times
"ISREDIT FLIP operands"	SPF_CMD("FLIP operands")
"ISREDIT HEX = ON|OFF"	SPF_CMD("HEX ON|OFF")
"ISREDIT var = HEX"	var = GET_PROFILE$("HEX")
"ISREDIT HIDE operand"	SPF_CMD("HIDE")
"ISREDIT RESET HIDE"	SPF_CMD("RESET HIDE")
"ISREDIT HILITE [ON|OFF] [AUTO] [FIND]"	SPF_CMD("HILITE [ON|OFF] [AUTO] [FIND]")
"ISREDIT INSERT label| linenum [lines]"	SPF_CMD("LINE I" & TSTR$(lines) & label) SPF_CMD("LINE N" & TSTR$(lines) & label)
"ISREDIT (var) = LABEL .label|linenum"	var = GET_Label$(GET_LPTR(label))
"ISREDIT LABEL .label| linenum = .newlabel	SPF_CMD("LINE '.newlabel' .label")
"ISREDIT (var) = LINE .label|linenum"	var = GET_LINE$(GET_LPTR(label_or_linenum))
"ISREDIT LINE .label| linenum = data"	SET_LINE(GET_LPTR(label_or_linenum),data)
"ISREDIT LINE_AFTER	SPF_CMD("LINE N

.label|linenum = data"	**label_or_linenum") SET_LINE(GET_LPTR(label_or_linenum)+1,data)**
"ISREDIT LINE_AFTER .label|linenum = NOTELINE data"	**SPF_CMD("LINE NOTE label_or_linenum") SET_LINE(GET_LPTR(label_or_linenum)+1,data)**

"ISREDIT LINE_BEFORE .label|linenum = data"	**DIM LPTR AS NUMBER
	LPTR = GET_LPTR(.label|linenum) - 1
	IF LPTR > 0 THEN
	SPF_CMD("LINE N !" & TSTR$(LPTR))
	SET_LINE(LPTR+1, data)
	END IF**

"ISREDIT LINE_BEFORE .label|linenum = NOTELINE data"	**DIM LPTR AS NUMBER
	LPTR = GET_LPTR(.label|linenum) - 1
	IF LPTR > 0 THEN
	SPF_CMD("LINE NOTE !" & TSTR$(LPTR))
	SET_LINE(LPTR+1, data)
	END IF**

"ISREDIT (var) = LINENUM .label"	**var = GET_LNUM(GET_LPTR(".label"))**
"ISREDIT LOCATE operands"	**SPF_CMD("LOCATE operands")**
"ISREDIT (var) = LRECL"	**var = GET_PROFILE$("LRECL")**
"ISREDIT (var) = MEMBER"	**var = GET_FILEBASE$**
"ISREDIT PASTE operands"	**SPF_CMD("PASTE operands")**
"ISREDIT PRESERVE ON|OFF|C"	**SPF_CMD("PRESERVE ON|OFF|C")**
"ISREDIT (var) = PRESERVE"	**var = GET_PROFILE$("PRESERVE")**
"ISREDIT PROCESS [DEST] [RANGE *cmd1* [*cmd2*]]"	The ISPF line command names **cmd1** and **cmd2** can be up to 6 characters and can contain any alphabetic or special character except blank, hyphen (-), or apostrophe ('). SPFLite line command names are not predeclared in this way, but use the GET_SRC_LCMD$ and GET_DEST_LCMD$ functions. You cannot define arbitrary line command names for use in a macro the way ISPF does this. If you use the macro as a primary command, possible source commands are C/CC and M/MM, while line-command macros use the name of

	the macro itself as the name of the source line range. Possible destination commands are A/AA, B/BB, H/HH, W/WW, O/RR and OR/ORR for both types of macros. SPFLite line command names are currently limited to four characters.
"ISREDIT (var) = RANGE_CMD"	**var = GET_SRC_LCMD$** -- or -- **var = GET_DEST_LCMD$**
"ISREDIT RCHANGE"	**SPF_CMD("RCHANGE")**
"ISREDIT (var) = RECFM"	**var = GET_PROFILE$("RECFM")**
"ISREDIT REPLACE operands"	**SPF_CMD("REPLACE operands")**
"ISREDIT RFIND"	**SPF_CMD("RFIND")**
"ISREDIT SEEK operands"	**SPF_CMD("FIND operands DX")**
"ISREDIT (times) = SEEK_COUNTS" **"ISREDIT (times,lines) = SEEK_COUNTS"**	**DIM msg, times, lines AS STRING** **SPF_CMD("FIND operands DX")** **msg = GET_MSG$** **times = PARSE$(msg, " ", 4)** **lines = PARSE$(msg, " ", 7)** **if lines = "" then lines = times**
"ISREDIT (var) = SESSION" -- may return EDIT or VIEW --	**var = GET_SESSION_TYPE$** -- may return EDIT, BROWSE or other values --
ZerrSm = "short message" **ZerrLm = "long message"** **"ISPEXEC "SETMSG MSG(msgCode)**	**Set_Msg ([rc,] ZerrLm)** -- see notes above about ISPEXEC SETMSG --
"ISREDIT SETUNDO operands"	**SPF_CMD("SETUNDO number")** -- SPFLite SETUNDO not compatible with ISPF --
"ISREDIT SHIFT (lnum_or_label [n]" **"ISREDIT SHIFT) lnum_or_label [n]"** **"ISREDIT SHIFT < lnum_or_label [n]"** **"ISREDIT SHIFT > lnum_or_label [n]"**	**SPF_CMD("LINE (n !" & GET_LPTR(lnum_or_label)))** **SPF_CMD("LINE)n !" & GET_LPTR(lnum_or_label)))** **SPF_CMD("LINE <n !" & GET_LPTR(lnum_or_label)))** **SPF_CMD("LINE >n !" & GET_LPTR(lnum_or_label)))**
"ISREDIT SORT operands"	**SPF_CMD("SORT operands")**

"ISREDIT SOURCE operand" "ISREDIT (var) = SOURCE" "ISREDIT RESET SOURCE"	SPF_CMD("SOURCE operand") var = GET_PROFILE$("SOURCE") SPF_CMD("RESET SOURCE")
"ISREDIT SUBMIT operands"	SPF_CMD("SUBMIT operands")
"ISREDIT TABS ON|OFF" "ISREDIT (var) = TABS"	SPF_CMD("TABS ON|OFF") var = GET_PROFILE$("TABS")
"ISREDIT (var) = TABSLINE"	DIM var AS STRING SPF_CMD("LINE TABS !2") var = GET_LINE$(3) SPF_CMD("LINE D !3")
"ISREDIT = TABSLINE = data"	SPF_CMD("LINE TABS !2") SET_LINE$(3,data) SPF_CMD("LINE D !3")
"ISREDIT TFLOW lnum_or_label [n]"	SPF_CMD("LINE TF" & n & "lnum_or_label")
"ISREDIT TSPLIT lnum_or_label [n]"	SPF_CMD("LINE TS" & n & "lnum_or_label")

Rexx Functions

Rexx	*thinBasic*
abbrev(longString,shortString)	StartsWith(longString,shortString,0)
abs(value)	abs(value)
address()	n/a
arg()	get_arg_count
arg(n)	get-arg$(n)
center(string,len) center(string,len,pad)	cset$(string,len) cset$(string,len,pad)
changestr(needle,haystack,newneedle)	replace$(haystack,needle,newneedle)
copies(string,n)	repeat$(n,string)
countstr(needle,haystack)	tally(haystack,needle)
date()	date$() (format operands are different)
delstr(string,pos,len)	strdelete$(string,pos,len)
str = directory() directory(newdirectory)	USES "FILE" str = DIR_GetCurrent USES "FILE" DIR_Change(newdirectory)
	USES "FILE" ' -- for all calls

