TOUCH TECHNOLOGIES, INC.

Sheerpower Enhancements

Version 10 Build 31 through Build 100

Supplement to A Guide to the Sheerpower Language

Developed by Touch Technologies, Inc. October 18, 2021

Table of Contents

Sheerpower Functions

_ELAPSED System Function

Format:

```
_ELAPSED
```

When a program starts, its elapsed time running is set to zero. The _ELAPSED system function returns what the current elapsed time is in seconds.

_ELAPSED is used with the START TIMER statement to help find bottlenecks in code by timing specific sections of code.

Because Sheerpower is so fast, often the code to be timed must be put inside of a loop to produce non-zero elapsed times. In this example, we iterate over the code 10,000,000 times.

Example: _ELAPSED System Function

```
start timer
for i = 1 to 10_000_000 // underscores can be used to make larger numbers easier to read
next i
print 'Elapsed time in the code above: '; _elapsed
end

Elapsed time in the code above:  .1
```

_ROUTINE System Function

Format:

```
_ROUTINE
```

The system function _ROUTINE returns the name of the current routine as a string. This function is useful when writing to a log file where you want to include the name of the current routine.

Example: _ROUTINE System Function

```
the_id$ = ‘B98726’
do_it
end

routine do_it
  print 'Missing student ID: '; the_id$;', routine '; _routine
end routine

Missing student ID: B98726, routine DO_IT
```

ASCII() Function – Return ASCII Value of Specified String Character

Format:

```
ASCII(str_expr$[, int_expr])
```

The ASCII() function has been enhanced with the option to return the ASCII decimal value of the nth character of a string (int_expr) versus only the string's first character. This enhancement on the ASCII() function is useful for performing mathematical operations on characters in a string or to convert individual string characters into byte values.

Example: ASCII() Function – Return ASCII Value of Specified String Character

```
line$ = 'To be or not to be: that is the question.'
character_number = 12
ascii_char_value = ascii(line$, character_number)
print ascii_char_value
end

116
```

BETWEEN$() Function

Format:

```
BETWEEN$(str_expr, str_delim, str_delim[, int_expr])
```

The BETWEEN$() function provides a quick method to parse strings and extract data from them. Define the start and end delimiters, and BETWEEN$() will return the data between the two delimiters.

Use the optional *occurrence* parameter (int_expr) to define which instance of the start and end delimiting values to locate before returning the value in between.

Example: BETWEEN$() to Extract Substring Data

```
url_string$ = 'https://www.amazon.com/Looking-Upside-Down-Nothing-Blahblahananda'
domain_substring$ = between$(url_string$, '//', '/')
print domain_substring$
end

www.amazon.com
```

Example: BETWEEN$() with Optional OCCURRENCE Parameter

```
html$ = '<input type="text" value="Sally"> <input type="text" value="Johnny">'
value_substring$ = between$(html$, 'value="', '"',2)
print value_substring$
end

Johnny
```

EPS Function

Format:

```
EPS
```

EPS is defined in mathematics as *epsilon*, the "close-enough" factor. It is generally used when writing formulas that converge on a final value. If the final value is within "epsilon" of the previously calculated value, the formula has converged onto the final value.

The EPS function always returns the smallest fractional value of 0.0000000000000001.

```
 print EPS

0.0000000000000001
```

Example: EPS Function

```
 n = 5499025
 x = n
 y = 1 // initial guess

 do while x - y > eps
   x = (x + y) / 2
   y = n / x
loop
print 'Square root of'; n; 'is about:'; x
end

Square root of 5499025 is about: 2345
```

FINDITEM() Function

Format:

```
FINDITEM(array_name, expr [, int_expr1, int_expr2])
```

The FINDITEM() function is used to rapidly search an array for a given data value.

Given an array name and a string expression containing the data value to find, FINDITEM() will return either an index into the array or a '0' if the string is not found.

FINDITEM() can be called with the optional *occurrence* (int_expr1) and/or *format option* (int_expr2) parameters. Use the optional *occurrence* parameter to define which instance of the string expression found to return as an index. "1" is the default occurrence.

Example: FINDITEM() with OCCURRENCE Parameter

```
dim a$(0)
a$(0) = 'Fred'
a$(0) = 'Apple'
a$(0) = 'tom'
a$(0) = 'sue'
a$(0) = 'TOM'

print finditem(a$, 'tom', 1)   // find the first occurrence of “tom” – the default if no occurrence parameter set
print finditem(a$, 'tom', 2)   // find the second occurrence of "tom" (5)
print finditem(a$, 'tom', 3)   // find the 3rd -- this will fail and return “0”
end

3
5
0
```

Use one of the format options to perform case-sensitive or case-regardless searches on strings for exact or partial matches. Format option values are:

Format Value	Description
0	The default format if no parameter is given, performs a case-regardless search for an exact match.
1	Performs a case-sensitive search for an exact match.
2	Performs a case-regardless search for a partial match on the first characters of the string being searched.
3	Performs a case-sensitive search for a partial match on the first characters of the string being searched.

Note: in the case of a tie, there is *no guaranteed order* returned of the tied values.

Example: FINDITEM() with FORMAT Parameter

```
dim a$(0)
a$(0) = 'Fred'
a$(0) = 'Apple'
a$(0) = 'tom'
a$(0) = 'sue'
a$(0) = 'TOM'

print finditem(a$,'TOM',1,0)   // format 0 case-regardless search
print finditem(a$,'TOM',1,1)   // format 1 case-sensitive search
print finditem(a$,'T',1,2)       // format 2 case-regardless partial search
print finditem(a$,'T',1,3)       // format 3 case-sensitive partial search
end
```

```
// results

3
5
3
5
```

JOIN() Function

Format:

```
JOIN(str_var, str_expr1[, str_expr2][, str_exp3] … [, str_expr16])
```

The JOIN() function allows you to join up to 16 string expressions at a time. JOIN() is over 10,000 faster than using the "+" operator for performing large numbers of joins on string expressions. The typical use of JOIN() is to join thousands of strings in a loop, two strings at a time.

The value of the first (root) string expression is updated to contain the new joined string expression. The result of JOIN() is the length of the final string.

Example: JOIN() Function

```
 a$ = 'Sally'
 b$ = ' and Fred'
 new_length = join(a$, b$, ' and more.')
 print a$
 print 'New string length: '; new_length
 end

Sally and Fred and more.
New string length:  24
```

MAXSIZE() Function for Arrays

Format:

```
MAXSIZE(array_name)
```

The MAXSIZE() function returns the maximum number of elements in an array.

Example: MAXSIZE() Function for Arrays

```
 dim client_name(5, 6)
 print maxsize(client_name)
 end

30
```

PHASH$() Function

Format:

```
PHASH$(str_expr[, int_expr])
```

The string expression is the text to be hashed, and the optional integer for the second parameter can be used to further randomize the hashing, resulting in a "salted hashed password."

The PHASH$() function returns a 24-byte string that is URL-SAFE. It can be used in a URL without having to URLENCODE$() it first. The 24-character hash is extremely unique given any text string to hash.

The PHASH$ function can process over 5GB of text per second, making it suitable for very large datasets.

PHASH$() uses a different hashing method than HASH$() and is recommended over the use of the HASH$() function because it produces a more random hash and is faster.

Example: PHASH$() Function

```
password$ = phash$('TRUTH')
input 'Password': pwd$
if  phash$(pwd$) = password$  then
  print 'That was the correct password.'
else
  print 'That was not the correct password.'
end if
end

password?  MONEY
That was not the correct password.
```

Example: PHASH$() Function with Salt

```
password$ = phash$('TRUTH', 23993)
print password$
end

fbOdJCu87od9s50kK7zuh32W
```

POS() Function – Search from End of String

Format:

```
POS(str_expr1, str_expr2[, int_expr])
```

The POS() function now supports a negative starting position to search from the end of a string going backward to the beginning. The function returns ‘0’ if not found or the numeric position of the found string.

The system function _INTEGER is also set to contain the negative starting point for a further negative search.

Example: POS() Function – Search from End of String

```
 print pos('this isa', 'is', -1)
 print _integer
 end


6
-2
```

SELECT | END SELECT Statement with CASE OF BOOLEAN

Format:

```
SELECT
  CASE OF boolean_expr
    ---
    ---   block of code
    ---
  CASE OF boolean_expr
    ---
    ---   block of code
    ---
END SELECT
```

The SELECT | END SELECT statement using CASE OF BOOLEAN_EXPR was added to provide an alternate, simpler method of handling "N-way by Conditions" from expressing them in a series of "if then … else … elseif then … end if" statements.

Example: SELECT | END SELECT Statement with CASE OF BOOLEAN_EXPR

```
 index = 3
 file$ = ''
 client$ = 'Jones'

 select
 case of index > 5
    print 'Index is over five'
 case of file$ = ''
    print 'file is empty'
 case of client$ = 'Johnson'
  print 'Client is '; client$
 end select
 end

file is empty
```

SORTED() Function with Arrays

Format:

```
SORTED(array_name[, int_expr][, true])
```

The SORTED() function is used to work with arrays in a sorted order using the following the parameters:

Parameter	Description
sorted(array_name)	Defining only the array name returns the highest index number stored in the array.
sorted(array_name,index)	Defining the array name with the index parameter returns a sort-pointer into the array. An index of “1” is the first (lowest) value in the sorted data (e.g., array_name(sorted(array_name,1)).
sorted(array_name,index,true)	Including the third parameter “true” returns a sort-pointer using a case sensitive sort.

If data is stored into one or more arrays using the same index number, SORTED() can be used to return pointers, one at a time. Indexing the arrays by the pointer will then return the data sorted by the order specified. The example below sorts each line of text read into the array by line length.

Example: SORTED() with Arrays

```
// To run this example, create “c:\Sheerpower\Samples\utterances.csv” containing a list of phrases in column A

 open file in_ch: name 'c:\Sheerpower\Samples\utterances.csv'

// These two arrays will dynamically expand as needed
  dim data$(0)
  dim line_len$(0)

  counter = 0
  do
    line input #in_ch, eof eof?: rec$
    if eof? then exit do
    counter++
    data$(counter) = rec$
    line_len$(counter) = str$(10000 + len(rec$))  // justify the numbers for nice sorting
  loop
  close #in_ch

// Since the line_len$() array and data arrays are related (stored using the same counter)
// indexing the data$() by the sorted order of line_len$() gives us the data in line length order
  for idx = 1 to 10
    ptr = sorted(line_len$, idx) // give us the pointer to the line_len$() array that makes it look sorted
    print data$(ptr)
  next idx
```

```
// results

King size please.
She's an accountant.
She wants to hug him.
She tried it herself.
Did you get your wish?
Whose turn is it next?
```

Below is an example of two arrays that are loaded with parallel data. First, the array data is printed without sorting. Next, the data is sorted by “name” and then again sorted by “age.”

Example: SORTED() with Arrays

```
dim names$(10), ages(10)
names$(1) = 'Fred'  \ ages(1) = 15
names$(2) = 'Sam'   \ ages(2) = 12
names$(3) = 'Al'    \ ages(3) = 4

print "Printing unsorted array data:"
for i = 1 to 3
  print names$(i), ages(i)
next i
print
print "Now printing array data sorted by 'name':"
for i = 1 to 3
  index = sorted(names$,i)
  print names$(index), ages(index)
next i
print
print "Now printing array data sorted by 'ages':"
for i = 1 to 3
  index = sorted(ages, i)
  print names$(index), ages(index)
next i

Printing unsorted array data:
Fred        15
Sam         12
Al           4

Now printing array data sorted by 'name':
Al           4
Fred        15
Sam         12

Now printing array data sorted by 'ages':
Al           4
Sam         12
Fred        15
```

The next example illustrates using SORTED() for automatic case-regardless sorting and for case-sensitive sorting using the parameter TRUE.

Example: SORTED() with Arrays (Case Regardless/Case Sensitive)

```
dim b$(5) sorted // automatically sort the elements (case regardless)
b$(1) = 'Apple'
b$(2) = 'bear'
b$(3) = 'Cat'
b$(4) = 'dog'
b$(5) = 'Elephant'

print '----show from automatically sorted array--------'
for i= 1 to 5
  print b$(i)
next i
print

print '----use sorted() function -- case sensitive -------'
for i= 1 to 5
  print b$(sorted(b$, i, true))  // TRUE parameter denotes "case sensitive"
next i
print
end

----show from automatically sorted array--------
Apple
bear
Cat
dog
Elephant

----use sorted() function -- case sensitive -------
Apple
Cat
Elephant
bear
dog
```

UUID$ Function

Format:

```
UUID$([int_expr1][, int_expr2])
```

The UUID$ function returns a *universally unique identifier,* used to generate a unique key value. UUIDs are often used in database tables because they are guaranteed to be unique across all computers. In addition, using UUIDs eliminates handler programs having to monitor for key value collisions. Each time the function is called, a new unique UUID is generated.

Note: UUIDs are also often called GUIDs (Globally Unique Identifiers). For more information on UUIDs: https://en.wikipedia.org/wiki/Universally_unique_identifier.

The UUID$ function can be called alone (uses the default format "0"), or with the optional *format* and/or *length* parameters. The format value options are:

Format Value	Description
0	The default format if no parameter is given. 128-bit UUID which is then base-64 encoded into a 22 character string.
1	32 hex digits in the order specified by the RFC 1422 so that one can attach meaning to the hex digits.
2	32 hex digits in the order specified by the RFC 1422 but grouped with four dashes for a total of 36 characters.
3	Same as above, but with additional { and } at the front and back. This is the standard display format for a UUID, 38 characters in total.

Example: UUID$ with Optional Format Parameter

```
print uuid$(0)      // or print uuid$ - a base64-encoded UUID - this is the default if no parameters are given
print uuid$(1)      // a UUID in hex digits
print uuid$(2)      // a UUID in hex, with dashes between elements
print uuid$(3)      // a UUID in hex with dashes, enclosed with braces
end

tD1zL_zXdESxOHyi0_iDLQ
e_0aK9UN4EqQ8gj8X8Dbaw
69B9474C44454F51BED423893BE1B115
35BB8A4E-5FA6-4A01-BB47-D3ECD065E361
```

Sheerpower's default UUID format (0) returns 22 bytes. We achieve this space savings by converting the 128-bit standard internal UUID into a base-64 encoded string, and then stripping off the trailing two "==" that the base-64 encoding generates. Then, to make the UUID *URL-safe*, we change any generated "+" to "-" and "/" to "_".

In addition, the key values are SPARSE, and therefore cannot be easily guessed or entered in as a typo.

The default UUIDs are generated at a rate of approximately 4.5 million/second. The UUIDs using the 1, 2, and 3 formats are generated at a rate of approximately 900,000/second.

When a *length* parameter is provided with UUID$, it denotes the number of characters to return. If the length is over the length of a single UUID, then multiple UUIDs are generated as needed.

Example: UUID$ with Optional Length Parameter

```
print uuid$(0, 30)    // a default UUID with a specified length
end

J__hzcAFQkWI1vejytTe1Q2uUL2qkl
```

To further identify sets of keys, one method would be to PREFIX the UUIDs with a value to group them by type (e.g., by table name or type) or with the date the UUID was generated.

Example: UUID$ with Identification Prefix

```
client_uuid$ = "client_" + uuid$
print client_uuid$
end

client_rCS0Eo10Qki5NlLakyoQGA
```

Example: UUID$ with Identification Date Prefix

```
date_uuid$ = date$ + uuid$(0, 50)
print date_uuid$
end

20200610lpkzVZtP7EOew2adNW3shQuQGA65Gww0KbUgg0Q3-YPwjL5dVD
```

XOR$() Function

Format:

```
XOR$(str_expr1, str_expr2)
```

The XOR$() function is used for bitwise XORing of the bits contained in two strings. XOR$() can be used in conjunction with the UUID$ function to perform highspeed 'one-time-pad' encryption.

Example: XOR$() Function

```
secret$ = 'hi there'
key$ = uuid$(0, len(secret$))
encrypted$ = xor$(secret$, key$)
decrypted$ = xor$(encrypted$, key$)
dump encrypted$
print
dump decrypted$
end

ENCRYPTED$ Len: 8  Alloc: 40  SID: (0,608)  Class 1  Dtype 14 fda$offset: 160
6D732CC0 26 50 53 44 00 07 36 0D                    &PSD..6.

DECRYPTED$ Len: 8  Alloc: 40  SID: (0,610)  Class 1  Dtype 14 fda$offset: 204
6D732C70 68 69 20 74 68 65 72 65                    hi there
```

Sheerpower Statements

ABORT Command

Format:

```
ABORT [int_expr]
```

The ABORT command allows a Sheerpower program, when run from a batch file, to give a termination status. The default status is “1.” You can specify your own status by adding a positive integer after the command.

When running from a batch, it is suggested that the batch (.BAT) file code look like the following:

```
// “program_name” is the name of the Sheerpower program being run
  @echo off
  program_name

// use these to check for errors
  echo error level: %ERRORLEVEL%
  IF %ERRORLEVEL% NEQ 0 Echo An error was found
  IF %ERRORLEVEL% EQU 0 Echo No error found
```

Example: ABORT Command

```
// create a batch (.BAT) file named sheerpower_abort_test.bat and save it into c:\sheerpower\samples
// the batch file should contain the following:

  @echo off
  sp4gl.exe
  echo error level: %ERRORLEVEL%
  IF %ERRORLEVEL% NEQ 0 Echo An error was found
  IF %ERRORLEVEL% EQU 0 Echo No error found
  echo done

// run the batch file from the command line:

  c:\Sheerpower\Samples> sheerpower_abort_test.bat

// the Sheerpower console window will open. In the console window, type the following:

  abort 99

// Sheerpower will abort and return the following information in the command line console window:

  error level: 99
  An error was found
  done
   c:\Sheerpower\Samples>
```

ASK SYSTEM: SPVERSION Statement

Format:

```
ASK SYSTEM: SPVERSION str_var
```

ASK SYSTEM: SPVERSION returns the current Sheerpower version and build number as an eight-character string in the format “Vxxx.yyy” where “xxx” is the version number and “yyy” is the build number.

The ASK SYSTEM: SPVERSION statement is used on production systems where, at runtime, the code can determine which features can be used and which cannot be used based on the version of Sheerpower being run.

Example: ASK SYSTEM: SPVERSION

```
 ask system: spversion v$
 print v$
 end

V010.099
```

ASSERT Statement

Format:

```
ASSERT boolean_expr[,str_expr]
```

The ASSERT statement allows you to test code elements for expected values and cause an exception if the value returned is not as expected. If the optional string expression was given with ASSERT, then it is displayed along with the exception.

When using the ASSERT statement, what you are “asserting” is anything that results in a TRUE value with a Boolean expression.

Example: ASSERT Command

```
 filename$ = "test.spsrc"
 telephone$ = "76044233331"
 assert len(filename$) <= 100, "File name is too long!"
 assert len(telephone$) <= 10, "Telephone number is too long!"
 end

--
?? Assertion failed, details: Telephone number is too long! at MAIN.5
MAIN.5:  assert len(telephone$) <= 10,  "Telephone number is too long!"
----------
```

DIM | REDIM Statements – Dynamically Expandable Arrays

Format:

```
DIM array_name(0)
REDIM array_name(0)
```

Dynamically expandable arrays expand with use as needed in a highly optimized way. They are the preferred way to define an array when you do not know how large the array needs to be. Dimensioning an array by zero indicates that it is a dynamically expandable array.

- To store data into a dynamic array, specify element "0." This tells Sheerpower to append the data to the end of the array.
- To reduce a dynamic array to zero elements, use REDIM ARRAY_NAME().
- To find the current size of the array, use the SIZE() function.

Example: Dynamically Expandable Arrays – Appending Data

```
dim taxes(0)
taxes(0) = 45
taxes(0) = 35
print 'Array elements: '; size(taxes)
end

Array elements:  2
```

You can also specify a position to store data into as well. For example, you can specify to store data into position "100" and, if the array is not already that size, it will be after the data is stored.

Example: Dynamically Expandable Arrays – Specifying Data Storage Position

```
dim taxes(0)
taxes(100) = 45
print 'Array elements: '; size(taxes)
end

Array elements: 100
```

Note: Clusters can be used instead of arrays and are faster and more versatile.

FOR | NEXT Loop Statement – Virtually Infinite Counter

Format:

```
FOR num_var = int_expr
  …
  …  [block of code]
  …
NEXT num_var
```

The FOR / NEXT loop statement has been enhanced to utilize a virtually infinite counter. This is useful when you want to a counter that counts until some condition is met (to a limit of 10 to the 18th power).

Previously, it was required to write the FOR / NEXT loop statement with an ending counter value, such as "for index = 1 **to 1000**." The "to nn" is no longer necessary in the statement. The next example illustrates how to find all the files in the Sheerpower folder and print their names without knowing in advance how many files there are to set the counter.

Example: FOR / NEXT Loop – Virtually Infinite Counter

```
for index = 1
  file$ = findfile$('c:\Sheerpower\*.*', index)
  if file$ = '' then exit for
  print 'Found '; file$
next index
end

Found c:\sheerpower\sheerpower.ini
Found c:\sheerpower\sp$install_runonce.spsrc
Found c:\sheerpower\sp4gl.exe
Found c:\sheerpower\sp4gl_system_info.txt
Found c:\sheerpower\spdev.exe
Found c:\sheerpower\spdev_nonsp.ini
Found c:\sheerpower\spdev_nonsp_user.ini
Found c:\sheerpower\spdev_profile_user.ini
Found c:\sheerpower\spdev_sp4gl.ini
Found c:\sheerpower\spdev_sp4gl_user.ini
Found c:\sheerpower\spdev_system_info.txt
Found c:\sheerpower\spdev_tools.ini
```

IF THEN Statement – THEN Optional

In IF THEN statements, "THEN" is now optional.

Example: IF THEN Statement – THEN Optional

```
if abc > xyz then
  print 'do something'
end if
```

Can now be written without "then":

```
if abc > xyz
 print 'do something'
end if
```

OPEN FILE | OPEN TABLE Statement with ACCESS UPDATE Option

Format:

```
OPEN FILE var_name: NAME 'file_spec'
        [, ACCESS INPUT | OUTPUT [,SHARE | SHARED] | UPDATE [,SHARE | SHARED] | OUTIN [,SHARE | SHARED]
        [,UNFORMATTED] [, UNIQUE] [, OPTIMIZE OFF] [, LOCKED]
```

```
OPEN TABLE table_name: NAME 'file_spec'
        [, ACCESS INPUT | OUTIN | UPDATE ] [, LOCK] [, DATAFILE file_spec] [, OPTIMIZE OFF] [,CACHESIZE num]
```

You can now use either ACCESS OUTIN or ACCESS UPDATE to open a table or a file. for both input and output. These terms are now Sheerpower synonyms.

Example: ACCESS UPDATE Option with OPEN FILE | OPEN TABLE Statement

```
open table cl: name 'sheerpower:samples\client', access update, lock
extract table cl
  include cl(state) = 'CA'
  exclude cl(phone)[1:3] = '619'
  sort ascending by cl(last)
end extract
print 'List of California clients by last name'
for each cl
  print cl(first); ' '; cl(last), cl(phone)
next cl
close table cl
end
```

```
List of California clients by last name
Dale Derringer     (818) 223-9014
Earl Errant        (408) 844-7676
```

OPEN FILE Statement with SHARED Option

Format:

```
OPEN FILE var_name: NAME 'file_spec'
        [, ACCESS INPUT | OUTPUT [,SHARE | SHARED] | UPDATE [,SHARE | SHARED] | OUTIN [,SHARE | SHARED]
        [,UNFORMATTED] [, UNIQUE] [, OPTIMIZE OFF] [, LOCKED]
```

You can now use either SHARE or SHARED as the second optional parameter to the OPEN FILE statement. These terms are now Sheerpower synonyms.

Using SHARE or SHARED allows the file to be read by other processes as it is written to. In addition, its data is automatically flushed to disk approximately every second. This is especially useful when log files need to be viewed while a program is still running.

Example: OPEN FILE Statement with SHARED Option

```
open file text_ch: name 'test_file.txt', access output, shared
print #text_ch: 'This is the first line of text.'
print #text_ch: 'This is the second line of text.'
close #text_ch

open file text_ch: name 'test_file.txt'
line input #text_ch: line_1$
line input #text_ch: line_2$
print line_1$
print line_2$
close #text_ch

This is the first line of text.
This is the second line of text.
```

OPEN FILE Statement with HEADERS Option

Format:

```
OPEN FILE num_var: NAME str_expr[,HEADERS str_expr]
```

The new option of HEADERS with the OPEN FILE statement is used when a RESTful API requires custom CGI HTTP header data. Note that each custom header requires a chr$(13) + chr$(10) at the end.

The HEADERS option with OPEN FILE only has effect if opening a URL. Otherwise, it is ignored.

Example: OPEN FILE with HEADERS

```
client_id$ = ‘12345’
url$ = ‘https://www.some-url.com’
my_headers$ = 'IntegrationToken: ' + client_id$ + chr$(13) + chr$(10)
open file url_ch: name url$, headers my_headers$
```

If you have more than one custom header, end each one with a chr$(13) + chr$(10) as shown below:

```
url$ = ‘https://www.some-url.com’
my_headers$ = 'IntegrationToken: 123' + chr$(13) + chr$(10) &
                   + 'user_id:' + chr$(13) + chr$(10)
open file url_ch: name url$, headers my_headers$
```

OPEN FILE Statement with VERB Option

Format:

```
OPEN FILE num_var: NAME str_expr[,VERB str_expr]
```

The VERB option to the OPEN FILE statement is used when interfacing to a webserver that reacts to specific verbs such as GET, PUT, POST, DELETE, PATCH, OPTIONS, etc., also referred to as *HTTP verbs*. The most common verbs are GET or POST, but some APIs request other verbs to be used. The new VERB option allows for these cases. The VERB option only has effect if opening a URL. Otherwise, it is ignored.

Example: OPEN FILE Statement with VERB Option

```
 open file my_api: name 'http://api.myurl.com', verb "DELETE"
```

Example: Specify Custom Header with VERB & DATA Options

```
open file url_ch: name 'http://office.ttinet.com/scripts/spiis.dll/echo/echo.html',
  header 'Content-Type: text/html; charset=UTF-8' + chr$(13)+chr$(10),
  verb 'POST',
  data 'id=1234,name=Fred Smith'
  do
   line input #url_ch: eof eof?: rec$
   if eof? then exit do
   print rec$
  loop
close #url_ch
delay
```

START TIMER Statement

Format:

```
START TIMER
    …
    … block of code
    …
_ELAPSED
```

The START TIMER statement works similar to using a stopwatch. When a program starts, its elapsed time running is set to zero. The START TIMER statement resets the program's elapsed time (_ELAPSED) to zero. The _ELAPSED system function returns what the current elapsed time is in seconds.

START TIMER and _ELAPSED are used together to help find code bottlenecks by timing specific sections of code (see also ELAPSED System Function).

Because Sheerpower is so fast, often the code to be timed must be put inside of a loop to produce non-zero elapsed times. In this example, we iterate over the code 10,000,000 times.

Example: START TIMER with _ELAPSED

```
start timer
for i = 1 to 10_000_000 // underscores can be used to make larger numbers easier to read
next i
print 'Elapsed time in the code above: '; _elapsed
end

Elapsed time in the code above:  .08
```

Sheerpower Directives %

Sheerpower directives are denoted with a "%" and are invoked when the compiler builds a program and/or when a program is compiled. The following are new and enhanced directives available for use in Sheerpower programs.

%INCLUDE Directive for Web Scripting

Format:

```
%INCLUDE [CONDITIONAL | ONCE:] file_spec
```

By default the %INCLUDE directive allows you to put common subroutines into a separate file to be shared among applications. The default file type if no extension is specified is .SPINC.

For the purpose of Sheerpower web scripting, %INCLUDE has been enhanced to call in other external files that contain webpage elements, such as .HTML, .CSS, and .JS.

```
%include "myfile.html"
%include "myfile.css"
%include "myfile.js"
```

If the file type is not one that includes SHEERPOWER CODE, then upon including the file, Sheerpower surrounds the file content with [[%spscript]] tags:

```
[[%spscript]]
  ...
  … contents of the file
  …
[[/%spscript]]
```

Example: %INCLUDE Directive for Web Scripting

```
routine output_footer
  %include '@..\wwwroot\footer.html'
end routine
```

When the CONDITIONAL option is used with %INCLUDE, no error is generated if the file to be included does not exist. This allows programs to conditionally include the files.