filespec("Drive",fullpath) filespec("Path",fullpath) filespec("Location",fullpath) filespec("Name",fullpath) filespec("Extension",fullpath)	LEFT$(FILE_PATHSPLIT(fullpath,%Path_root),2) MID$(FILE_PATHSPLIT(fullpath,%Path_Rootpath),3) FILE_PATHSPLIT(fullpath,%Path_Rootpath) FILE_PATHSPLIT(fullpath,%Path_FileExt) FILE_PATHSPLIT(fullpath,%Path_Ext)
format()	format$() (format operands are different)
insert(new,target,pos)	strinsert$(target,new,pos)
left(string,len) left(string,len,pad)	left$(string,len) lset$(left$(string,len),len,pad)
lastpos(needle,haystack) lastpos(needle,haystack,start)	instr(-1,needle,haystack) instr(start-len(haystack)-1,needle,haystack)
length(string)	len(string)
lower(string)	lcase$(string)
max(str1,str2,etc) max(num1,num2,etc)	max$(str1,str2,etc) max(num1,num2,etc)
min(str1,str2,etc) min(num1,num2,etc)	min$(str1,str2,etc) min(num1,num2,etc)
pos(needle,haystack) pos(needle,haystack,start)	instr(,needle,haystack) instr(start,needle,haystack)
reverse(string)	strreverse$(string)
right(string,len)	right$(string,len)
sign(number)	sgn(number)
space(string) space(string,n) space(string,0)	trimfull$(string) replace$(trimfull$(string)," ",lset$("",3)) remove$(string, " ")
strip(string) strip(string,"Both") strip(string,"Leading") strip(string,"Trailing") strip(string,,char) strip(string,"Both",char) strip(string,"Leading",char) strip(string,"Trailing",char)	trim$(string) trim$(string) ltrim$(string) rtrim$(string) trim$(string,char) trim$(string,char) ltrim$(string,char) rtrim$(string,char)

substr(data,pos,len) **substr(data,pos)**	**mid$(data,pos,len)** **mid$(data,pos)**
time()	**time$** (format is different)
translate(string)	**ucase$(string)**
translate(string,newCharSet,oldCharSet)	**replace$(string, ANY oldCharSet, newCharSet)**
trunc(string) **' trunc is often used to extract a "qualified** **' option" like "3" from a string like "3.4"**	**extract$(string,".")**
upper(string)	**ucase$(string)**
verify(string,charset)	**verify(,string,charset)**

thinBasic Essentials

Overview

The macro language used by SPFLite is **thinBasic** (www.thinBasic.com) which is an interpreted variation of PowerBasic (www.PowerBasic.com). Both **thinBasic** and SPFLite itself are written in PowerBasic, which simplifies the linkage conventions between them. The interface between SPFLite and **thinBasic** is quite straightforward, which makes possible a very efficient implementation.

> thinBasic is Copyright 2004 - 2013 by thinBasic
> PowerBasic is Copyright by PowerBasic, Inc. 2061 Englewood Road Englewood, FL 34223

The SPFLite install will install the components of **thinBasic** needed to operate as the macro scripting language. It does **not** do a full install of **thinBasic**. If you plan to, or wish to explore **thinBasic** as a normal scripting language, you should obtain and install the full product from the **thinBasic** web site.

The Basic Language

The material following this point is meant as a quick summary of the Basic language used. The definitive answer as to syntax, usage etc. is the **thinBasic** Help document itself. You can reach the **thinBasic** Help file from within SPFLite by entering the command `HELP THINBASIC`

The **thinBasic** Help document is quite large and includes information on all aspects of **thinBasic**, including all the add-on modules, Software Developer Kit, etc. We have tried to provide the essentials for most coding tasks here to simplify things. Just don't consider it as the final word on what **thinBasic** has to offer.

We are mindful that **thinBasic** is a large language, one that offers everything you need to write macros. You may (rightly) conclude that it's "more than you need" or even "too much", but it's not likely you will say that it's "less than you need". The overview we provide here is presented with the aim of not overwhelming you, but giving you (mostly) only what you need to understand the principles and concepts to get some useful work done.

Because **thinBasic** is so large, we don't claim to have total knowledge or understanding of every last one of its inner workings. We may end up learning as much about **thinBasic** from you as you learn from us. This will be a learning experience for everyone, but it's going to be interesting !

Data Types

Although **thinBasic** supports most normal data types, unless you have some very particular requirement we suggest you use only the following two.

STRING
For the storage of normal text data. Strings are dynamically sized, and can contain strings (in theory) up to 2 GB characters long. In practice, string lengths are limited by available memory.

NUMBER
For the storage of numerical data. Number variables are Extended Precision floating point data of 80 bits in the range of 3.4 x 10^-4932 to 1.2 x 10^4932. This format is more than capable of handling any of your numeric variable requirements. And as **thinBasic** uses this format internally for numeric calculations, it avoids repetitive internal data conversions if numeric variable are stored in this format.

Defining Variables

Every variable used in your macro must be declared before usage.

Variables are declared using the DIM keyword. The DIM keyword works just like standard Basic DIM except that a VALUE clause followed by a numeric or string expression can be entered. If the VALUE clause is used, and multiple variable names are specified, every variable will be initialized to the same value in the VALUE expression.

The VALUE expression can be any Basic expression, including function calls and the constants TRUE and FALSE>

Syntax

```
DIM VarName [ , Varname [ ,...]] As NUMBER [ VALUE
numeric-expression ]
DIM VarName [ , Varname [ ,...]] As STRING [ VALUE
string-expression ]
DIM AUTO
```

Examples

Code	Comment
`DIM MyVar AS NUMBER`	Define MyVar as a Number
`DIM Var1, Var2 AS STRING VALUE "AAA"`	Define Var1 and Var2 as STRINGs whose initial value is “AAA”
`DIM Var2 AS NUMBER VALUE 100`	This will declare a variable (Var2) whose initial value will be 100.

Operators

The macro language supports the following arithmetic and relational operators:

Arithmetic

+	Addition (for numeric expressions, concatenation for strings)
-	Subtraction
*****	Multiplication
/	Division
****	Integer Division
^	Exponentiation
&	String concatenation (also see + above)

Relational

AND	And relationship
OR	Or relationship
NOT	Negation
<	Less than
<=	Less than or Equal
>	Greater than
>=	Greater than or Equal
<>	Not equal
=	Equal

Compound Numeric Operators

+= Add variable value to right expression before assigning result back to variable

-= Subtract variable value to right expression before assigning result back to variabl

***=** Multiply variable value to right expression before assigning result back to variable

/= Divide (floating point version) variable by right expression before assigning result back to variable

\= Divide (integer version) variable by right expression before assigning result back to variable

Compound String Operators

+=	Add variable value to right expression before assigning result back to variable
&=	Add variable value to right expression before assigning result back to variable
.=	Add variable value to right expression before assigning result back to variable

Implicit Assignment Operators

Implicit assignment takes place when a statement starts with a variable name followed by an operator.

For example:

```
B + 2
```

is considered as an implicit assignment to variable B and is equivalent the one of the below statement statements:

```
B = B + 2
```

```
B += 2
```

Implicit assignment is actually supported only for scalar numeric variables.

Supported operators are: **+ - * / **

Labels

In some BASIC variants, you are permitted to define labels using a user-defined word, followed by a colon. These labels are referenced by GOTO and GOSUB statements.

thinBasic does not support GOTO or GOSUB statements, and you cannot define statement labels.

Language Keywords

Flow Control

DO / LOOP / UNTIL	DO [{{WHILE | UNTIL} *Expression* }]
	  … your code …
	  … [EXIT DO]
	  … [ITERATE DO]
	  … your code …
	
	LOOP [{{WHILE | UNTIL} *Expression* }]

Expression is a numeric expression, in which non-zero values represent logical TRUE, and zero values represent logical FALSE.

If *Expression* contains string values, these must be part of a larger expression that results in a numeric value that can be treated as TRUE or FALSE. For example, the expression A = "X" produces a TRUE or FALSE value, but a simple string like "X" cannot be used as an expression in condition tests like WHILE or UNTIL. (That is, numbers can be converted to a "*truth value*", but strings cannot be.)

DO/LOOP statements are quite flexible. They allow you to create loops with the test for the terminating condition at the top of the loop, the bottom of the loop, both places, or none of the above.

The **WHILE** and **UNTIL** keywords are used to add tests to a **DO/LOOP**. Use the **WHILE** if the loop should be repeated if *expression* is **TRUE**, and terminated if *expression* is **FALSE.** **UNTIL** has the opposite effect; that is, the loop will be terminated if *expression* is **TRUE**, and repeated if **FALSE**.

When **WHILE** or **UNTIL** appears after the **DO** keyword, the test is performed **before** executing the body of the do-loop block. When **WHILE** or **UNTIL** appears after the **LOOP** keyword, the test is performed **after** executing the body of the do-loop block. It is possible to put a "before test" and an "after test" on the same do-loop block, though usually only one or the other is needed.

Note that **DO** is ended by **LOOP** and not by **END DO**.

An **EXIT DO** statement can be used to force an exit from a loop regardless of the **WHILE** / **UNTIL** condition. (**EXIT DO** is like a **break** or **leave** statement in some languages.)

An **ITERATE DO** statement transfers control to the bottom of the loop and triggers condition evaluation. (**ITERATE DO** is like a **continue** statement in some languages.) Because **ITERATE DO** creates the equivalent of a "backward-reference GOTO" statement, great care is needed to ensure you do not create a logic error that would cause the program to be "stuck in a loop".

FOR / NEXT	FOR *Counter* = *Start* TO *Stop* [STEP *Increment*]
	  … your code …
	  … [EXIT FOR]
	  … [ITERATE FOR]
	  … your code …
	NEXT [Counter]

When a **FOR** statement is encountered, *start* is assigned to *Counter,* and *Counter* is tested to see if it is greater than (or, for negative *increment*, less than) *stop*. If the initial *Start* value makes the "stopping condition" immediately true, the body of the FOR/NEXT loop is not executed at all.

If the initial *Start* value does not make the "stopping condition" immediately true, the statements within the **FOR/NEXT** loop are executed, *increment* is added to *Counter*, and *Counter* is again tested against *stop*. The statements in the loop are executed repeatedly until the stopping condition is true, at which time control passes to the statement immediately following the **NEXT**.

Counter is a numeric variable serving as the loop counter.

Start is a numeric expression specifying the value initially assigned to *Counter*.

Stop is a numeric expression giving the value that *Counter* must reach for the loop to be terminated.

Increment is an optional numeric expression defining the amount by which *Counter* is incremented with each loop execution. If not specified, *Increment* defaults to 1.

Note that **FOR** is ended by **NEXT** and not by **END FOR**.

An **EXIT FOR** statement can be used to force an exit from a loop regardless of stopping condition. (**EXIT FOR** is like a **break** or **leave** statement in some languages.)

An **ITERATE FOR** statement transfers control to the bottom of the loop and triggers evaluation of the stopping condition. (**ITERATE FOR** is like a **continue** statement in some languages.) Because **ITERATE FOR** creates the equivalent of a "backward-reference GOTO" statement, great care is needed to ensure you do not create a logic error that would cause the program to be "stuck in a loop".

It is possible to modify the *Counter* variable inside the FOR/NEXT loop using assignment statements. However, experience has shown that such techniques often result in programming logic errors, and is thus not recommended. Instead of modifying the *Counter* you could do one of the following:

- Code an inner IF statement to selectively execute statements in a given loop iteration
- Use the EXIT FOR statement to leave the loop altogether
- Use the ITERATE FOR statement to begin the next iteration of the for-loop. Because of the potential risk of being "stuck in a loop" by using this statement, it is often safer to enclose a group of statements within your loop by an inner IF block, rather than trying to "skip" them using an ITERATE FOR statement.

NOTE The NEXT statement **does not** require the name of the *counter* variable. *Counter* is optional and is treated as comments.

IF / THEN / ELSE / ELSEIF / END IF	IF *Expression* THEN
	  … your code …
	[ELSEIF *Expression* THEN
	  … your code …
	]
	[ELSE
	  … your code …
	]
	END IF

NOTE:	Macro script does **not** support the single line IF / ELSE statement. The ELSE clause will be ignored. i.e. IF A = B then C = D else C = E will not function as you expect.

In executing **IF** blocks, the truth of the *expression* in the initial **IF** statement is checked first. If it evaluates to **FALSE** (zero), each of the following **ELSEIF** statements is examined in order. There can be as many **ELSEIF** statements as desired. As soon as one is found to be **TRUE** (non-zero), the statement(s) following the associated **THEN** and before the next **ELSEIF** or **ELSE** will be executed.

Execution then jumps to the statement just after the terminating **END IF** without making any further tests. If none of the test expressions evaluates to **TRUE**, the statement(s) in the **ELSE** clause (which is optional) are executed.

IF blocks must be terminated with a matching **END IF** statement. Note that the **END IF** statement requires a space and the **ELSEIF** statement does not.

SELECT / CASE / END SELECT	SELECT CASE expression
	  CASE testlist
	  {statements}
	  [CASE testlist
	  {statements}]
	  [CASE ELSE
	  {statements}]
	END SELECT

testlist is one or more tests, separated by commas, to be performed on expression. expression can be of type **STRING** or **NUMBER**.

When a **SELECT** statement is encountered, expression is evaluated using the testlist in the first **CASE** clause. If the evaluation is **FALSE**, the evaluation is repeated using the next testlist. As soon as an evaluation is **TRUE** (non-zero), the statements following that **CASE** clause are executed, up to the next CASE clause.

Execution then passes to the statement following the **END SELECT** statement. If none of the evaluations is **TRUE**, the statements following the optional **CASE ELSE** clause are executed.

The tests that may be performed by a **CASE** clause include: equality, inequality, greater than, less than, and range ("from-to") testing. The **SELECT CASE** block can do string or numeric tests, but these cannot be interchanged.

Examples of numeric CASE clause tests:

```
SELECT CASE numeric_expression
   CASE > b                          ' relational; is expression
   > b?
   CASE 14                           ' equality (= assumed); is
   equal to 14?
   CASE b TO 99                      ' range; is it between
   variable b and 99
   CASE 14, b                        ' two equality tests; equal
   to 14 or b
```

Examples of string CASE clause tests:

```
SELECT CASE string_expression
   CASE > b$                          ' relational; is
expression > b$?
   CASE "X"                           ' equality (= is assumed);
is equal to "X"?
   CASE "A" TO "C"                    ' range; is between "A"
and "C"
   CASE "Y", b$                       ' two equality tests; is
equal to "Y" or b$?
```

WHILE / WEND	WHILE numeric_expression
	  … your code …
	  … [EXIT WHILE]
	  … [ITERATE WHILE]
	  … your code …
	WEND

If numeric_expression is **TRUE** (it evaluates to a non-zero value), all of the statements between the **WHILE** and the terminating **WEND** are executed. Control then jumps back to the **WHILE** statement and repeats the test. If it is still **TRUE**, the enclosed statements are executed again. This process is repeated until the test expression evaluates to zero, or an **EXIT WHILE** statement is encountered. In either case, execution passes to the statement following **WEND**.

If numeric_expression evaluates to **FALSE** (zero) on the first pass, none of the statements in the loop are executed.

Loops built with **WHILE/WEND** statements can be nested (enclosed within each other). Each **WEND** matches the most recent unmatched **WHILE**.

Note that **WHILE** is ended by **WEND** and not by **END WHILE**.

Be aware that the WHILE/WEND statement is somewhat redundant as a language feature. You can do exactly the same thing (and more) using a DO WHILE/LOOP. As a coding style, you may wish to standardize on using DO statements rather than WHILE statements.

STOP | HALT	STOP HALT([ret-code,] [str-msg] [,str-msg] ...)

Stops execution of the script. In a stand-alone **thinBasic** program, STOP would halt the program execution. Because **thinBasic** is being used as a macro script engine, STOP returns control to SPFLite, which returns you to the edit session from which you originally launched the **thinBasic** macro.

The HALT command optionally allows you to set a macro Return code and message before terminating.

String Handling

Note: The following functions should not be considered a complete list of all available **thinBasic** string functions, but this set will probably do for 99% of your requirements. If time permits, explore the full **thinBasic** Help document to review the complete set. Enter `HELP thinBasic` at the SPFLite command prompt to open the **thinBasic** help file.

ASC	`Number = ASC(string-expression [,position])`

ASC returns the character code of a particular character in the *string-expression*. The returned value will be in the range of 0 to 255.

The optional *position* parameter determines which character is to be checked. The first character is one, the second two, etc. If the *position* parameter is missing, the first character is presumed. If *position* is negative, ASC counts from the end of the string in reverse. That is, -1 specifies the last character, -2 specifies the second to last character, etc.