Example: %INCLUDE Directive with CONDITIONAL Option

```
routine output_footer
  %include conditional: '@..\wwwroot\footer.html'
end routine
```

When the ONCE option is used with %INCLUDE, it guarantees that the included file will be included only once, even if the same include directive occurs elsewhere in the program or included files.

Example: %INCLUDE Directive with ONCE Option

```
routine output_footer
  %include once: '@..\wwwroot\footer.html'
end routine
```

Note: If the file that is included is modified, the program must be re-run for it to re-include the updated version of the file.

%TEST and %TEST_IGNORE Directives with OPTION TEST ON | OFF

Format:

```
OPTION TEST [ON]
  %TEST [code to compile in test mode]
  %TEST_IGNORE [code to ignore/not compile in test mode]
OPTION TEST OFF
```

Unit testing is available using two new directives, %TEST and %TEST_IGNORE. Lines of code that begin with either of these directives are ignored unless the Sheerpower program is started with the /TEST option using the command line, or the program contains the OPTION TEST ON statement.

Note: "ON" is optional to include with OPTION TEST as it is the default qualifier.

To start your Sheerpower program with /TEST using the command line, use the following format:

```
c:\Sheerpower> sp4gl.exe file_spec /test
```

Example: %TEST and %TEST_IGNORE Directives with OPTION TEST ON | OFF

```
option test

%test x$ = 'New York'
%test y$ = 'San Diego'
%test print 'The values are '; x$; ' and '; y$
%test %include 'c:\sheerpower\samples\my_module.spinc'
%test_ignore print ‘This code will not compile because it is prefixed with the %test_ignore directive.’
%test stop

option test off

%test print 'This code will not compile because we are not in test mode anymore'
end
```

```
The values are New York and San Diego
```

ASK TABLE: COUNT Statement

Format:

```
ASK TABLE table_name: COUNT num_var
```

COUNT used with the ASK TABLE statement instantly returns the number of rows in the current table.

Example: ASK TABLE: COUNT ROWS

```
open table cust: name 'sheerpower:\samples\customer'
ask table cust: count num_rows
print num_rows
close table cust
end

14
```

ASK | SET TABLE: DATA Statement

Format:

```
ASK TABLE table_name: DATA str_var
SET TABLE table_name: DATA str_expr
```

The ASK | SET TABLE: DATA statement enables you to get all the data for the current record/row. It eliminates the need to create fields such as WHOLE or ALL that stand for the entire data record.

ASK | SET TABLE: DATA is especially useful when copying data from one table to another where the two tables share the same definitions (record layout).

When doing the SET, if the data is longer than the recordsize of the current record, then the data is truncated. If the data is shorter than the recordsize of the current record, then the data is filled with nulls (byte values of zero).

The next example shows how to copy an entire record from one table and add it to another table using ASK | SET TABLE: DATA.

Note: The next example only works if you have the ARCHIVE_DATA and DAILY_DATA table files installed with the Sheerpower installer into C:\Sheerpower\Samples.

Example: ASK | SET TABLE: DATA

```
// This example only works if you have ARCHIVE_DATA and DAILY_DATA table files installed with the Sheerpower
// installer into C:\Sheerpower\Samples

 open table daily: name 'sheerpower:samples\daily_data'
 open table arch : name 'sheerpower:samples\archive_data', access update  // update is a synonym for outin

 extract table arch
 end extract
 print 'Number of records in ARCHIVE_DATA: '; _extracted
 delay

 extract table daily
   ask table daily: data row$
   print row$
 // Copy add of the data from the current daily_data table record
 // into a new record in archive_data
   add table arch
     set table arch: data row$
   end add
 end extract

 extract table arch
   ask table arch: data newrow$
   print 'The daily data is now moved to the archive data table: '; newrow$
 end extract
 print
 print 'Updated number of records in ARCHIVE_DATA: '; _extracted
 end

Number of records in ARCHIVE_DATA:  0
 [Press the ENTER key to continue]

Number of records in ARCHIVE_DATA:  0
     1AMY                    JOHNSON                      1276
      450062
100123JOSEPH                 FRANKLIN                     2333
       72800
100046ROBERT                 HOWARD                       2333
4      92000
The daily data is now moved to the archive data table:      1AMY
          JOHNSON                        1276      450062
The daily data is now moved to the archive data table: 100123JOSEPH
          FRANKLIN                       2333       72800
The daily data is now moved to the archive data table: 100046ROBERT
          HOWARD                         23334      92000
Updated number of records in ARCHIVE_DATA:  3
```

ASK | SET DATA Statement

Format:

```
ASK #ch: data str_var
SET #ch: data str_expr
```

The ASK | SET DATA statement enables you to store and retrieve temporary data without having to read/write a temporary disk file.

When opening a file, DATA:// is a designator for a data channel. A data channel does not use any disk space. Instead it uses memory. As a result, reading and writing a data channel is extremely fast.

Example: ASK | SET Statement with DATA://

```
open file mydata_ch: name 'data://'
for i = 1 to 10
 print #mydata_ch: i, sqr(i); chr$(13)+chr$(10)
next i
ask #mydata_ch: data mydata$
print mydata$
set #mydata_ch: data '' //clear data channel
ask #mydata_ch: data mydata$
print mydata$
close #mydata_ch

 1  1
 2  1.414213562373
 3  1.732050807569
 4  2
 5  2.2360679775
 6  2.449489742783
 7  2.645751311065
 8  2.828427124746
 9  3
 10  3.162277660168
```

ASK TABLE: ENGINE Statement

Format:

```
ASK TABLE table_name: ENGINE str_var
```

The ASK TABLE: ENGINE statement returns the name of the database engine that the table is controlled by.

Example: ASK TABLE: ENGINE

```
 open table pers: name 'sheerpower:\samples\personalinfo'
 ask table pers: engine e$
 print e$
 close table pers
 end

ARS
```

OPEN TABLE with CACHESIZE Option – Increase

Format:

```
OPEN TABLE table_name: NAME 'table_filename', CACHESIZE num_mb
```

The CACHESIZE option with the OPEN TABLE statement now supports 2000MB (2GB) of cache size (increased from 100MB).

Example: OPEN TABLE with CACHESIZE Option – Increase

```
 open table pay: name '@payroll', access outin, cachesize 2000
// sets up a cachesize of 2GB for the given table
```

SORT BY in EXTRACT | END EXTRACT with LENGTH Option

Format:

```
SORT [ASCENDING | DESCENDING] BY expr, LENGTH int_expr
```

The LENGTH option enhancement to the SORT statement provides a way to sort variable length data, allowing for the specification of a fixed length to achieve the sorting.

The following example first trims out trailing spaces, and then sorts up to 40 characters of the customer name. If the trimmed customer name is less than 40 characters, when sorting, Sheerpower supplies additional characters at the end to make it 40 characters.

If the length option is not specified and the data being sorted is from a simple expression, Sheerpower defaults to a length of 100.

Example: SORT BY in EXTRACT | END EXTRACT with Length Option

```
open table cust: name 'sheerpower:\samples\customer', access outin
extract table cust
  sort by trim$(cust(name)), length 40  // sort by the first 40 characters of cust(name)
end extract
for each cust
  print cust(name)
next cust
close table cust
end

Alpha Products Inc
Flower Power, Inc
Loprice Drug Stores, Inc
MicroNet Solutions
Monroe Data Systems
```

The LENGTH option to SORT is also useful when sorting “conditionally.”

In the next example, if the client’s first name begins with the letter “A” or greater, then the data will be sorted by first names. Otherwise, the data will be sorted by the client's last name.

Note: The next example only works if you have the DAILY_DATA table files installed (from the Sheerpower installer) into C:\Sheerpower\Samples.

Example: Sorting Conditionally using SORT BY with Length Option

```
open table d_data: name 'sheerpower:\samples\daily_data', access update
extract table d_data
  if d_data(first_name)[1:1] = 'A' then
    sort by d_data(first_name), length 40
  else
    sort by d_data(last_name), length 40
  end if
end extract
for each d_data
  print d_data(first_name); ' '; d_data(last_name)
next d_data
close table d_data
end

AMY JOHNSON
JOSEPH FRANKLIN
ROBERT HOWARD
```

Debugging Commands and Features

DEBUG SHOW Command

Format:

```
DEBUG SHOW [#chnl_expr:] var1[, var2][, var3][, ...]
```

The DEBUG SHOW command provides a simple method to display the names and values of variables. Optionally, the DEBUG SHOW text can be output to a file by specifying a channel number. See PRINT CLUSTER Statement for how to print cluster variables for debugging.

Example: DEBUG SHOW

```
open file log_ch: name '@debug_log.txt', access output
a = 56
b$ = 'hi there'
debug show a, b$  // display in the console
debug show #log_ch: a, b$ // output to debug_log.txt

A = 56
B$ = "hi there" (8)
```

DEBUG STACK Command

Format:

```
DEBUG STACK int_expr
```

The new DEBUG STACK command is used to store the stack details (up to the specified maximum stack depth *num_int*) into the GLOBAL symbol. The GLOBAL symbol that contains the stack information for each process is:

```
stack_nnnnnnnn
```

where “nnnnnnnn” is the 8-digit decimal value of the process ID (PID) assigned by Windows. The value returned is:

```
nn;label1;label2;...
```

where “nn” is the maximum stack depth shown (specified in int_expr), and the labels are the routine names in the stack. Steps to retrieve the PID can be found here: $DEBUG ON | $DEBUG OFF

Example: DEBUG STACK Command

```
debug stack 5
do_it
routine do_it
  do_it_more
end routine
routine do_it_more
  delay
end routine
```

```
// 1) run the sample program; when it delays, get the PID in the title of the console window
// 2) open a new Sheerpower console window and type in the following two lines of code, replacing ‘nnnnnnnn’
//     (in red below) with the PID including enough leading 0’s to be 8 digits in length:

  ask system, global ‘stack_nnnnnnnn’: value the_stack$
  print the_stack$   // press [Enter]

02;DO_IT.1;MAIN.2
```

The result of running the sample program shows: 1) the stack is two levels deep (02), 2) the top of the stack has DO_IT (DO_IT.1), and 3) that DO_IT was called from the second line in the main program (MAIN.2).

DUMP Statement

Format:

```
DUMP str_expr
```

The DUMP statement writes the contents of a string expression to the console window. The dump data includes hex values. It provides a simple way to view strings of binary data. DUMP is used primarily for debugging purposes.

Example: DUMP Statement

```
 a$ = 'Sally'
 dump a$
 end

A$ Len: 5  Alloc: 37  SID: (0,995)  Class 1  Dtype 14 fda$offset: 0
5A162D60 53 61 6C 6C 79                              Sally
```

The Alloc, SID, Class, Dtype, and fda$offset are data for internal use by Sheerpower developers and subject to change.

DUMP OF TABLES Debug Output

The crash debug output that Sheerpower provides has been enhanced to include a new section of “Dump of tables.” For each table, the dump output includes the LUN number, the last number of records extracted, and if there is a “current” record.

```
--- Dump of tables ---

ADDRESS          lun 4  Extracted: 0      Current: No  DataFile "@..\data\ADDRESS.ARS"
AUDIT            lun 5  Extracted: 0      Current: No  DataFile "@..\data\AUDIT.ARS"
BANK             lun 6  Extracted: 0      Current: No  DataFile "@..\data\BANK.ARS"
BANK_DETAIL      lun 7  Extracted: 0      Current: No  DataFile "@..\data\BANK_DETAIL.ARS"

--- End of tables ---
```

A program crash will always write out the debug dump data. Use the SHOW ALL command to force the output of the debug text without a crash to a file "[program_name]_debug.txt."

Example: DUMP OF TABLES Debug Output

```
 scanfor$ = ucase$(scanfor$)
 total      = 0

 open table vendor: name 'sheerpower:samples\vendor', access update
 extract table vendor
  include scan(vendor(name), scanfor$) > 0
  print vendor(name); tab(30); vendor(phone);' '; vendor(city);
  print tab(60); vendor(balance)
  total = total + vendor(balance)
 end extract
 print '===================='
 print using 'Total: $#,###,###': total

 show all // writes debug data to the debug file of "xxx_debug.txt"
 end

// partial results
CAMILLI, JASON            (655) 723-1569 FALLBROOK       $8,771.50
JUDD, NATHAN              (651) 728-5170 FALLBROOK       $9,220.50
PIKULA, CHANCE W.         (637) 728-5170 FALLBROOK       $9,239.50
====================
*****************
Writing all debug information to...
 C:\Users\sarce\Desktop\test_debug.txt
Done.

// Dump of tables excerpt from debug output text file:
--- Dump of tables ---

VENDOR                    lun 1   Extracted: 1383    Current: No   DataFile "sheerpower:samples\VENDOR.ARS"

--- End of tables ---
```

LIST STATS Command – Output File

Format:

```
DEBUG ON
 STATS ON
  LIST STATS [: routine_name][, routine_name][, ...]
 STATS OFF
DEBUG OFF
```

The LIST STATS command has been enhanced to always create an output file when it is executed that contains the code runtime statistics. The file name will always be in the format "[program_name]_stats.txt" and stored in the same directory location as the source program file.

Note: Both DEBUG ON and STATS ON must be in the code prior to LIST STATS for LIST STATS to work.

Example: LIST STATS – Output File

```
debug on
stats on
dim name$(0)
for i = 1
  input 'Please enter your name': name$(i)
  if _exit  then exit for
  print 'Hello, '; name$(i); '!'
next i
list stats
stats off
debug off
end

Please enter your name? Tester <--- type a name and press [Enter]
Hello, Tester!
Please enter your name? exit    <--- type 'exit' and press [Enter]
Hello, Tester!

Writing code runtime statistics to...
 C:\Sheerpower\Samples\list_stats_example_stats.txt
Done.
```

SET | CANCEL WATCH Statements

Format:

```
DEBUG ON
  SET WATCH var1[, var2][, var3][,…]
  CANCEL WATCH var1[, var2][, var3][,…]
DEBUG OFF
```

The SET WATCH command is used to open a “watch” window that displays whenever the value of the variable(s) being watched changes during debugging. CANCEL WATCH is used to turn off the debug watching feature. The WATCH only watches regular assignments. It does not watch the INPUT statement at this time.

Note: DEBUG ON must be set before using SET WATCH.