CHOOSE$	`String = CHOOSE$(index, string-expression [,string-expression] ... )`

Return one of several values, based upon the value of an index. It returns one of the parameters based upon the value of *index*. That is, if *index* is one, choice1 is returned. If two, choice2 is returned, etc. If *index* is not equal to one of the choice values, "" (zero-length string) is returned.

CHR$	`String = CHR$(expression [,expression] [,...] )` `String = CHR$(numvar1 to numvar2 [,...] )`

This function creates a string of characters. Arguments must be normal characters, or codes in the range of 0 to 255.

CHR$** creates and returns a string. There are two forms of arguments available, and they may be intermixed in a single **CHR$ function. The created string may contain no characters, one character, or multiple characters, depending upon the arguments you use. You may specify any number of arguments for this function.

If the argument is a numeric expression, it is translated into the character defined by that number. A character code of -1 is treated as a special case. If you use it as an argument, **CHR$** returns an empty (zero length) string for that character. For example, **CHR$(65, -1, 66)** returns **"AB"**.

CHR$(numvar1 TO numvar2) returns a sequence of all characters from **CHR$(numvar1)** through **CHR$(numvar2)** inclusive. The characters may be ascending or descending in sequence. For example, **CHR$(65 TO 70)** returns the string **"ABCDEF"**. **CHR$(52 T0 50)** returns the string **"432"**, and **CHR$(65 TO 65)** returns the string **"A"**.

CHR$ complements the **ASC** function, which returns the numeric character code of a

nominated character in a string.

$CRLF	$CRLF

This function returns a Carriage Return / Line Feed pair. Equivalent to CHR$(13, 10)

$DQ	$DQ

This function returns a Double-Quote character Equivalent to CHR$(34)

CSET$	Result-String = CSET$(string-expression, str-len [USING string-exression2])

Return a string containing a centered (padded) string. **CSET$** centers the string *string-expression* into a string of *str-len* characters.

If *string-expression2* is "" (zero-length string) or is not specified, **CSET$** pads string-expression with space characters. Otherwise, **CSET$** pads the string with the first character of string-expression2.

If *string-expression* is shorter then *str-len*, **CSET$** centers *string-expression* within *Result-String*, padding both sides as described above; otherwise, **CSET$** returns the left-most *str-len* bytes of *string-expression*.

DATE$	Result-String = DATE$

Returns the system date. You can assign DATE$ to a string variable, which stores 10 characters in the form "mm-dd-yyyy", where mm represents the month, dd the day, and yyyy the year.

EXTRACT$	Result-String = EXTARCT$([start,] string-expression, [ANY] Match-Str

EXTRACT$ returns a sub-string of *String-Expression*, starting with its first character (or the character specified by *start*) and up to (but not including) the first occurrence of *Match-Str*. If *Match-Str* is not present in *String-Expression*, or either string parameter is nul, all of String-Expression is returned.

start is the optional starting position to begin extracting. If *start* is not specified, it will start at position 1. If *start* is zero, or beyond the length of *String-Expression*, a nul string is returned. If *start* is negative, the starting position is counted from right to left: if -1, the search begins at the last character; if -2, the second to last, and so forth.

String-Expression is the string from which to extract. *Match-Str* is the string expression to extract up to. **EXTRACT$** is case-sensitive.

If the ANY keyword is included, *Match-Str* specifies a list of single characters to be searched

for individually, a match on any one of which will cause the extract operation to be performed up to that character.

A similar function to **EXTRACT$** is **PARSE$**, which extracts delimited substrings from a string.

FORMAT$	Result-String = FORMAT$(numeric-expression [, Format-Mask])

Returns a string version of *numeric expression*, formatted according to the *Format-Mask* operand.

If no *Format-Mask* operand is present, the number will be returned as a maximum of 16 significant digits.

> Note: If you wish to just concatenate a string and a numeric value without formatting, this can be done with a simple expression of the form *string-constant + numeric-expression* or *string-constant & numeric-expression.*

Format-Mask

Format characters that will determine how *numeric-expression* should be formatted. This expression is termed the mask. There may be up to 18 digit-formatting digits on either side of the decimal point. The mask may not contain literal characters unless each character is preceded with a backslash (\) escape character, or the literal characters are enclosed in quotes.

Format-Mask may contain one, two or three formatting masks, separated by semicolon (;) characters:

One mask — If *Format-Mask* contains just one format mask, the mask is used to format all possible values of *numeric-expression*. For example:

```
x$ = FORMAT$(z, "000.00")
```

Two masks — If *Format-Mask* contains two format masks, the first mask is used for positive values (=> 0), and the second mask is used for negative values (< 0). For example:

```
x$ = FORMAT$(-100, "+00000.00;-000")
```

Three masks — If *Format-Mask* contains three masks, the first mask is used for positive values (> 0), the second mask for negative values (< 0), and the third mask is used if *numeric-expression* is zero (0). For example:

```
FOR y = -0.5 TO 0.5 STEP 0.5

x$ = FORMAT$(y, "+.0;-.0; .0")

NEXT y
```

Digit placeholders in a mask do not have to be contiguous. This allows you to format a single number into multiple displayed parts. For example:

```
A$ = FORMAT$(123456, "00\:00\:00") ' 12:34:56
```

The following table shows the characters you can use to create the user-defined format

strings (masks) and the definition of each formatting character:

Character	Definition
Empty string	["" (a zero-length string)] No formatting takes place. The number is formatted similarly to STR$, but without the leading space that STR$ applies to non-negative numbers.

```
A$=FORMAT$(0.2)        ' .200000002980232

A$=FORMAT$(0.2!, "")   ' .200000002980232

A$=FORMAT$(123)        ' 123

A$=FORMAT$(123456)     ' 123456
```

0	[zero] Digit placeholder. A digit or 0 will be inserted in that position.

If there is a digit in numeric-expression in the position where the 0 appears in the format string, return that digit. Otherwise, return "0". If the number being formatted has fewer digits than there are zeros (on either side of the decimal point) in the format expression, leading or trailing zeros are added. If the number has more digits to the right of the decimal point than there are zeros to the right of the decimal point in the format expression, the number is rounded to as many decimal places as there are zeros in the mask.

If the number has more digits to the left of the decimal point than there are zeros to the left of the decimal point in the format expression, the extra digits are displayed without truncation. If the numeric value is negative, the negation symbol will be treated as a decimal digit. Therefore, care should be exercised when displaying negative values with this placeholder style. In such cases, it is recommended that multiple masks be used.

```
' Numeric padded with leading zero characters

A$ = FORMAT$(999, "00000000")      ' 00000999
```

#	[Number symbol] Digit placeholder. If there is a digit for this position, it is replaced with a digit, nothing, or a user-specified character.

Unlike the 0 digit placeholder, if the numeric value has the fewer digits than there are # characters on either side of the decimal placeholder, then it will either:

a) Omit this character position from the final formatted string; or

Substitute a user-specified replacement character if one has been defined (see the asterisk (*) character for more information). To specify leading spaces, prefix the mask with "* " (asterisk and a space character).

For example:

```
' No leading spaces and trailing spaces

A$ = FORMAT$(0.75, "####.###")     ' 0.75

' Up to 3 Leading spaces before decimal

A$ = FORMAT$(0.75, "* ##.###")     ' 0.75
```

```
' Using asterisks for padding characters

A$ = FORMAT$(0.75, "*=##.###")      ' ===0.75=
```

FORMAT$ may also return a string that is larger than the number of characters in the mask:

```
A$ = FORMAT$(999999.9, "#.#")       ' 999999.9
```

. [period] Decimal placeholder. Determines the position of the decimal point in the resultant formatted string.

If any numeric field is specified to the left of the decimal point, at least one digit will always result, even if only a zero. The zero is not considered to be a "leading" zero if it is the only digit to the left of the decimal. Placing more than one period character in the *Format-Mask* string will produce undefined results.