Example: SET | CANCEL WATCH

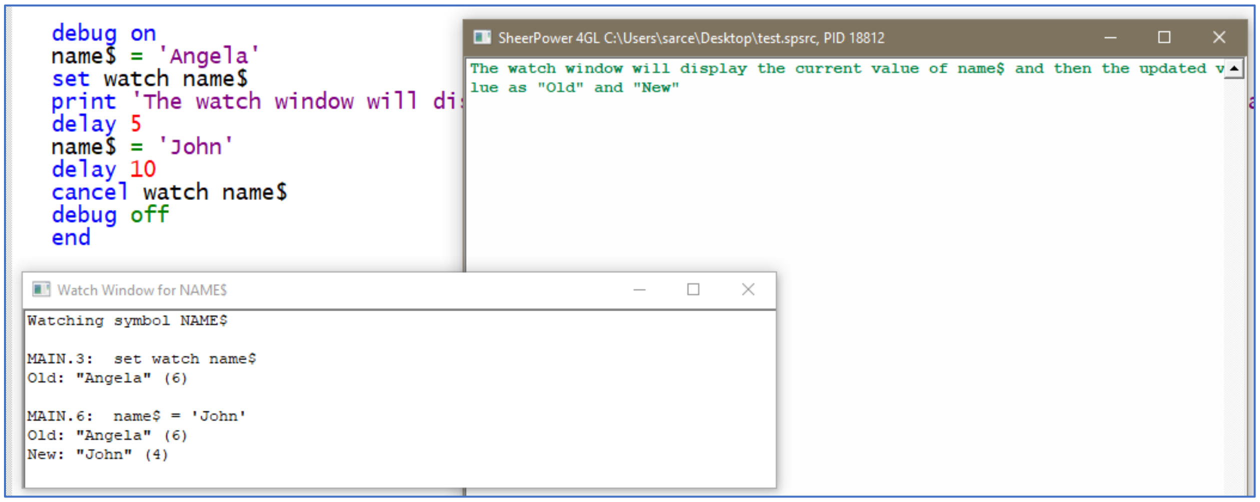
```
debug on
name$ = 'Angela'
set watch name$
print 'The watch window will display the current value of name$ and then the updated value as "Old" and "New"'
delay 5
name$ = ‘John’
delay 10
cancel watch name$
debug off
```

```
// the SP4GL console window will open
The watch window will display the current value of name$ and then the updated value as “Old” and “New”

// a separate WATCH window will open:
Watching symbol NAME$

MAIN.3:  set watch name$
Old: "Angela" (6)

MAIN.6:  name$ = 'John'
Old: "Angela" (6)
New: "John" (4)
```



```
debug on
name$ = 'Angela'
set watch name$
print 'The watch window will di
delay 5
name$ = 'John'
delay 10
cancel watch name$
debug off
end
```

SheerPower 4GL C:\Users\sarce\Desktop\test.spsrc, PID 18812

```
The watch window will display the current value of name$ and then the updated v
lue as "Old" and "New"
```

Watch Window for NAME$

```
Watching symbol NAME$

MAIN.3:  set watch name$
Old: "Angela" (6)

MAIN.6:  name$ = 'John'
Old: "Angela" (6)
New: "John" (4)
```

Debugging Running Programs – External Debug Commands

When a Sheerpower program is currently running in the background and you need to debug it without shutting it down, there are a series of commands that you can send to it. These external commands come in as a GLOBAL symbol in the form of cmd_nnnnnnn where nnnnnnn is the PID number.

Note: All *external* debug commands in Sheerpower start with a "$" symbol.

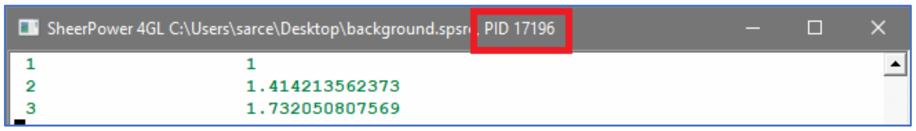
$DEBUG ON | $DEBUG OFF

Format:

```
$DEBUG ON
$DEBUG OFF
```

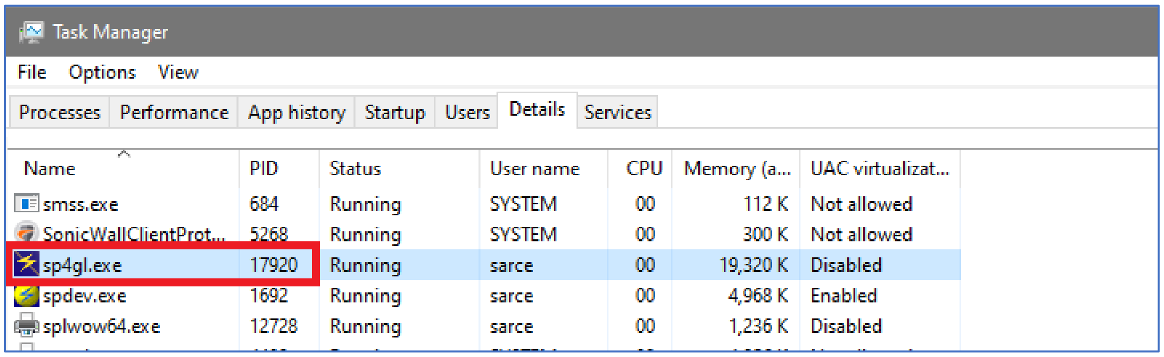
$DEBUG ON and $DEBUG OFF control the value of _DEBUG which returns either a TRUE or FALSE value. TRUE if $DEBUG ON is set. FALSE if $DEBUG OFF is set. To use $DEBUG ON and $DEBUG OFF when a program is running:

1) Get the PID of the program that is running by looking in the title bar of the open Sheerpower console window.


SheerPower 4GL C:\Users\sarce\Desktop\background.spsr - PID 17196

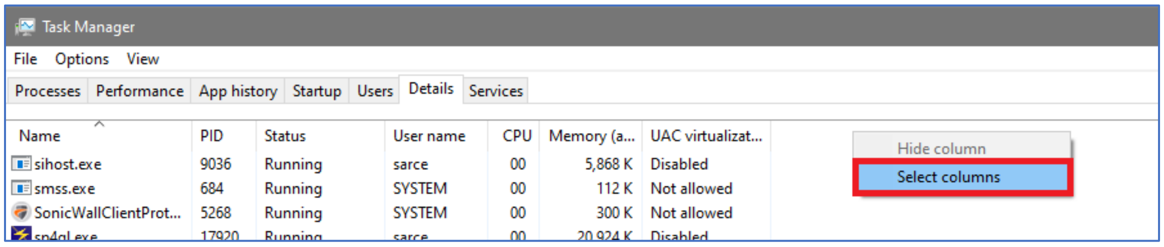
```
1              1
2              1.414213562373
3              1.732050807569
```

Alternatively, you can use the Task Manager to retrieve the PID from the Details tab.


Task Manager
File Options View
Processes Performance App history Startup Users Details Services

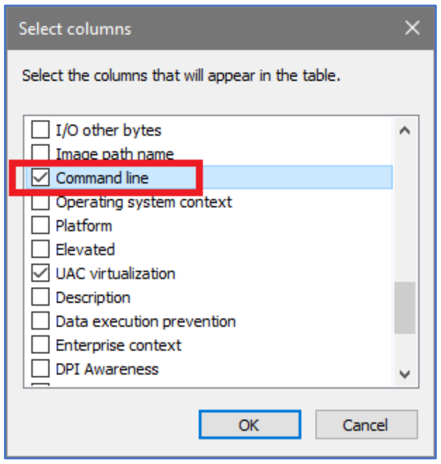
Name	PID	Status	User name	CPU	Memory (a...	UAC virtualizat...
smss.exe	684	Running	SYSTEM	00	112 K	Not allowed
SonicWallClientProt...	5268	Running	SYSTEM	00	300 K	Not allowed
sp4gl.exe	17920	Running	sarce	00	19,320 K	Disabled
spdev.exe	1692	Running	sarce	00	4,968 K	Enabled
splwow64.exe	12728	Running	sarce	00	1,236 K	Disabled

If there is more than one instance of sp4gl.exe running, the "Command line" column can be added to the table shown in the Details tab which will display the file paths for each. To add the Command line column, right click anywhere in the column headings and choose "Select columns."


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Hide column
Select columns

Name	PID	Status	User name	CPU	Memory (a...	UAC virtualizat...
sihost.exe	9036	Running	sarce	00	5,868 K	Disabled
smss.exe	684	Running	SYSTEM	00	112 K	Not allowed
SonicWallClientProt...	5268	Running	SYSTEM	00	300 K	Not allowed
sp4gl.exe	17920	Running	sarce	00	20,924 K	Disabled

Scroll down until you find “Command line” in the list, then check the box beside it and click on “OK.”

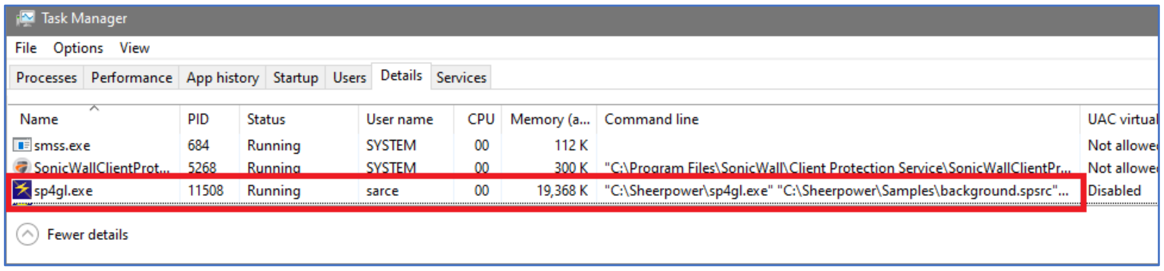


Select columns

Select the columns that will appear in the table.

- [ ] I/O other bytes
- [ ] Image path name
- [x] Command line
- [ ] Operating system context
- [ ] Platform
- [ ] Elevated
- [x] UAC virtualization
- [ ] Description
- [ ] Data execution prevention
- [ ] Enterprise context
- [ ] DPI Awareness

The file path is now displayed in the Command line column.



Task Manager
File Options View
Processes Performance App history Startup Users Details Services

Name	PID	Status	User name	CPU	Memory (a...	Command line	UAC virtual
smss.exe	684	Running	SYSTEM	00	112 K		Not allowe
SonicWallClientProt...	5268	Running	SYSTEM	00	300 K	"C:\Program Files\SonicWall\Client Protection Service\SonicWallClientPr...	Not allowe
sp4gl.exe	11508	Running	sarce	00	19,368 K	"C:\Sheerpower\sp4gl.exe" "C:\Sheerpower\Samples\background.spsrc"...	Disabled

2) Convert the PID to have enough leading zeros to be exactly 8 digits in length. For example, if the PID is 1234 then it must be converted to 00001234. If it is 52834 then it gets converted to 00052834.

3) Run a new instance of the Sheerpower console window (sp4gl.exe) on the same system as the program and enter the following command to enable debugging:

```
set system, global 'cmd_nnnnnnnn': value '$debug on' // replace nnnnnnnn with the actual PID
```

Use the example below to see how $DEBUG ON and $DEBUG OFF works when the command is executed externally from the running program.

Example: $DEBUG ON | $DEBUG OFF

```
// A simple background process that we will control externally
for idx = 1 to 500
  if _debug then
    print '... debugging'
  end if
  print idx, sqr(idx)
  delay 1
next idx
end

// see results on next page
```

```
1             1
2             1.414213562373
3             1.732050807569
4             2
5             2.2360679775
6             2.449489742783
7             2.645751311065
8             2.828427124746
9             3
10            3.162277660168
11            3.316624790355
12            3.464101615138
13            3.605551275464
14            3.741657386774
15            3.872983346207
16            4
17            4.123105625618
18            4.242640687119
19            4.358898943541
20            4.472135955
... debugging                      // $DEBUG ON has been executed, _DEBUG = TRUE
21            4.582575694956
... debugging
22            4.690415759823
... debugging
23            4.795831523313
... debugging
24            4.898979485566
... debugging
25            5
... debugging
26            5.099019513593
... debugging
27            5.196152422707
... debugging
28            5.291502622129
... debugging
29            5.385164807135
... debugging
30            5.477225575052
... debugging
31            5.56776436283
... debugging
32            5.656854249492
... debugging
33            5.744562646538     // $DEBUG OFF has been executed, _DEBUG = FALSE
34            5.830951894845
35            5.9160797831
```

$SHOW ALL

Format:

```
$SHOW ALL
```

$SHOW ALL generates a text file located in the same directory as the program that is currently running. This text file contains useful information such as call stack and recent routines, dump of variables, and dump of files. The text file name will be the same as the program name but with "_show_all.txt" appended to the end.

Note: $DEBUG ON does not need to be set in order to use $SHOW ALL.