% [percent] Percentage placeholder. *numeric-expression* will be multiplied by 100, and adds a trailing percent symbol. For example:

```
x$ = FORMAT$(1 / 5, "0.0%")          ' 20.0%
```

, [comma] Thousand separator. Used to separate thousands from hundreds within a number that has four or more digits to the left of the decimal point. In order to be recognized as a format character, the comma must be placed immediately after a digit placeholder character (also see Restrictions below).

```
A$ = FORMAT$(1234567, "#,")          ' 1,234,567

A$ = FORMAT$(12345, "#,.00")         ' 12,345.00

A$ = FORMAT$(12345, "#.00,")         ' 12,345.00

A$ = FORMAT$(1212.46, "$00,000.00") ' $01,212.46

A$ = FORMAT$(1000, """#""#,")        ' #1,000

A$ = FORMAT$(1234567, "0,")          ' 1,234,567
```

*x [asterisk] Digit placeholder and fill-character. Instructs FORMAT$ to insert a digit or character "x" in that position. If there is a digit in *numeric-expression* at the position where the * appears in the format string, that digit is used; otherwise, the "x" character is used (where "x" represents your own choice of character). The *x specifier acts as two digit (#) fields.

```
A$ = FORMAT$(9999.9,"$**####,.00")   ' $**9,999.90

A$ = FORMAT$(0,"$*=###0,.00#")       ' $=====0.00=

A$ = FORMAT$(0,"$* ####0,.00")       ' $ 0.00
```

E- e- E+ e+ [e] Scientific format. FORMAT$ will use scientific notation in the formatted output. Use E- or e- to place a minus sign in front of negative exponents. Use E+ or e+ to place a minus sign in front of negative exponents and a plus sign in front of non-negative exponents.

In order to be recognized as a format sequence, the E-, e-, E+, or e+ must be placed between two digit placeholder characters. For example:

```
A$ = FORMAT$( 99.999, "0.0E-##")    ' 1.0E2

A$ = FORMAT$(-99.999, "0.0E-##")    ' -1.0E2

A$ = FORMAT$( 99.999, "0.0E+##")    ' 1.0E+2

A$ = FORMAT$(-99.999, "0.0E+##")    ' -1.0E+2

A$ = FORMAT$(0.1, "0.0e+##")        ' 1.0e-1
```

" [double-quote] Quoted string. FORMAT$ treats all characters up to the next quotation mark as-is, without interpreting them as digit placeholders or format characters. Also see backslash. For example:

```
A$ = FORMAT$(5.55, """XYZ=""#.##\#") ' XYZ=5.55#

A$ = FORMAT$(25, """x=""#")          ' x=25

A$ = FORMAT$(999, """Total ""#")     ' Total 999
```

\x [backslash] Escaped character prefix. FORMAT$C treats the character "x" immediately following the backslash (\) as a literal character rather than a digit placeholder or a formatting character. Many characters in a mask have a special meaning and cannot be used as literal characters unless they are preceded by a backslash.

The backslash itself is not copied. To display a backslash, use two backslashes (\\). To display a literal double-quote, use two double-quote characters.

IIF$	Result-String = IIF$(numeric-expression, true-string, false-string)

IIF$ returns one of two values based on *numeric-expression.* If *numeric-expression* evaluates to TRUE (non-zero), the *true-string* is returned, else the *false-string* is returned. Compare to the **IIF** function.

INSTR	Result-var = INSTR(([Position,] Main-Str, [ANY] Match-Str)

INSTR returns the position of *Match-Str* within *Main-Str*. The return value is indexed to one, while zero means "not found".

Position specifies the character position to begin the search. If *Position* is one or greater, *Main-Str* is searched left to right. The value one starts at the first character, two the second, etc. If *Position* is -1 or less, *Main-Str* is searched from right to left. The value -1 starts at the last character, -2 the second to last, etc. If *Position* is not given, the default value of +1 is assumed.

```
x& = INSTR("xyz", "y")     ' returns 2

x& = INSTR("xyz", "a")     ' returns 0

a$ = "My Dog" : b$ = " "
```

```
x& = INSTR(a$, b$)          ' returns 3
```

It is important to note that in all cases, even when *Position* is negative, the return value of **INSTR()** is the absolute position of the match, from left to right, starting with the first character.

ANY

If the **ANY** keyword is included, *Match-Str* specifies a list of single characters. INSTR searches for each of these characters individually. As soon as any one of these characters is found, INSTR returns the position of the match.

```
x& = INSTR(-2, "efcdef", ANY "ef")     returns a result of 5
```

INSTR is case-sensitive, meaning that upper-case and lower-case letters must match exactly in *Match-Str* and *Main-Str*.

LCASE$	Result-str = LCASE$(string-expression)

Returns a lowercase version of the specified *string-expression*.

LEFT$	Result-str = LEFT$(string-expression, Num-chars)

Num-chars specifies the number of characters in *string-expression* to be returned.

LEFT$ returns a string consisting of the left most *Num-chars* characters of its string argument. If *Num-chars* is greater than or equal to the length of *string-expression*, all of *string-expression* is returned. If *Num-chars* is zero, **LEFT$** returns an empty string.

LEN	Result-var = LEN(string-expression)

LEN *will return the current length of string-expression.*

LSET$	Result-str = LSET$(string-expression, str-len [USING pad-string])

LSET$ left-aligns the string *string-expression* into a string of *str-len* characters.

USING

If *pad-string* is "" (a zero-length string) or is not specified, LSET$ pads *string-expression* with space characters. Otherwise, **LSET$** pads the string with the first character of *pad-string*.

If *string-expression* is shorter then *str-len*, LSET$ left-justifies *string-expression* within *result-str*, padding the right side as described above; otherwise, **LSET$** returns the left-most *str-len* bytes of *string-expression*.

LTRIM$	Result-str = LTRIM$(string-to-trim [, [ANY] Chars-to-trim])

String-to-trim is the string-expression from which to remove characters, and *Chars-to-trim* is the string-expression containing the characters to remove.

If *Chars-to-trim* is not specified, **LTRIM$** removes leading spaces. LTRIM$ returns a sub-string of *String-to-trim*, from the first non-*Chars-to-trim* (or non-space) to the end of the string. If *Chars-to-trim* (or a space) is not present at the beginning of *String-to-trim*, all of *String-to-trim* is returned.

If the **ANY** keyword is included, *Chars-to-trim* specifies a list of single characters to be searched for individually. A match on any one of these as a leading character will cause the character to be removed from the result.

LTRIM$ is case sensitive

MCASE$	Result-str = MCASE$(string-expression)

MCASE$ returns a string equivalent to *string-expression*, except that the first letter of each word is capitalized, while the remaining characters are forced to lowercase. A word is considered to be a consecutive series of letters.

MID$	Result-str = MID$(string-expression, start-loc [, length])

The **MID$** function returns a part of a string-expression. The *start-loc* specifies the location of the first character of the extracted string. The *length* specifies how many characters, starting at *start-loc* are to be returned.

If *length* is omitted, or there aren't enough characters in *String-expression*, all remaining characters are returned. If there are no characters at the *Start-loc* position, an empty string is returned.

PARSE$	Result-str = PARSE$(string-expression, [ANY] str-delimiter, index)

Returns a delimited field from *string-expression. string-expression* is the string to parse. If *string-expression* is "" (a zero-length string) or contains no delimiter character(s), the string is considered to contain exactly one field. In this case, PARSE$ will return *string-expression*.

The *str-delimiter* contains delimiter character(s). A delimiter is a character, list of characters, or string, that is used to mark the end of a field in string-expression. For example, if you consider a sentence to be a list of words, the delimiter between the works is a space (or punctuation). A delimiter is not considered part of a field, but as the divider between fields, so the delimiter is never returned by PARSE$.

If *str-delim* is not specified or is "" (a zero-length string), standard comma-delimited (optionally quoted) fields are presumed. In this case only, the following parsing rules apply. If a standard field is enclosed in optional quotes, they are removed. If any characters appear between a quoted field and the next comma delimiter, they are discarded. If no leading quote is found, any leading or trailing blank spaces are trimmed before the field is returned.

Delimiters are case-sensitive, so capitalization may be a consideration.

ANY

If the **ANY** keyword is used, *str-delimiter* contains a set of characters, any of which may act as a delimiter character. If the *ANY* keyword is omitted, the entire *str-delimiter* string acts as a single delimiter.

index

A variable or expression that specifies the delimited field number to return. The first field is 1, and so on up to the maximum number of fields contained in string-expression, which may be determined with the PARSECOUNT function. If *index* is negative, string-expression is parsed from right to left. In this case, *index* = -1 returns the last field in string-expression, -2 returns the second to last, etc. If *index* evaluates to zero, or is outside of the actual field count, an empty string is returned.

PARSECOUNT	Result-str = PARSE$(string-expression, [ANY] str-delimiter)

Return the count of delimited strings in a *string-expression*. PARSECOUNT uses the same rules as PARSE$ in the determination of fields within string-expression. Individual fields within string-expression are evaluated, and the tally of the fields forms the result value.

It is important to note that PARSECOUNT may only be used with string data which contains variable length sub-fields, each of which is separated by a delimiter. To determine the count of fixed length data, divide the String-Expression length by the sub-field length. If this function is used with fixed length data, the results are undefined.

string-expression

This is the string to examine and parse. If String-Expression is "" (a zero-length string) or contains no delimiter character(s), the string is considered to contain exactly one sub-field. In this case, PARSECOUNT returns the value 1.

str-delimiter

This defines one or more characters to use as a delimiter. To be valid, the entire delimiter must match exactly, but the delimiter itself is never returned as part of the field.

If *str-delimiter* is not specified, or contains an empty string, special rules apply. The delimiter is assumed to be a comma. Fields may optionally be enclosed in quotes, and are ignored before the result string is returned. Any characters that appear between a quote mark and the next comma delimiter character are discarded. If no leading quote is found, any leading or trailing quotes are trimmed before the result string is returned.

ANY

If the ANY keyword is used, *str-delimiter* contains a set of characters, any of which may act as a delimiter character. If the ANY keyword is omitted, the entire *str-delimiter* string acts as a single delimiter.

REMOVE$	Result-str = REMOVE$(string-expression, [ANY] chars-to-remove)

Return a copy of a string with characters or strings removed.

string-expression is the string from which to remove characters. *chars-to-remove* is the string expression to remove all occurrences of. If *chars-to-remove* is not present in *string-expression*, all of *string-expression* is returned intact.

ANY

If the **ANY** keyword is included, *chars-to-remove* specifies a list of single characters to be searched for individually, a match on any one of which will cause that character to be removed from the result.

REMOVE$ is case-sensitive.

REPEAT$	Result-str = REPEAT$(number-of-times, string-expression)

Returns a string consisting of multiple copies of the specified string. *number-of-times* is an expression, constant or variable, specifying the number of copies of *string-expression* to be included in the result.

string-expression is the string to be duplicated.

REPLACE$	Result-str = REPLACE$(string-expression, [ANY] match-string, new-string)

Within a specified string , replace all occurrences of one string with another string. The **REPLACE$** statement replaces all occurrences of *Match-String* in *string-expression* with *New-String*. The replacement can cause string-expression to grow or condense in size. *string-expression* must be a string variable; *Match-String* and *New-String* may be string expressions. **REPLACE$** is case-sensitive. When a match is found, the scan for the next match begins at the position immediately following the prior match.

ANY

If you use the **ANY** option, within *String-expression*, each occurrence of each character in *Match-String* will be replaced with the corresponding character in *New-String*. In this case, *Match-string* and *New-String* must be the same length, because there is a one-to-one correspondence between their characters.

RIGHT$	Result-str = RIGHT$(string-expression, Num-chars)

Num-chars specifies the number of characters in *string-expression* to be returned.

RIGHT$ returns a string consisting of the right most *Num-chars* characters of its string argument. If *Num-chars* is greater than or equal to the length of *string-expression*, all of *string-expression* is returned. If *Num-chars* is zero, **RIGHT$** returns an empty string.

RSET$	Result-str = RSET$(string-expression, str-len [USING pad-string])

RSET$ left-aligns the string *string-expression* into a string of *str-len* characters.

USING

If *pad-string* is "" (a zero-length string) or is not specified, **RSET$** pads *string-expression* with space characters. Otherwise, **RSET$** pads the string with the first character of *pad-string*.

If *string-expression* is shorter then *str-len*, **RSET$** right-justifies *string-expression* within *result-str*, padding the left side as described above; otherwise, **RSET$** returns the right-most *str-len* bytes of *string-expression*.

RTRIM$	Result-str = RTRIM$(string-to-trim [, [ANY] Chars-to-trim])

String-to-trim is the string-expression from which to remove characters, and *Chars-to-trim* is the string-expression containing the characters to remove.

String-to-trim is the *string-expression* from which to remove characters, and *Chars-to-trim* is the string expression specifying the characters that should be removed from the right hand side of *String-to-trim*.

If *Chars-to-trim* is not specified, **RTRIM$** removes trailing spaces. **RTRIM$** returns a sub-string of *String-to-trim*, from the beginning of the string to the character preceding the consecutive occurrences of *Chars-to-trim* (or space), which continues to the end of the original string . If *Chars-to-trim* (or a space) is not present at the end of *String-to-trim*, all of *String-to-trim* is returned.

If the **ANY** keyword is included, *Chars-to-trim* specifies a list of single characters to be searched for individually - a match on any one of which as a trailing character will cause the character to be removed from the result.

RTRIM$ is case-sensitive.

STR$	Result-str = STR$(num-expression [, digits])

Return the representation of a number in printable string form.

STR$ returns the string form of a variable or expression in printable text form. *digits* is an optional expression specifying the maximum total number of digits to appear in the result. If *num-expression* is greater than or equal to zero, **STR$** adds a leading space character; if *num-expression* is less than zero, **STR$** adds a leading negation (minus) character.

For example, **STR$(14)** returns a three-character string, of which the first character is a space, and the second and third are "1" and "4".

digits specifies the maximum number of significant digits (1 to 18) desired in the result.

The complementary function is **VAL**, which takes a string argument and returns the numeric equivalent.

Thus, **number = VAL(STR$(number))**.

A numeric value may also be converted to a string with the **TSTR$** function which returns the string without the leading blank or - character, or the FORMAT$ function. **FORMAT$** is capable of many additional formatting options.

STRCONST$	Result-str = STRCONST$(str-expression)

Return a string representing the mnemonic string passed as *str-expression*. *str-expression* may be any of "CRLF", "TAB", "CR", "LF", "FF", "VT", "SPC", "DQ", "NUL", "ESC"

STRDELETE$	Result-str = STRDELETE$(str-expression, start, count)

Delete a specified number of characters from *str-expression*.

Returns a string based on copying *str-expression*, but with *count* characters deleted starting at position *start*. The first character in the string is position 1, etc.

STRINSERT$	Result-str = STRINSERT$(str-expression, ins-expression, position)

Insert *ins-expression* string at a specified position within *str-expression*.

Returns a string consisting of the string expression *str-expression*, with the string expression ins-expression inserted at *position*. If *position* is greater than the length of str-expression, ins-expression is appended to str-expression. The first character in the string is position 1, etc.

STRREVERSE$	Result-str = STRREVERSE$(str-expression)

Reverse the contents of str-expression.

TALLY	Num-var = TALLY(str-expression, [ANY] Match-string)

Count the number of occurrences of specified characters or strings within *str-expression* .

str-expression is the string in which to count characters. *Match-String* is the string expression to count all occurrences of. If *Match-String* is not present in *str-expression*, zero is returned. When a match is found, the scan for the next match begins at the position immediately following the prior match.

ANY

If the **ANY** keyword is included, *Match-String* specifies a list of single characters to be searched for individually: a match on any one of which will cause the count to be incremented for each occurrence of that character. Note that repeated characters in *Match-String* will not increase the tally. For example:

```
X = TALLY("ABCD", ANY "BDB")      ' returns 2, not 3
```

TALLY is case-sensitive, so be wary of capitalization.

TIME$	Result-str = TIME$

Return a string with system time in the format of HH:MM:SS using 24-hour time.

TRIM	Result-str = TRIM$(str-expression [, [ANY] Chars-To-Trim)

Removes leading and trailing characters or substrings from *str-expression*.

TRIM$** combines the functionality of **LTRIM$ and **RTRIM$** into a single function. *str-expression* is the string expression from which to remove characters, and *Chars-To-Trim* is the string expression to remove leading and trailing occurrences. If *Chars-To-Trim* is not specified, **TRIM$** removes leading and trailing spaces.

ANY

If the **ANY** keyword is included, *Chars-To-Trim* specifies a list of single characters to be searched for individually, a match on any one of which as a leading or trailing character will cause the character to be removed from the result.