To use $SHOW ALL when a program is running, follow the same steps as outlined in the previous section $DEBUG ON | $DEBUG OFF. The steps are summarized below:

1) Get the PID of the running Sheerpower program.
2) In a new Sheerpower console window running on the same computer as the Sheerpower program, enter the following command, making sure the PID has enough leading zeros to be 8 digits in length:

```
set system, global 'cmd_00013456': value '$show all' // replace nnnnnnnn with the actual PID
```

The generated text file will contain results that look like the following:

```
Sheerpower V010.099 show all output on 09-AUG-2021 17:30:59

Filename: C:\Sheerpower\Samples\background.spsrc
---
No error reported at MAIN.7

SYSTEXT---> The operation completed successfully.
_STRING--->
_INTEGER--> 0
---

--- Call stack and recent routines ---
MAIN.7:  delay 1
----------
--- End of calls ---

--- Dump of variables ---

IDX = 37

--- End of variables ---

--- Dump of tables ---

--- End of tables --- // continued on next page
--- Dump of open files ---
Last status: The operation completed successfully. (00000000)

channel: 000 status: 00000000 00000000  name: sys$output
  open id: 1       flags : (t)input (t)output lock stream cancel
  locked : 0       row   : 30      col   : 1
  pagelen: 30      reclen: 2048    margin: 80      zone  : 20
  cursize: 0       read  : 0       write : 74      blocks: 37
  control: 43      rewind: 0       update: 0       delete: 0

--- End of open files ---

--- End of output ---
```

New Features for Writing Improved Code

Numeric Constants Support Underscores “_”

Sheerpower now supports using underscores “_” inside of numeric constants. This makes large numbers easier to read within the code.

```
Both of these numeric values are treated the same in Sheerpower:

 a = 456_789.10
 a = 456789.10
```

Variable Name Spelling Suggestions

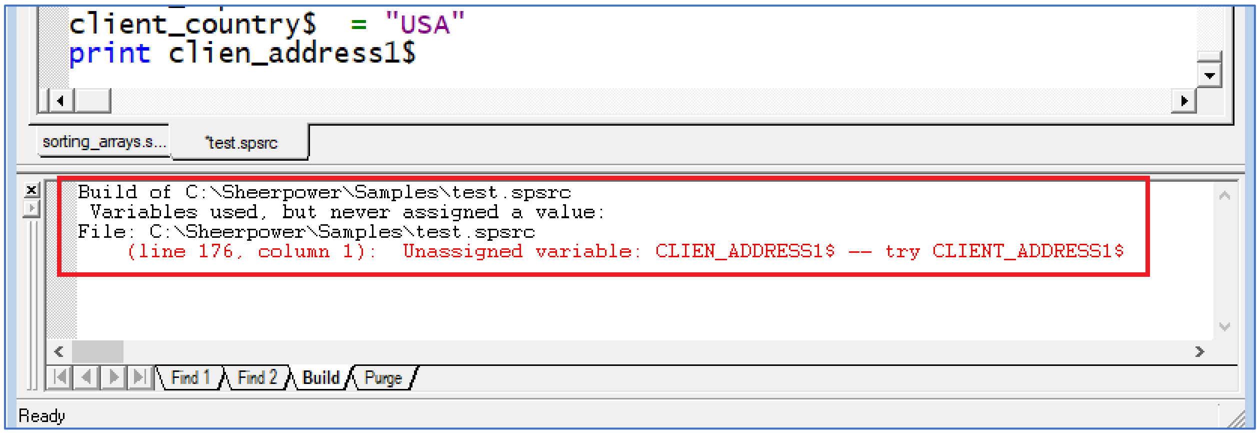
If a variable name is misspelled, Sheerpower suggests variable name spelling suggestions. The variable name suggestions are displayed in the Build tab window at the bottom.

Example: Variable Name Spelling Suggestions in SPDEV

```
 client_address1$ = "465 Highland Road"
 client_address2$ = " "
 client_zip$        = "92001"
 client_country$  = "USA"
 print clien_address1$     // variable name contains a typo
 end

// results displayed in the “Build” output window at the bottom of SPDEV

Build of C:\Sheerpower\Samples\test.spsrc
 Variables used, but never assigned a value:
File: C:\Sheerpower\Samples\test.spsrc
    (line 176, column 1):  Unassigned variable: CLIEN_ADDRESS1$ -- try CLIENT_ADDRESS1$
```



```
client_country$  = "USA"
print clien_address1$
```

```
Build of C:\Sheerpower\Samples\test.spsrc
 Variables used, but never assigned a value:
File: C:\Sheerpower\Samples\test.spsrc
    (line 176, column 1):  Unassigned variable: CLIEN_ADDRESS1$ -- try CLIENT_ADDRESS1$
```

Comments in Sheerpower Programs

Format:

```
// comment text
!  comment text (except !=)
/* comment text */
```

The above formats are all supported in Sheerpower to add comments into programs.

Inline comments in “C” style that start with /* and end with */ do not work across line boundaries yet.

Note: Any programs that have a comment line that begins with “!=” will no longer work. This is because != is now used in Sheerpower to mean “not equal to” along with <>.

Example: Comments in Sheerpower Programs

```
 dim name$(10)                               // setup array
 for i = 1 to 3                              ! begin the loop
   input 'Please enter your name': name$(i)  /* ask for a name */
   if _exit then exit for                    // end if they want
   print 'Hello, '; name$(i)                 ! print hello
 next i                                      /* end the loop */
 end

Please enter your name? Mary
Hello, Mary
Please enter your name? exit
```

Support for Both “!=” and “<>” for “NOT EQUAL TO”

Sheerpower now supports "!=" when comparing two values for “not equal to.” Previously, Sheerpower only supported the syntax of "<>."

Example: Support for Both “!=” and “<>” for “NOT EQUAL TO”

```
 a = 5
 b = 8
 if a <> b then
   print 'The numbers do not match!'
 end if

 if a != b then
   print 'The numbers still do not match!'
 end if
 end

The numbers do not match!
The numbers still do not match!
```

SCOPED ROUTINE with STATIC Variables

In a SCOPED routine, by default, all variables inside the routine are private and reset to nulls and zeros upon both entry and exit.

Any variables in the routine declared as STATIC are not reset upon entry or exit. Instead, they retain their values on re-entry to the routine.

Example: Scoped Routine

```
// ssn$ is reset upon exit, but last_date$ is not.
print 'First time:'
do_it
print 'Second time:'
do_it
end

scoped routine do_it
  print 'last_date$ = '; last_date$
  print 'ssn$ = '; ssn$
  static last_date$
   ssn$ = '123'
   if last_date$ <> date$ then last_date$ = date$
   print
end routine
```

```
First time:
last_date$ =
ssn$ =

Second time:
last_date$ = 20200601
ssn$ =
```

LOCAL ROUTINE

In Sheerpower, you can have routines, SCOPED routines, PRIVATE routines, and LOCAL routines (see Routines in Sheerpower). A LOCAL routine has a different way of setting up its namespace than a PRIVATE routine. In a PRIVATE routine, the namespace is based on the *name of the routine*. In a LOCAL routine, the namespace is based on the *namespace of the calling routine* (see Understanding Variable “Namespace” When Using Routines).

LOCAL routines are useful to segment your code when your PRIVATE or SCOPED routines become too long and/or complex.

Case Use Example for LOCAL ROUTINE

Your program contains a private routine that has grown in size so it is too difficult to follow easily and you need to break it up into smaller routines for clarity, management, and scaling. Next, you identify chunks of code within the private routine. However, the chunks of code use a lot of variables from within the private routine. This makes it impossible to quickly copy the code and paste it into a new routine. Why? Because the scoping of variables will be all wrong. Your new routine will have no easy access to the calling private routine's variables.

This scenario is when you will use LOCAL ROUTINES.

A local routine inherits the scoping of the private routine that calls it. Any variables created in the new local routine will be scoped to the private routine that called it.

A local routine must be called before it is defined (lexically). This is required so we can understand it's scoping when the local routine is defined later on in the code.

Note: When you call a local routine, you must prefix it with the word LOCAL. And when you define a local routine, you must also prefix it with the word LOCAL.

To create the local routine, simply create it **below** the parent routine with the defining LOCAL ROUTINE statement and END ROUTINE statement, and then cut/paste the block of code from the parent routine into it. Insert the call to the new local routine where the block of code was cut.

These are the steps to create a local routine:

1) From within the private routine, identify a block of code that can be isolated and inserted into its own local routine. For example, you may have a long routine that deals with financials and one section handles retrieving payroll tax information that the rest of the routine requires.
2) Choose a name for the block of code that will be moved into its own local routine (e.g., retrieve_payroll_tax).
3) Highlight/select the block of code to be moved and use [Ctrl]+[X] to cut the code into the Windows clipboard.
4) Replace the cut block of code with the call to the new local routine that includes the LOCAL prefix (e.g., **local retrieve_payroll_tax**).
5) Scroll down in the code to anywhere past the END ROUTINE statement of the calling private routine.
6) Define the new local routine – type in "LOCAL ROUTINE" and then the routine name (e.g., **local routine retrieve_payroll_tax**).
7) Paste in the code block that you cut out with [Ctrl]+[V].
8) Insert the END ROUTINE statement at the end of the new local routine.

The code pasted into the newly defined local routine of pay_get_tax will be scoped exactly as if the code was still inside of the private routine it was moved from. The local routine will have full access to variables in the routine that called it. Any private variables that the local routine needs access to must be prefixed with the name of the private routine and "$" and the variable name (e.g., do_it$total).

```
private routine my_parent
  ...
  ...
  local my_smaller_routine
  ...
  ...
end routine
```

And the new local routine is placed below the parent routine in the code:

```
local routine my_smaller_routine
  ...
  ...
end routine
```

GROUP & META GROUP Variables

Format:

```
GROUP group_name: var1[, var2][, var3][, …]
META GROUP group_name: var1[, var2][, var3][, …]
```

The feature of GROUP and META GROUP variables is available in Sheerpower. This feature makes it easy to both reset and print groups of variables with a single statement. The variables can be real, integer, string or Boolean.

A GROUP variable is created by defining the group name and which variables are to be included in the group.

A META GROUP variable is a *group of groups* and is created by defining the meta group name and which GROUPS are to be included within it.

Example: Create GROUPs and META GROUP

```
// create two separate groups
  group clients: first_name$, last_name$, address1$, address2$, city$, state$, zip
  group products: sku$, title$, description$, weight, price, shipping$

// create a meta group with both groups
  meta group all: clients, products
```

PRINT GROUP STATEMENT

Format:

```
PRINT GROUP group_name
```

The PRINT GROUP statement is mainly used for debugging purposes. It provides a quick way to display groups of related variables and their values.

The PRINT GROUP statement used with the name of a META GROUP will print the values of *all the variables* in *all the groups* defined within the meta group. PRINT GROUP with the name of an individual GROUP will print the values of all variables within that group only.

Example: PRINT GROUP Statements

```
group identity: id$, name$
group history: last_date$, purchase_type$
meta group customers: history, identity  // put both groups into one meta group

name$ = 'Paul'                          // put data in group 'identity'
id$ = uuid$

last_date$ = date$                      // put data in group 'history'
purchase_type$ = 'credit'

print group customers            // print meta group – the data for all groups – with this one statement
print group identity             // print only the data for one group
end

--- Group HISTORY ---
LAST_DATE$ = "20200611" (8)
PURCHASE_TYPE$ = "credit" (6)

--- Group IDENTITY ---
ID$ = "siDWFPFaAkaVEniLZNqROg" (22)
NAME$ = "Paul" (4)

--- Group IDENTITY ---
ID$ = "siDWFPFaAkaVEniLZNqROg" (22)
NAME$ = "Paul" (4)
```

RESET GROUP Statement

Format:

```
RESET GROUP group_name
```

Often, when starting a new transaction there are several variables that need to be reset (cleared, re-initialized). For example, when a user submits a form in a browser – the data stored in the variables then need to be reset for the next user.

In order to make it easier to reset these variables, you can specify that each variable belongs to one or more groups. This provides the ability to quickly reset the values of all variables within the named group by using a single RESET GROUP statement.

The RESET GROUP statement used with the name of a META GROUP will reset the values of *all the variables* in *all the groups* defined within the meta group. RESET GROUP used with the name of an individual group will reset only the value of the variables within that group.

Example: RESET GROUP Statement

```
group identity: id$, name$
group history: last_date$, purchase_type$
meta group customers: history, identity  // put both groups into one meta group

name$ = 'Paul'                           // put data in group 'identity'
id$ = uuid$

last_date$ = date$                       // put data in group 'history'
purchase_type$ = 'credit'

print group customers            // print meta group – the data for all groups – with this one statement
reset group customers            // reset meta group – the data in all groups – with this one statement
print group customers            // show meta group "customers" has been reset
end

--- Group HISTORY ---
LAST_DATE$ = "20200611" (8)
PURCHASE_TYPE$ = "credit" (6)

--- Group IDENTITY ---
ID$ = "tNv7gSN3GEqEJQthM5cFpQ" (22)
NAME$ = "Paul" (4)

--- Group HISTORY ---
LAST_DATE$ = "" (0)
PURCHASE_TYPE$ = "" (0)

--- Group IDENTITY ---
ID$ = "" (0)
NAME$ = "" (0)
```

MODULE | END MODULE Statements

Format:

```
MODULE module_name
   ...
   ...  [variables, routines]
   ...
END MODULE
```

Modules allow you to build libraries of routines and variables without being concerned with name conflicts.

Modules are generally placed in Sheerpower include (.SPINC) files. Use the MODULE | END MODULE statements at the start and end of the file to define the module.

Note: Modules cannot be nested.

You must use the %INCLUDE directive at the top of your main program to include the module as a source file into the current Sheerpower program before the program can access it. Use CONDITIONAL if you want to conditionally include modules without generating an error if the files do not exist.

```
%include ‘file_spec’
%include conditional: 'file_spec'
```

Example: %INCLUDE MODULE

```
  %include conditional: 'C:\Sheerpower\Samples\client_module.spinc' // no error returned if file not found
```

To access a routine or variable within a module from the main program, the format is:

```
module_name.object
```

where “module_name” is the name of the module, and “object” is a routine or variable within the module. There is a DOT (.) between the module name and the object name.

Example: Access Routine or Variable in a Module

```
// access the “get_client” routine from module “client_maintenance” from within the main program
  client_maintenance.get_client

// access the variable “date_added$” from module “client_maintenance” from within the main program
  print client_maintenance.date_added$
```

Variables and routines in the main program can be accessed from within a module using the format:

```
main_program_name.object
```

where “object” is the name of the variable or routine in the main program. For example, to access a routine called “update_transaction” in a main program named “abc_manufacturing_tracker,” you would use this line of code in your module routine:

Example: Access Routine or Variable in a Module

```
// access the “update_client” routine from the main program “abc_processing” from within a module
  abc_processing.update_client
```

LOGICALS, SYMBOLS & GLOBALS: Interprocess Communication Methods

Sheerpower provides three major methods for interprocess communication:

Logicals	Survive a system reboot.
Symbols	Reset after a system reboot.
Globals	Reset when all Sheerpower programs have been terminated.

1. Logicals

Format:

```
SET | ASK SYSTEM, LOGICAL str_expr: VALUE str_expr
```

Logicals are available system wide. Logicals use registry keys but persist through a system reboot. In addition, logicals can be used in a file specification just like a drive letter can be used, or to store data such as configuration values for a configuration program that asks for the company name.

Example: SET | ASK SYSTEM with LOGICAL

```
set system, logical ‘company’: value ‘Touch Technologies, Inc.’
ask system, logical ‘company’: value the_company$

print the_company$

set system, logical 'Sheerpower': value 'c:\sheerpower'
ask system, logical 'Sheerpower': value z$

print 'Logical set to '; z$
end

Touch Technologies, Inc.
Logical set to c:\sheerpower
```

2. SYMBOLS

Format:

```
SET | ASK SYSTEM, SYMBOL str_expr: VALUE str_expr
```

Symbols are available system wide to other Sheerpower processes. They persist even if all Sheerpower programs have been closed. They use Windows Registry Keys, so can also be read and written to by non-Sheerpower programs. Symbols do not survive a system reboot. Since symbols do not persist, you can ask for the value of a symbol. If the symbol is blank but it should have had a value, it means the system has rebooted.