TSTR$	`Result-str = TSTR$(num-expression [, digits] )`

Return the representation of a number in printable string form.

TSTR$ returns the string form of a variable or expression in printable text form. *digits* is an optional expression specifying the maximum total number of digits to appear in the result.

For example, **TSTR$(14)** returns a two-character string, "14".

digits specifies the maximum number of significant digits (1 to 18) desired in the result.

The complementary function is **VAL**, which takes a string argument and returns the numeric equivalent.

Thus, **number = VAL(TSTR$(number))**.

A numeric value may also be converted to a string with the FORMAT$ function. **FORMAT$** can optionally perform many additional formatting functions.

UCASE$	`Result-str = UCASE$(string-expression)`

Returns an uppercase version of the specified *string-expression*.

VAL	`Num-var = VAL(string-expression)`

Convert a text string to a numeric value.

The **VAL** function converts a string argument to a number. Leading white-space characters (spaces, tabs, carriage-returns, and linefeeds) are skipped and ignored. Evaluation of the number continues until a non-numeric character is found, or the end of the string is reached. If no number is found, the VAL() function returns zero (0). Format characters (like commas) are not allowed, and will cause early termination of the evaluation.

VAL interprets the letters "e" and "d" (and "E" and "D") as the symbols for exponentiation and scientific notation:

```
i& = VAL("10.101e3")        ' 10101 ~ 10.101*(10^3)

j& = VAL("2D4")             ' 20000 ~ 2 * (10 ^ 4)
```

Hexadecimal, Binary and Octal conversions

VAL can also be used to convert string arguments that are in the form of Hexadecimal, Binary and Octal numbers. Hexadecimal values should be prefixed with "&H" and Binary with "&B". Octal values may be prefixed "&O", "&Q" or just "&". If the string-expression contains a leading zero, the result is returned as an unsigned value; otherwise, a signed value is returned. For example:

```
i& = VAL("&HF5F3")          ' Hex, returns -2573 (signed)

j& = VAL("&H0F5F3")         ' Hex, returns 62963 (unsigned)
```

```
x& = VAL("&B0100101101")    ' Binary, returns 301 (unsigned)
y& = VAL("&O4574514")       ' Octal, returns 1243468 (signed)
```

Valid hex characters include 0 to 9, A to F (and a to f). Valid Octal characters include 0 to 7, and binary 0 to 1.

VAL stops analyzing *string-expression* when non-numeric characters are encountered. When dealing with Hexadecimal, Binary, and Octal number systems, the period character is classified as non-numeric.

VERIFY	`Num-var = VERIFY([start,] string-expression, Match-String)`

Determine whether each character of a string is present in another string.

VERIFY returns zero if each character in *string-expression* is present in *Match-String*. If not, it returns the position of the first non-matching character in *string-expression*.

This function is useful for determining if a string contains only digits. For example, VERIFY(NUM, "0123456789") = 0 if NUM is a numeric string.

VERIFY is case-sensitive, so capitalization matters.

If *start* evaluates to a position outside of the string on either side, or if *start* is zero, **VERIFY** returns zero.

Numeric Handling

Note: The following functions should not be considered a complete list of all available **thinBasic** numeric functions, but this set will probably do for 99% of your requirements. If time permits, explore the full **thinBasic** Help document to review the complete set. Enter `HELP thinBasic` at the SPFLite command prompt to open the **thinBasic** help file.

ABS	`Num-var = ABS(numeric-expression)`

Return the absolute value of a expression.

The absolute value of a number is its non-negative value . For example, the absolute value of -3 is 3, and the absolute value of +3 is also 3. The absolute value of 0 is 0.

ATN	`Num-var = ATN(numeric-expression)`

Return the arctangent of an argument.

ATN returns the arctangent (Inverse Tangent) of *numeric-expression*; that is, the angle whose tangent is *numeric-expression*.

The result is in radians rather than degrees. To convert radians to degrees, multiply the radian value by 180/pi, or 57.29577951308232.

CEIL	`Num-var = CEIL(numeric-expression)`

Convert a variable or expression into an value, by returning the smallest integral value that is greater than or equal to its argument.

The **CEIL** function rounds upward, returning the smallest integral value that is greater than or equal to *numeric-expression*.

For example, **`y = CEIL(1.5)`** places the value 2 into y.

CHOOSE	`Num-var = CHOOSE(index, string-expression [,string-expression] ... )`

Return one of several values, based upon the value of an index. It returns one of the parameters based upon the value of *index*. That is, if *index* is one, choice1 is returned. If two, choice2 is returned, etc. If *index* is not equal to one of the choice values, 0 is returned.

COS	`Num-var = COS(numeric-expression)`

Return the cosine of an argument.

numeric-expression is an angle specified in radians. To convert radians to degrees, multiply by 57.2957795130823##. To convert degrees to radians, multiply by 0.0174532925199433.

COS returns a value that always ranges between -1 and +1 inclusive.

DECR	`DECR  numeric-variable`

Decrement a variable by 1.

Note: A variable may also be decremented by the notation ***variable -= 1***

EXP **EXP2** **EXP10**	`Num-var = EXP(numeric-expression)`

Return a base number raised to a power. The base is **e** for **EXP**, 2 for **EXP2**, and **10** for **EXP10**.

EXP returns e to the nth power, where n is a numeric variable or expression and e is the base for natural logarithms, approximately 2.718282. Among other uses, this provides a simple way to obtain the value of e itself:

e = EXP(1)

EXP2(n) returns 2 to the nth power, where **n** is a numeric variable or expression.

EXP10(n) returns 10 to the nth power, where **n** is a numeric variable or expression.

The **EXP** functions provide a convenient alternative to the ^ operator, which works with any base.

FIX	`Num-var = FIX(numeric-expression)`

Truncate a number to an integer value.

FIX strips off the fractional part of its argument, and returns the integer part. Unlike **INT**, **FIX** does not perform any form of rounding or scaling.

FALSE **TRUE**	`Same as literal value of 0` `Same as literal value of -1`

FALSE and TRUE may be used anywhere the literal values of 0 and -1 can be used, including the VALUE clause of DIM statements.

Be aware that, while the value of TRUE is -1, a "true" value for purposes of conditional expressions such as in IF statements is a **non-zero** value, not just the value -1. This is similar to how conditions are tested in the C language. You can use TRUE and FALSE to set a flag variable, and you can test a flag for FALSE, but it's best not to test a flag for TRUE. Instead of using **IF flag = TRUE THEN**, you should simply say **IF flag THEN**. If you want to **force** a numeric expression to be exactly equal to either TRUE or FALSE, you can use the **thinBasic** function **IsTrue.**

FRAC	`Num-var = FRAC(Numeric-expression)`

Return the fractional part of a number.

FRAC returns the number after the decimal point of a number or expression. **FRAC** rounds

the result to fit the precision of Numeric Expression..

IIF	`Num-var = IIF(Num-expression, true-part, false-part)`

Return one of two values based upon a TRUE/FALSE evaluation.

If *num-expression* evaluates to TRUE (non-zero), the *true-part* is returned, else the *false-part* is returned. *num-expression* is evaluated as a normal Boolean expression.

Both *true-part* and *false-part* should generate a normal numeric return value.

Compare to the **IIF$** function.

INCR	`INCR  numeric-variable`

Increment a variable by 1.

Note: A variable may also be incremented by the notation ***variable* += 1**

INT	`Num-var = INT(Num-expression)`

INT rounds *num-expression* to the largest integral value that is less than or equal to *num_expression.*

ISFALSE **ISTRUE**	`ISFALSE(expression)` `ISTRUE(expression)`

ISTRUE returns -1 (**TRUE**) when *expression* evaluates as non-zero; otherwise, it returns zero (**FALSE**). **ISFALSE** returns -1 when *expression* evaluates as 0 (**FALSE**); otherwise, it returns zero.

Truth table

operator	*expression*	Result
ISTRUE	= 0	0 (False)
ISTRUE	<> 0	-1 (True)
ISFALSE	= 0	-1 (True)
ISFALSE	<> 0	0 (False)

MAX	`Num-var = MAX(Number1, [,Number2] ...)`

Return the argument with the largest (maximum) value.

This function takes any number of arguments and return the argument with the largest (maximum) value.

MIN	`Num-var = MIN(Number1, [,Number2] ...)`

Return the argument with the smallest (minimum) value.

This function takes any number of arguments and return the argument with the smallest (minimum) value.

MOD	`Num-var = MOD(Quotient, Divisor)`

Return the remainder of the division between two numbers.

The `MOD` operator divides the two operands, *Quotient* and *Divisor*, and returns the remainder of that division. The result of the initial division is truncated to an integer value, before the remainder is calculated.

RANDOMIZE	`[ number ]`

Seed the random number generator.

number is a seed value. If number is not specified, a random value based on the TIME is used.

Values returned by the random number generator (**RND**) depend on an initial seed value. For a given seed value, **RND** always returns the same sequence of values, yielding a predictable pseudo-random number sequence. Thus, any program that depends on **RND** will run exactly the same way each time unless a different seed is given.

RND	`Num-var = RND(From-value, To-Value)`

Return a random number.

If no operands are provided, **RND** returns a random value that is less than 1, but greater than or equal to 0. Numbers generated by **RND** aren't really random, but are the result of applying a pseudo-random transformation algorithm to a starting ("seed") value. Given the same seed, the **RND** algorithm always produces the same sequence of "random" numbers.

When From-value and To-value are provided, **RND(a, b)** returns a integer in the range of a to b inclusive. a and b can each be a numeric literal or a numeric expression.

ROUND	`Num-var = ROUND(Num-exprerssion, Num-decimal-places)`

Round a numeric value to a specified number of decimal places.

Num-decimal-places is an expression specifying the number of decimal places required in the result.

Rounding is done according to the "banker's rounding" principle: if the fractional digit being rounded off is exactly five, with no trailing digits, the number is rounded to the nearest even number. This provides better results, on average, than the simple "round up at five" approach.

```
A% = ROUND(0.5, 0)      ' 0

A% = ROUND(1.5, 0)      ' 2

A% = ROUND(2.5, 0)      ' 2
```

```
A% = ROUND(2.51, 0)      ' 3
```

SGN	`Num-var = SGN(Num-exprerssion)`

Return the sign of a numeric expression.

If *num-expression* is positive, SGN returns 1. If *num-expression* is zero, **SGN** returns 0. If *num-expression* is negative, **SGN** returns -1.

SIN	`Num-var = SIN(Num-exprerssion)`

Return the sine of its argument.

num-expression is an angle specified in radians. `SIN` returns a value between -1 and +1.

To convert radians to degrees, multiply by 57.29577951308232. To convert degrees to radians, multiply by 0.0174532925199433.

SQR	`Num-var = SQR(Num-exprerssion)`

Return the square root of its argument.

num-expression must be greater than or equal to zero. SQR calculates square roots using an optimized algorithm. That is, `y` **`= SQR(x)`** takes less time to execute than **`y = x^0.5`**.

Attempting to take the square root of a negative number does not produce any run-time errors, but the results of such an operation are undefined.

Note that unlike some other languages, the square root function is SQR and not SQRT.

User Interaction

BEEP	

Emit a tone through the computer's speaker.

INPUTBOX$	Return-str = INPUTBOX$(msg-str [, Title-str [, Default-str [,Use-Pass]]])

INPUTBOX$ is used to prompt the user to input a value. The value is returned in Return-str. Note that except for the message string, all the other parameters are optional.

Msg-str is a string expression containing the message to show and prompt the user.

Title-str is a string expression containing the title of the message box. If omitted, the title will be ThinBasic.

Default-str is a string expression containing the default starting value of the answer box. If omitted, the default value will be an empty (null) string.

Use-Pass is a TRUE/FALSE flag to allow entry of password values. If you specify this as TRUE, your input will be obscured with * asterisks. This option is probably not of much use in SPFLite macros.

INPUTBOX$ will allow you to click OK (or press Enter) or click Cancel. If you click on Cancel, the function will return an empty (null) string as a response, even if you typed something into the text box.

Note: If you use this function, you should consider using the SPF_Loop_Check function as well.

MSGBOX	num-var = MSGBOX(hParent, Message [, Dialog-Style [, Title [, Timeout [, UpdateCountDown, [, CountDownFormat]]]]])

Name	Type	Optional	Meaning
hParent	Number	No	Handle to the owner window of the message box to be message box has no owner window. Always code this
Message	String	No	A string expression that contains the message to be dis
Dialog-Style	Number	Yes	Specifies the contents and behavior of the dialog box. T flags from the groups of flags allowed below for Dialog-
Title	String	Yes	A string expression that contains the dialog box title. If this parameter is an empty string, the default title "thin
Timeout	Number	Yes	Number of milliseconds after which the message box w %IDTIMEOUT return code.

UpdateCountDown	Number	Yes	Number of milliseconds time window for window captio count down will be shown
CountDownFormat	String	Yes	Format of the count down. If not present a standard one

Display a Message Box for showing user any message and/or get user response.

NOTE: A call to MSGBOX can cause SPFLite to detect an apparent "loop" condition, because a macro that calls MSGBOX may wait indefinitely until the user clicks on OK or a similar button. You can control how SPFLite handles this situation by calling the function SPF_Loop_Check.
NOTE: **thinBasic** supports the additional MSGBOX parameters: Timeout, UpdateCountDown and CountDownFormat. See the **thinBasic** documentation for more details. These options may not be useful in SPFLite macros.

Dialog-style is an equate to control the style of the box. It may be one of the following. These are numeric equate values. If you want more than one option, you can add options together. For example, to create a Yes/No box with NO as the default button, and a "warning" icon, you can specify the dialog style as %MB_YESNO+%MB_DEFBUTTON2+%MB_ICONWARNING. You can also use the OR operator instead of a + sign.

Symbol	Meaning
%MB_OK	The message box contains one push button: OK. This is the default. Because this style is the default, and the operand can be omitted, the shortest MSGBOX call you can write is **MSGBOX(0,"message")**.
%MB_OKCANCEL	The message box contains two push buttons: OK and Cancel.
%MB_YESNO	The message box contains two push buttons: Yes and No.
%MB_YESNOCANCEL	The message box contains three push buttons: Yes, No, and Cancel.
%MB_RETRYCANCEL	The message box contains two push buttons: Retry and Cancel.
%MB_ABORTRETRYIGNORE	The message box contains three push buttons: Abort, Retry, and Ignore.
%MB_CANCELTRYCONTINUE	The message box contains three push buttons: Cancel, Try Again, Continue.
%MB_DEFBUTTON1	The first button is the default button.
%MB_DEFBUTTON2	The second button is the default button.
%MB_DEFBUTTON3	The third button is the default button.
%MB_ICONEXCLAMATION	An exclamation-point icon appears in the message box.
%MB_ICONERROR	A stop-sign icon appears in the message box.
%MB_ICONINFORMATION	An icon consisting of a lowercase letter i in a circle appears in the message box.
%MB_ICONQUESTION	A question-mark icon appears in the message box.
%MB_ICONSTOP	A stop-sign icon appears in the message box.
%MB_ICONWARNING	An exclamation-point icon appears in the message box.

%MB_ICONASTERISK	An icon consisting of a lowercase letter i in a circle appears in the message box.
%MB_ICONHAND	A stop-sign icon appears in the message box.
%MB_APPLMODAL	*This option may not be useful in SPFLite macros*
%MB_SYSTEMMODAL	*This option may not be useful in SPFLite macros*
%MB_TOPMOST	*This option may not be useful in SPFLite macros*
%MB_HELP	Adds a Help button to the message box. When the user clicks the Help button o presses F1, the system sends a %WM_HELP message to the owner. *This option may not be useful in SPFLite macros*

num-var is a variable containing the user's response, which is useful if the message box contains more than one button to click on. It can contains one of the following equate values:

Symbol	Meaning
%IDABORT	ABORT button was clicked
%IDCANCEL	CANCEL button was clicked
%IDCONTINUE	CONTINUE button was clicked
%IDIGNORE	IGNORE button was clicked
%IDNO	NO button was clicked
%IDOK	OK button was clicked
%IDRETRY	RETRY button was clicked
%IDTRYAGAIN	TRYAGAIN button was clicked
%IDYES	YES button was clicked
%IDTIMEOUT	Timeout parameter was supplied, and MSGBOX timed out waiting for a user response

NOTE: If you use this function, you should consider using the SPF_Loop_Check function as well.