Example: SET | ASK SYSTEM with SYMBOL

```
// this sample just shows the syntax

  ask system, symbol 'uptime': value start_time$
  set system, symbol 'uptime': value fulltime$
```

3. GLOBALS

Format:

```
GLOBAL str_expr: VALUE str_expr
```

Globals are available system wide, are extremely fast, and are all deleted when the last Sheerpower program closes. Globals are ideal for one Sheerpower program to rapidly communicate with other Sheerpower programs. Globals are the preferred method of interprocess communication in Sheerpower.

In this example, the first program starts the second one – the second has a loop that, every once in a while, checks the first one for something also sends data to another program.

```
// this sample just shows the syntax

  ask system, global 'uptime': value start_time$
  set system, global 'uptime': value fulltime$
```

In addition, once every fraction of a second, each Sheerpower program writes out two globals that contain program-specific information. The names take the form:

```
label_nnnnnnnn
stack_nnnnnnnn
```

…where "nnnnnnnn" is the PID (Process ID) of the Sheerpower program. The PID is assigned by Windows automatically.

These globals can be used to monitor the progress of any Sheerpower program that is running.

CLUSTERS in Sheerpower

When programming, there is often a need to combine multiple variables into a single named object. Doing so makes it easier to keep track of your variables and adds clarity to your code. In Sheerpower, this object is called a CLUSTER.

Each cluster is given a name and a list of variables associated with that name. You can think of cluster variable names as you would column headings in a spreadsheet. A Sheerpower cluster can be either a *scalar cluster* (one dimensional) containing no rows, or a *cluster array* containing rows and columns like an in-memory spreadsheet. In fact, cluster arrays can easily contain spreadsheet information. Because of this, Sheerpower includes a rich set of features to input spreadsheet files directly into clusters and perform operations on them, including database-like operations of sorting, including, excluding, and searching.

Note: You can have up to 512 clusters in a single Sheerpower program, and up to 256 variables in each cluster.

Creating Sheerpower Clusters

Format:

```
CLUSTER cluster_name: var1[, var2][, var3] … [, var256]
```

To create a cluster in Sheerpower (scalar or array), first the cluster name and associated related variable names are defined using the format above.

Notice that the format for referencing cluster variables is:

```
cluster_name->var_name
```

…with no spaces entered around the "->" symbols. The variable names assigned will become "headers" used when outputting cluster data with the PRINT CLUSTER statement (mainly for debugging purposes).

Creating Multiple Related Clusters

Format:

```
CLUSTER cluster_name USING root_cluster_name
```

If you have multiple clusters all based on the same "root" cluster, you can define the root cluster once and then reference it when defining related clusters. In this example, the MEALS cluster first defined and then used to further define the related breakfast, lunch, and dinner clusters. The MEALS cluster becomes the root cluster.

Example: Creating Multiple Related Clusters

```
cluster meals: protein$, liquid$, carb$
cluster breakfast using meals
cluster lunch using meals
cluster dinner using meals

breakfast->protein$ = “eggs”
breakfast->liquid$ = “tea”
breakfast->carb$ = “toast”

lunch->protein$ = “chicken”
lunch->liquid$ = “coffee”
lunch->carb$ = “rice”

dinner->protein$ = “steak”
dinner->liquid$ = “wine”
dinner->carb$ = “potatoes”
```

ADD CLUSTER Statement

There are three ways to add rows to a cluster that contains multiple rows of data. **The first (and less common method)** is to explicitly specify each row to be populated using SET CLUSTER to set the new row to be added as current. Then, the new data can be populated into the row. See SET CLUSTER Statement for more details.

Format:

```
SET CLUSTER cluster_name: ROW int_expr
```

If the row number specified exceeds the current number of rows in the cluster, the cluster automatically expands to accommodate it.

The next example shows how to use SET CLUSTER to add data into a cluster to the current row.

Example: Use SET CLUSTER to Add Data in Explicit Row Number

```
cluster payroll: tax, gross
print "Payroll cluster has no rows of data added yet: "; size(payroll)

set cluster payroll: row 2  // make the current row be “2”
payroll->tax = 42.57
payroll->gross = 200
print "Cluster now has: "; size(payroll); " rows."

set cluster payroll: row 15    // make the current row be “15”
payroll->tax = 52
payroll->gross = 230
print "Cluster now has: "; size(payroll); " rows."
end
```

```
//results

Payroll cluster has no rows of data added yet:  0
Cluster now has:  2  rows.
Cluster now has:  15  rows.
```

The **second and *preferred method*** to populate a cluster is to use the ADD CLUSTER statement to add the new row to the end of the cluster.

Format:

```
ADD CLUSTER cluster_name[: ROW int_expr]
```

In the next example, a new cluster array is created with the name “student” in the first line. Each row of the cluster will contain information on a different student: name, age, and grade level. A total of three students are added, making the total size of the cluster “3.”

Example: ADD CLUSTER Statement

```
 cluster student: name$, age

 add cluster student
 student->name$ = “Joan Ark”
 student->age = 18
 student->level = 12

 add cluster student
 student->name$ = “John Smith”
 student->age = 16
 student->level = 10

 add cluster student
 student->name$ = “Desmond Jones”
 student->age = 15
 student->level = 10

 print size(student)
 end

3
```

Note: Unlike the ADD TABLE statement, ADD CLUSTER ***does not*** have a corresponding END ADD statement.

The ADD CLUSTER statement establishes a new row. In the last example, since the cluster was newly created, the first ADD CLUSTER statement created row 1.

```
 add cluster student
```

The next lines of code below the first ADD CLUSTER statement stored information into each variable of row 1, just like storing data into the columns of a row in a spreadsheet.

```
student->name$ = “Joan Ark”
student->age = 18
student->level = 12
```

	Column #1	Column #2	Column #3
Headers →	**Name**	**Age**	**Level**
Row #1:	Joan Ark	18	12

The next two ADD CLUSTER statements created and added data into rows 2 and 3.

```
add cluster student
student->name$ = “John Smith”
student->age = 16
student->level = 10

add cluster student
student->name$ = “Desmond Jones”
student->age = 15
student->level = 10
```

	Column #1	Column #2	Column #3
Headers →	**Name**	**Age**	**Level**
Row #1:	Joan Ark	18	12
Row #2:	John Smith	16	10
Row #3:	Desmond Jones	15	10

Note: If you first create a *scalar cluster* with data stored into the variables and then transform the cluster into a *cluster array* by adding rows to it, the initial data that was stored into the scalar cluster variables will be deleted.

And the **third method** to populate a cluster is with a shortcut of ADD CLUSTER when adding rows to a cluster that contains ONLY constant data.

Example: ADD CLUSTER Statement with Constant Data SHORTCUT

```
cluster people: id$, firstname$, lastname$
add cluster people: id$='one', firstname$='Fred', lastname$='Smith'
add cluster people: id$='two', firstname$='Sally', lastname$='Sue'
```

SET CLUSTER Statement

Format:

```
SET CLUSTER cluster_name: ROW num_expr
```

The SET CLUSTER statement is used to make a row CURRENT. After adding a new row, that row is said to be *current*. To access the information from a different row, it must first be made current, as shown in the next example.

Example: SET CLUSTER Statement

```
cluster student: name$, age, level

add cluster student
student->name$ = “Shirley Rogers”
student->age = 12
student->level = 6

add cluster student
student->name$ = “John Smith”
student->age = 13
student->level = 6

add cluster student
student->name$ = “Andrea Johnson”
student->age = 13
student->level = 7

print ‘Third row is current: ‘; student->name$
print

set cluster student: row 1
print ‘First row is current: ‘; student->name$
end
```

```
Third row is current: Andrea Johnson

First row is current: Shirley Rogers
```

ASK CLUSTER Statement

Format:

```
ASK CLUSTER cluster_name: row num_var
```

The ASK CLUSTER statement is used to find out which row is current.

Example: ASK CLUSTER Statement

```
cluster client: ssn$, id$, weight, age%

for i=1 to 5
  set cluster client: row i
  client->ssn$ = '12-34-56-' + str$(i)
  client->id$ = str$(i)
  client->weight = 1.01*i
  client->age% = 101*i
next i

set cluster client: row 3
ask cluster client: row x
print x
end

3
```

SIZE() Function

Format:

```
SIZE(cluster_name)
```

The SIZE() function is used to find out how many rows are in a cluster.

Example: SIZE() Function

```
cluster client: ssn$, id$, weight, age%

for I = 1 to 5
  set cluster client: row i
  client->ssn$ = '12-34-56-' + str$(i)
  client->id$ = str$(i)
  client->weight = 1.01*i
  client->age% = 101*i
next i

print size(client)
end

5
```

COLLECT CLUSTER | END COLLECT Statements

Format:

```
COLLECT CLUSTER cluster_name
  …
  …   [block of code]
  …
END COLLECT
```

When working with cluster arrays, rows are typically operated on one at a time. The COLLECT CLUSTER | END COLLECT statements are used to collect each row.

Note: In Sheerpower, "COLLECT" and "_COLLECTED" are the equivalent of "EXTRACT" and "_EXTRACTED" and can be used interchangeably. See Full List of Synonyms in Sheerpower.

Example: COLLECT CLUSTER | END COLLECT Statements

```
cluster student: name$, age, level

add cluster student
student->name$ = "Joan Ark"
student->age = 18
student->level = 12

add cluster student
student->name$ = "John Smith"
student->age = 16
student->level = 10

add cluster student
student->name$ = "Desmond Jones"
student->age = 15
student->level = 10

ages = 0
counter = 0
collect cluster student
  print student->name$, student->age, student->level
  ages = ages + student->age
  counter++
end collect

print 'The average age is '; ages/counter
end
```

```
Joan Ark                18      12
John Smith              16      10
Desmond Jones           15      10
The average age is 16.3333333333333333
```

COLLECT | END COLLECT iterates through each row of a cluster. While doing so, it creates a COLLECTION of rows. A collection can be a subset of the entire cluster array and can be sorted by various criteria (see INCLUDE, EXCLUDE and SORT BY Statements for more on sorting).

To iterate through a collection, use the FOR | NEXT loop statement.

The next example sorts the students by name and then prints out the sorted list.

Example: FOR | NEXT Statement to Iterate Through a Collection

```
cluster student: name$, age, level

add cluster student
student->name$ = "Joan Ark"
student->age = 18
student->level = 12

add cluster student
student->name$ = "John Smith"
student->age = 16
student->level = 10

add cluster student
student->name$ = "Desmond Jones"
student->age = 15
student->level = 10

collect cluster student
  sort by student->name$
end collect

for each student
  print student->name$
next student
end
```

```
Desmond Jones
Joan Ark
John Smith
```

UNIQUE Option with COLLECT | END COLLECT Statements

```
COLLECT CLUSTER cluster_name[: UNIQUE cluster_name->var_name]
  …
  …    [block of code]
  …
END COLLECT
```

The COLLECT CLUSTER statement can be used with the UNIQUE option to find each *unique occurrence* of a given cluster field.

The next example shows how to create a collection of unique cluster field occurrences – collecting the count of unique grade levels using _COLLECTED, and then printing out each level with the corresponding count of students.

Example: COLLECT | END COLLECT with UNIQUE Option

```
cluster student: name$, age, level

add cluster student
student->name$ = "Joan Ark"
student->age = 18
student->level = 12

add cluster student
student->name$ = "John Smith"
student->age = 16
student->level = 10

add cluster student
student->name$ = "Desmond Jones"
student->age = 15
student->level = 10

collect cluster student: unique student->level
end collect

count = _collected
for each student
  count = _collected // store the count of students in the current level
  print student->level, count
next student
end

12          1
10          2
```

INCLUDE, EXCLUDE and SORT BY with COLLECT | END COLLECT Statements

Format:

```
INCLUDE | EXCLUDE cluster_name->logical_expr
SORT BY cluster_name->var_name
```

INCLUDE and EXCLUDE statements can be used include or exclude specific rows in a cluster when creating a collection. The SORT BY statement is used to sort a collection. Any number of INCLUDE, EXCLUDE, or SORT BY statements can be used on a cluster array.

The next example shows how to create a collection that excludes students whose age is greater than 18.

Example: INCLUDE, EXCLUDE, and SORT BY Statements

```
cluster student: name$, age, level

add cluster student
student->name$ = “Joan Ark”
student->age = 18
student->level = 12

add cluster student
student->name$ = “John Smith”
student->age = 16
student->level = 10

add cluster student
student->name$ = “Desmond Jones”
student->age = 15
student->level = 10

collect cluster student
   exclude student->age > 16  // do not collect records where the age is greater than 16
   sort by student->name$     // sort the records alphabetically by name
end collect

for each student
  print student->name$
next student
end
```

```
Desmond Jones
John Smith
```

CLUSTER INPUT Statement

Format:

```
CLUSTER INPUT NAME file_spec[, DATA str_expr][, #chnl_expr]
        [, HEADERS int_expr][, INCLUDE str_expr][, EXCLUDE str_expr]
        [, TAB][, RECORD str_delim][, FIELD str_delim]: cluster_name
```

The CLUSTER INPUT statement is used for inputting data directly into a cluster array, such as large spreadsheets in the .CSV format. By default, CLUSTER INPUT uses a comma delimiter to input comma separated files (e.g., .CSV, .TXT, etc.)

The options available for use with CLUSTER INPUT are listed below.

Parameter	**Description**
DATA	Used to specify rows containing specific values to input into the cluster.
#CHNL_EXPR	Specify an open channel to input the data from.
HEADERS	Used to skip header rows (with a numeric value to indicate the number of rows to skip).
INCLUDE | EXCLUDE	Used to include and exclude specific columns by column ID letters or numbers in a comma separated list. Supports numeric or alphanumeric column IDs (e.g., “1,3,5” and “a,b,ac”) and column ranges (e.g., “a-ab”) and combinations of both (e.g., “1,3,5-10”).
TAB	Changes the default delimiter to be a TAB for tab delimited data files (e.g., TSV)
RECORD, FIELD	Used to specify custom record and field delimiters.

To input the data from a .CSV spreadsheet, first define the cluster and cluster variables, then use CLUSTER INPUT to input the spreadsheet data into the cluster while skipping any header rows using the HEADERS option (“HEADER” will work too).

```
cluster cities: city$, country$, population, region$, lat$, lng$
cluster input name ‘@world_cities.csv’, headers 1: cities  // skips inputting the single header line at the top
```

CLUSTER INPUT does the following:

- opens the file
- reads a line of data from the file
- breaks the line up into fields using the delimiter, honoring quoted data
- makes a new cluster array row at the end of the array
- stores each column (field) into the corresponding cluster variable
- reports any data conversion errors if a non-numeric is stored into a numeric variable
- ignores extra fields if there are more fields than cluster variables and ignores extra variables
- continues until the entire file has been read
- closes the file

If the spreadsheet is TAB delimited (.TSV) use the TAB option as shown below..

Example: CLUSTER INPUT with TAB

```
cluster input name 'cities_file.tsv', tab: cities
```

If the file has a *custom* field delimiter, use the FIELD option, and specify the delimiter as a string expression.

Example: CLUSTER INPUT with FIELD Option for Custom Delimiter

```
cluster input name 'cities_file.txt', field '~': cities
```

Sheerpower can also input data from files that have special record and field delimiters, such as EDI (Electronic Data Interchange) files, by using the RECORD and FIELD options.

For example, an EDI X12 file uses a tilde (~) to end each record, and an asterisk (*) for each field. To input an EDI file into a cluster:

Example: CLUSTER INPUT with RECORD and FIELD Options for Delimiters

```
cluster input name 'some_edi_file.x12', record '~', field '*': cities
```

There are situations where you only want to work with specific rows of data in a spreadsheet. In these cases, use the DATA option with CLUSTER INPUT to input one row at a time, examine the data, and then conditionally input the data into a cluster array.

Example: CLUSTER INPUT with DATA Option (Input Specific Rows)

```
cluster cities: city$, country$, population, region$, lat$, lng$
open file cities_ch: name '@world_cities.csv'
my_row$ = 'San Diego, US, 3, 32.715, -117.161'
cluster input data my_row$, headers 1: cities
print cluster cities

"San Diego"," US",3," 32.715"," -117.161",""
```

Sheerpower can also read data directly into a cluster from an open channel.

Example: CLUSTER INPUT with Open Channel

```
cluster cities: city$, country$, population, region$, lat$, lng$
open file cities_ch: name '@world_cities.csv'
cluster input #cities_ch: cities
close #cities_ch
print cluster cities: all

CITY,COUNTRY,POPULATION,REGION,LAT,LNG
"Sandwich","US",34886,"MA","41.7588889","-70.4944444"
"Minot","US",34885,"ND","48.2325000","-101.2958333"
"Leavenworth","US",34880,"KS","39.3111111","-94.9222222"
"Azogues","EC",34877,"04","-2.7333333","-78.8333333"
"Alpharetta","US",34869,"GA","34.0752778","-84.2941667"
"Cumberland","US",34843,"RI","41.9666667","-71.4333333"
```

Sometimes you only want to input selected columns from the spreadsheet. This can be done using either the INCLUDE or EXCLUDE options. Specifying only the columns needed can significantly speed up the input of very large files.

Example: CLUSTER INPUT with INCLUDE and EXCLUDE

```
cluster cities: city$, population
open file cities_ch: name '@world_cities.csv'
cluster input name '@world_cities.csv', include 'a,c': cities  // only input columns "a" and "c"
print cluster cities, header 'CITY,POPULATION':all
end

CITY,POPULATION
"Dzierzoniow",34888
"Sandwich",34886
"Minot",34885
"Leavenworth",34880
"Azogues",34877
```

INCLUDE and EXCLUDE can take either individual column ID names "a,b,c,aa,bz" or ranges "a-c,aa,bz." Column numbers are also supported "1-3,27."

```
cluster input name '@world_cities.csv', exclude 'b-d': cities  // input all columns except "b" through "d"
```

PRINT CLUSTER Statement

Format:

```
PRINT CLUSTER cluster_name[, #chnl_expr][, HEADER str_expr]
                              [, INCLUDE str_expr][, EXLUDE str_expr][, TAB]
                              [, RECORD str_delim] [, FIELD str_delim][, LIST][, UNQUOTED][:ALL | ROW int_expr]
```

The PRINT CLUSTER statement is used to output the contents of a cluster to a file. By default, PRINT cluster outputs the value of each variable in the current row as a comma delimited list. If the variable is a string variable, quotes are placed around the data.

The options available for use with PRINT CLUSTER are listed below.

Parameter	Description
#CHNL_EXPR	Specifies the channel number to output the print data to.
INCLUDE | EXCLUDE	Limit which variables to output given their relative column positions specified in a comma separated list. Supports numeric or alphanumeric column IDs (e.g., "1,3,5" and "a,b,ac") and column ranges (e.g., "a-ab") and combinations of both (e.g., "1,3,5-10").
HEADER | HEADERS	Used to specify a custom header row when used with the ALL option. A null header string suppresses outputting any header.

LIST	Outputs a vertical list of variable names and their values. This is useful when debugging to easily see each variable and its value.
UNQUOTED	Suppresses placing quotes around string data. Some types of delimited files do not support quoted data (e.g., EDI files).
TAB | RECORD | FIELD	Used to specify record and field (column) delimiters. The default delimiter is a “new line” for records and a comma for fields.
ALL	Prints all of the variables from all cluster rows with a default header generated from the cluster variable names.
ROW	Prints the variables and their values of the row specified in int_expr.

Example: PRINT CLUSTER – Default

```
cluster client: ssn$, id$
for i=1 to 5
  set cluster client: row i
  client->ssn$='12-34-56-' + str$(i)
  client->id$ = str$(i)
next i

set cluster client: row 1
print cluster client

"12-34-56-1","1"
```

To print all rows in a cluster, use the ALL option. By default, the headers (consisting of the variable names) will be printed as well.

Example: PRINT CLUSTER: ALL with Default Header

```
cluster client: ssn$, id$, weight, age%
for i=1 to 5
 set cluster client: row i
 client->ssn$='12-34-56-' + str$(i)
 client->id$ = str$(i)
 client->weight=1.01*i
 client->age% = 101*i
next i
print cluster client: all

SSN,ID,WEIGHT,AGE
"12-34-56-1","1",1.01,101
"12-34-56-2","2",2.02,202
"12-34-56-3","3",3.03,303
"12-34-56-4","4",4.04,404
"12-34-56-5","5",5.05,505
```

To print all rows in a cluster with custom headers, use the HEADERS option with a list of headers defined in a string.

Note: You can use "HEADERS" or "HEADER" – both will work (Sheerpower synonyms).

Example: PRINT CLUSTER: ALL with Custom Headers

```
 cluster client: ssn$, id$, weight, age%
 for i=1 to 5
  set cluster client: row i
  client->ssn$='12-34-56-' + str$(i)
  client->id$ = str$(i)
  client->weight=1.01*i
  client->age% = 101*i
 next i

 print cluster client, headers 'COLUMN 1,COLUMN 2,COLUMN 3,COLUMN 4': all
 end

COLUMN 1,COLUMN 2,COLUMN 3,COLUMN 4
"12-34-56-1","1",1.01,101
"12-34-56-2","2",2.02,202
"12-34-56-3","3",3.03,303
"12-34-56-4","4",4.04,404
"12-34-56-5","5",5.05,505
```

To print all rows in a cluster without the headers, use the HEADERS option and define a null string.

Example: PRINT CLUSTER: Headers Suppressed

```
 cluster client: ssn$, id$, weight, age%
 for i=1 to 5
  set cluster client: row i
  client->ssn$='12-34-56-' + str$(i)
  client->id$ = str$(i)
  client->weight=1.01*i
  client->age% = 101*i
 next i

 print cluster client, headers " ": all
 end

"12-34-56-1","1",1.01,101
"12-34-56-2","2",2.02,202
"12-34-56-3","3",3.03,303
"12-34-56-4","4",4.04,404
"12-34-56-5","5",5.05,505
```

To print a list of the cluster variables and their values for the *current* row, use the LIST option. Use the ALL option to print a list of each cluster variable and values.

Example: PRINT CLUSTER: LIST

```
cluster client: ssn$, id$, weight, age%

for i=1 to 5
 set cluster client: row i
 client->ssn$='12-34-56-' + str$(i)
 client->id$ = str$(i)
 client->weight=1.01*i
 client->age% = 101*i
next i

print cluster client, list
end

--- Row 5 ---
CLIENT->SSN$ = "12-34-56-5" (10)
CLIENT->ID$ = "5" (1)
CLIENT->WEIGHT = 5.05
CLIENT->AGE% = 505
```

To print a specific row in the cluster, use the ROW option.

Example: PRINT CLUSTER: ROW

```
cluster client: ssn$, id$, weight, age%

for i=1 to 10
 set cluster client: row i
 client->ssn$='12-34-56-' + str$(i)
 client->id$ = str$(i)
 client->weight=1.01*i
 client->age% = 101*i
next i

print cluster client: row 5
end

"12-34-56-5","5",5.05,505
```

To output the print results to a channel, specify an open channel as shown in the next example.

Example: PRINT CLUSTER to an Open Channel

```
cluster client: ssn$, id$, weight, age%
for i=1 to 5
 set cluster client: row i
 client->ssn$='12-34-56-' + str$(i)
 client->id$ = str$(i)
 client->weight=1.01*i
 client->age% = 101*i
next i

open file print_ch: name '@cluster_print.txt', access update  // print results written to cluster_print.txt
print cluster client, #print_ch: all
close #print_ch
```

To include or exclude specific columns to print, use the INCLUDE or EXCLUDE options. INCLUDE and EXCLUDE can take a list of individual column numbers “1,3,5” or ranges “1-3,7-9” or a combination of both ranges and individual column numbers “1-3,27.” Column ID names are also supported, such as “a,b,d-r,aa,bd-bf,ga.”

Example: PRINT CLUSTER with INCLUDE Option

```
cluster client: ssn$, id$, weight, age%
for i=1 to 5
 set cluster client: row i
 client->ssn$='12-34-56-' + str$(i)
 client->id$ = str$(i)
 client->weight=1.01*i
 client->age% = 101*i
next i
print cluster client, include '2-4',headers 'ID,WEIGHT,AGE'
end

ID,WEIGHT,AGE
"5",5.05,505
```

Example: PRINT CLUSTER with EXCLUDE Option

```
cluster client: ssn$, id$, weight, age%
for i=1 to 5
 set cluster client: row i
 client->ssn$='12-34-56-' + str$(i)
 client->id$ = str$(i)
 client->weight=1.01*i
 client->age% = 101*i
next i
print cluster client, exclude '3'

"12-34-56-5","5",505
```

To print different record or field delimiters, use the RECORD and FIELD options with the delimiter specified in a string expression.

Example: PRINT CLUSTER with RECORD and FIELD Options

```
cluster student: name$, age, level

add cluster student
student->name$ = “Joan Ark”
student->age = 18
student->level = 12

add cluster student
student->name$ = “John Smith”
student->age = 16
student->level = 10

print cluster student, record '~', field '*’: all
end

NAME*AGE*LEVEL~
"Joan Ark"*18*12~
"John Smith"*16*10~
```

FINDROW() Function

Format:

```
FINDROW(cluster_name->var_name, str_expr[, int_expr1][, int_expr2])
```

The FINDROW() function is used to search a cluster array for information and supports all data types. The FINDROW() function is highly optimized. It can perform over 10 million searches per second if the data is found, and 15 million per second if the data is not found. This makes FINDROW() ideal for tasks that require fast lookups. If the search is successful, the “found” cluster array row is now current.

Given the cluster name, variable to search in, and data to be searched for, FINDROW() returns either the first row where the data was found or returns a “0” if the data was not found.

FINDROW() has an optional third parameter to find the Nth occurrence of a value defined in int_expr1. FINDROW() locates the Nth occurrence of any given value at a rate of over 10 million searches per second.

Format Option	Description
0	The default if left undefined in int_expr2; perform case-regardless searches.
1	Perform case-sensitive searches. When using this parameter, be sure to also include the third parameter int_expr1 to define the Nth occurrence to find (e.g., “1” to find the first occurrence by default).

Note: After setting the format option to “1” to perform case-sensitive searches with FINDROW(), the setting remains “sticky” until the program ends or the default format option of “0” is set.

Example: FINDROW() Function

```
cluster student: name$, age, level

add cluster student
student->name$ = “Joan Ark”
student->age = 18
student->level = 12

add cluster student
student->name$ = “John Smith”
student->age = 16
student->level = 10

add cluster student
student->name$ = “Desmond Jones”
student->age = 15
student->level = 10

add cluster student
student->name$ = “joan ark”
student->age = 18
student->level = 12

print findrow(student->name$, “John Smith”, 1, 1)      // find the first instance of John Smith, case sensitive
print findrow(student->age, 14)                        // none will be found
print findrow(student->name$, "joan ark", 1)           // first instance, still case sensitive (sticky setting)
print findrow(student->name$, "joan ark", 1, 0)        // first instance, now case regardless
end

2
0
4
1
```

Directly after calling the FINDROW() function, the variable _COLLECTED contains the number of occurrences of whatever was found.

The example on the next page shows how to use FINDROW() to find specific instances of data in a cluster, print out how many instances were found using _COLLECTED, and then print out the variable values for each row found in a FOR NEXT loop statement.

Example: FINDROW() with _COLLECTED

```
cluster student: name$, city$

add cluster student
student->name$ = "Joan Ark"
student->city$ = "New York City"

add cluster student
student->name$ = "Jason Nordahl"
student->city$ = "Helena"

add cluster student
student->name$ = "Frank Abbott"
student->city$ = "San Diego"

add cluster student
student->name$ = "Sarah Walters"
student->city$ = "San Diego"

row = findrow(student->city$, 'San Diego')
print ‘Number of students from San Diego: ‘; _collected

for index = 1 to _collected
  row = findrow(student->city$, “San Diego”, index)
  print student->city$, student->name$
next index
end
```

```
2
```

FINDROW() with Synthetic Keys

A *synthetic key* is the result of creating a data field using other data fields. In Sheerpower, synthetic keys are used to search a cluster using multiple variables. This method is significantly more efficient than sequentially searching by individual variables (typically a million times faster for large clusters).

For example, you want to perform lookups in a cluster given a last name and a birthdate. However, there is no variable in the cluster that contains both the last name and birthdate. The solution is to create a synthetic key. The basic steps are listed below.

1) Add new variable to the cluster. This will be the synthetic key.
2) Populate the new variable with the key lookup information.
3) Use FINDROW() to do the key lookups.

Example: FINDROW() Lookups with Synthetic Keys

```
// Do lookups using a synthetic key
cluster student: first_name$, last_name$, lookup_key$

// Add a few students
add cluster student: first_name$='Fred',  last_name$ = 'Smith'
add cluster student: first_name$='Sally', last_name$ = 'Sue'

// Set up the synthetic key for lookups
collect cluster student
  student->lookup_key$ = left(student->first_name$, 1) + student->last_name$
end collect

do
  line input 'First initial of the first name and last name': mykey$
  if mykey$ = '' then exit do

  row = findrow(student->lookup_key$, mykey$)
  if row = 0 then
    print '?? Could not find: '; mykey$
    repeat do
  end if

  print 'Found: '; student->first_name$;' '; student->last_name$
loop
end

First initial of the first name and last name? fsmith  // ← type in "fsmith"
Found: Fred Smith
First initial of the first name and last name? ssue
Found: Sally Sue
First initial of the first name and last name? ajones
?? Could not find: ajones
First initial of the first name and last name? exit
```

JSON$() Function

Format:

```
str = JSON$(cluster_name[,num_expr])
```

Given a cluster and optional row number, the JSON$() function generates the appropriate JSON string. To create JSON objects using clusters, see Example: Embedded Cluster Template JSON Objects Using PREFIX and Clusters with Adhoc JSON Objects.

Parameter	Description
0	Default – JSON is generated for the current row.
-1	JSON is generated for all rows as a JSON array.
Specified row > 0	JSON is generated for the specified row.

Example: JSON$() Function

```
cluster name: first$, last$
cluster multi: ssn$, prefix cluster name, tax, is_okay?
add cluster multi
multi->ssn$     = '111-22-3333'
multi->name->first$ = 'Mister'
multi->name->last$  = 'Smith'
multi->tax      = 45.67
multi->is_okay? = true

add cluster multi
multi->ssn$     = '222-33-4444'
multi->name->first$ = 'Mister2 "they say"'
multi->name->last$  = 'Smith2'
multi->tax      =  45.678
multi->is_okay? = false

print 'Current row json'
x$ =  json$(multi)
print x$
print

print 'Row one json'
x$ =  json$(multi, 1)
print x$
print

print 'All rows json'
x$ =  json$(multi, -1)
print x$
end
```

Current row json

```
{
  "multi":
    {
    "ssn":"222-33-4444",
    "name": {
      "first":"Mister2 \"they say\"",
      "last":"Smith2"
      },
    "tax":45.678,
    "is_okay":false
    }
}
```

Row one json

```
{
  "multi":
    {
    "ssn":"111-22-3333",
    "name": {
      "first":"Mister",
      "last":"Smith"
      },
    "tax":45.67,
    "is_okay":true
    }
}
```

All rows json

```
{
  "multi":[
    {
    "ssn":"111-22-3333",
    "name": {
      "first":"Mister",
      "last":"Smith"
      },
    "tax":45.67,
    "is_okay":true
    },
    {
    "ssn":"222-33-4444",
    "name": {
      "first":"Mister2 \"they say\"",
      "last":"Smith2"
      },
    "tax":45.678,
    "is_okay":false
    }
  ]
}
```

COPY CLUSTER Statement

Format:

```
COPY CLUSTER cluster_name TO cluster_name [: ALL | APPEND]
```

The COPY CLUSTER statement is used to copy data from one cluster to another. By default, COPY CLUSTER copies only the current record from the source to the destination.

Note: The source and destination clusters must have the same cluster "root" (see Creating Multiple Related Clusters).

The ALL and APPEND parameters are available to use with the COPY CLUSTER statement.

Parameter	Description
ALL	Copies all rows starting with row one of the cluster source to the corresponding destination cluster array row.
APPEND	Copies all rows starting with row one of the source cluster, appending each row to the end of the destination cluster.

Example: COPY CLUSTER Statement

```
cluster student: name$, age, level

add cluster student
student->name$ = "Joan Ark"
student->age = 18
student->level = 12

 add cluster student     // this record will be "current"
student->name$ = "Desmond Jones"
student->age = 15
student->level = 10

cluster new_student using student            // creates a new cluster using the source as a template
copy cluster student to new_student          // copy only the "current" record from student to new_student
print cluster new_student, headers '': all   // print all the records with headers suppressed

"Desmond Jones",15,10
```

Example: COPY CLUSTER Statement with ALL Option

```
cluster student: name$, age, level

add cluster student
student->name$ = "Joan Ark"
student->age = 18
student->level = 12

add cluster student
student->name$ = "John Smith"
student->age = 16
student->level = 10

cluster new_student using student            // creates a new cluster using the source as a template
copy cluster student to new_student: all     // copy all the records
print cluster new_student: all               // print all the records with headers

NAME,AGE,LEVEL
"Joan Ark",18,12
"John Smith",16,10
```

See the next page for the example using COPY CLUSTER with the APPEND option.

Example: COPY CLUSTER Statement with APPEND Option

```
cluster student: name$, age, level

add cluster student
student->name$ = "Joan Ark"
student->age = 18
student->level = 12

add cluster student
student->name$ = "John Smith"
student->age = 16
student->level = 10

add cluster student
student->name$ = "Desmond Jones"
student->age = 15
student->level = 10

cluster new_student using student    // creates a new cluster using the source as a template

add cluster new_student   // add one record
new_student->name$ = "Eric James"
new_student->age = 19
new_student->level = 12

print 'New cluster with one record:'
print cluster new_student: all
print

copy cluster student to new_student: append

print 'New cluster with 3 records appended to the 1st with headers suppressed:'
print cluster new_student, header ' ': all
end
```

```
New cluster with one record:
NAME,AGE,LEVEL
"Eric James",19,12

New cluster with 3 records appended to the 1st with headers suppressed:
"Eric James",19,12
"Joan Ark",18,12
"John Smith",16,10
"Desmond Jones",15,10
```

RESET CLUSTER Statement

Format:

```
RESET CLUSTER cluster_name[: ALL]
```

The RESET CLUSTER statement is used to reset either the value of the current cluster row or the entire cluster to nulls and zeros. This statement works for scalar clusters as well.

By default, RESET CLUSTER resets all values in the *current row* to nulls and zeros.

Example: RESET CLUSTER Statement – Current Row

```
cluster client: ssn$, id$
for i = 1 to 5
  set cluster client: row i
  client->ssn$ = '12-34-56-' + str$(i)
  client->id$ = str$(i)
next i
ask cluster client: row x
print 'The current row is: ';
print x
print 'The current row contains: ';
print cluster client

reset cluster client
print 'Now the current row contains: ';
print cluster client

The current row is:  5
The current row contains: "12-34-56-5","5"
Now the current row contains: "",""
```

Use the ALL option with RESET CLUSTER to reset the entire cluster array to nulls and zeros.

Example: RESET CLUSTER Statement with ALL Option

```
cluster client: ssn$, id$
for I = 1 to 5
  set cluster client: row i
  client->ssn$ = '12-34-56-' + str$(i)
  client->id$ = str$(i)
next i
print 'The original number of cluster rows is: ';
print size(client)

reset cluster client: all
print 'And now the number of cluster rows is: ';
print size(client)

The number of cluster rows is: 5
And now the number of cluster rows is: 0
```

Scalar Clusters

Format:

```
CLUSTER cluster_name: var1[, var2][, var3] … [, var256]
```

Scalar clusters have no rows. They are typically used to store related information about a single overall concept. An example overall concept is "meals" with the related information being the types of food one eats during each meal: protein, liquid, and carbs. This example shows how to create a scalar cluster and store data into the cluster variables.

Example: Create a Scalar Cluster with Data

```
cluster meals: protein$, liquid$, carb$
meals->protein$ = "eggs"
meals->liquid$ = "tea"
meals->carb$ = "toast"
```

There's a faster method to create the scalar cluster with constant variables:

```
  cluster payroll: ssn$, state$ = 'NH'
  print payroll->state$

NH
```

ENUM Statement

Format:

```
ENUM cluster_name: str1[, str2][, str3] … [, str256]
```

The ENUM (enumerated) statement creates a scalar cluster where each member is assigned a sequential number starting at 1.

Example: ENUM Statement

```
  enum season: spring, summer, fall, winter
  print season->summer
  print enum season
  enum season9 using season
  print enum season9

 2
1,2,3,4
1,2,3,4
```

ENUM is useful when making "state machines" and other situations where assigning sequence numbers to variables is handy.

Embedded Cluster Templates

A "cluster template" is a cluster whose primary use is as a reference and not for data storage itself. There may be standard variables that you want multiple clusters to contain. For example, the standard address format which includes address line 1, address line 2, city, state, zip code. This address format will be used for the client, the user, and the company clusters. A separate cluster for these variables can be set up as a TEMPLATE that can then be embedded into other clusters. There will be no data stored in this cluster template. The example below shows the cluster address used as a template by other clusters.

Example: Embedded Cluster Templates

```
cluster address: addr1$, addr2$, city$, state$, zip$   // cluster template
cluster client: client_id$, client_name$, cluster address
cluster user: login$, name$, cluster address, age
cluster company: company_name$, cluster address
client->city$ = 'San Diego'  // data is then stored into the cluster variables as usual:
```

Cluster templates also can be embedded into other clusters and used as JSON OJBECTS as shown in the example below.

Example: Embedded Cluster Template JSON Objects Using PREFIX

```
 cluster name: first$, last$
 cluster multi: ssn$, prefix cluster name, tax, is_okay?
 add cluster multi
 multi->ssn$     = '222-33-4444'
 multi->name->first$   = 'Mister2 "they say"'
 multi->name->last$    = 'Smith2'
 multi->tax      = 45.678
 multi->is_okay? = false

 print 'Current row json'
 x$ =  json$(multi)
 print x$
 print
end

Current row json
{
  "multi":
    {
    "ssn":"222-33-4444",
    "name": {
      "first":"Mister2 \"they say\"",
      "last":"Smith2"
      },
    "tax":45.678,
    "is_okay":false
    }
}
```

To store data into a cluster variable that is part of a JSON object, the cluster name and the cluster template name must both be specified as shown below.

Example: Store Data in JSON Object Cluster Variable

```
multi->name->first$   = 'Mister'
multi->name->last$    = 'Smith'
```

Clusters with Adhoc JSON Objects

JSON objects can be created without using embedded cluster templates. When defining the cluster, specify the name of the object when describing the variable. The example below will create an object named "address" that describes the address variables to include.

Example: Clusters with Adhoc JSON Objects

```
cluster  student: id$, address->addr1$, address->addr2$, address->city$, address->state$, address->zip$
student->id$ = "John Henry"
student->address->addr1$ = "23 Hummingbird Way"
student->address->addr2$ = "Box 456"
student->address->city$ = "Escondido"
student->address->state$ = "CA"
student->address->zip$ = "92345"

 x$ =  json$(student)
 print x$
 print
 end

{
  "student":
    {
    "id":"John Henry",
    "address": {
      "addr1":"23 Hummingbird Way",
      "addr2":"Box 456",
      "city":"Escondido",
      "state":"CA",
      "zip":"92345"
      }
    }
}
```

Passing Clusters into Routines

Format:

```
routine_name WITH root_cluster_name cluster_name
```

Clusters can be passed into routines using either the cluster’s name or the cluster’s root name. In addition, clusters outside of the routine can be directly referenced from inside of the routine; they are globally available.

The routine is called by providing the routine name, the root cluster name, followed by the name of the cluster being passed into the routine.

From inside of the routine, you can both read and write clusters. This provides an easy method to pass a lot of variables into a routine without having to pass in all the individual variable names.

Example: Passing Clusters into Routines

```
// define root cluster
  cluster meal: type$, protein$, liquid$, carb$

// define related clusters
  cluster breakfast using meal
  cluster lunch using meal

// add data to clusters
  breakfast->type$ = "Breakfast"
  breakfast->protein$ = "eggs"
  breakfast->liquid$ = "tea"
  breakfast->carb$ = "toast"

  lunch->type$ = "Lunch"
  lunch->protein$ = "chicken"
  lunch->liquid$ = "coffee"
  lunch->carb$ = "rice"

// reference clusters outside of a routine with the root cluster name
  routine show_one_meal with meal
   print cluster meal
  end routine

// call the routine
  show_one_meal with meal breakfast
  show_one_meal with meal lunch
  end

"Breakfast","eggs","tea","toast"
"Lunch","chicken","coffee","rice"
```

Private Clusters

Just as with private variables, clusters can also be private in a routine. The cluster and its variables cannot be seen from outside of the routine.

Example: Private Clusters

```
program test

do_work with firstname$ = 'Sally'
do_work with firstname$ = 'Fred'
do_work with firstname$ = 'Tom'
do_work with firstname$ = 'Jane'
do_work with firstname$ = 'Julie'

private routine do_work with firstname$
   cluster my_test: b$
  add cluster my_test
  my_test->b$ = firstname$
  print 'Rows: '; size(my_test)
end routine
end
```

```
Rows:  1
Rows:  2
Rows:  3
Rows:  4
Rows:  5
```

Note: See Sheerpower and Program Segmentation for more on private routines and variables in Sheerpower.

New Sheerpower Synonyms

Full List of Synonyms in Sheerpower

_EXTRACTED	=	_COLLECTED
EXTRACT	=	COLLECT
EXTRACTED	=	COLLECTED
REEXTRACT	=	RECOLLECTED
HEADERS	=	HEADER
OUTIN	=	UPDATE
STRUCTURE	=	TABLE
GO	=	CONT
LNH	=	LISTNH
OLD	=	BUILD
REEXTRACT	=	RECOLLECTED
RNH	=	RUNNH
STATS	=	STATISTICS
FILEINFO$()	=	FILESPEC$()
STRUCT	=	CLUSTER

ARSCHK -COUNT Option

Format:

```
Command Line Format:
  ARSCHK filename.ars [-COUNT]
```

The ARSCHK ARS utility has been enhanced to include a fourth parameter option of -COUNT. This parameter tells ARSCHK to perform critical validations and output the number of records in the ARS file being checked. Without -COUNT, all the data buckets are validated which can take a long time on tables that have millions of records.

Example: ARSCHK -COUNT Option

```
// command line entry that prints to a logfile
  c:\Sheerpower\Samples>arschk client.ars -count > arschk_log.txt

// command line entry that prints to the command line console
  c:\Sheerpower\Samples>arschk client.ars -count

ARSCHK V10.88
client.ars ARS analysis starting
 Checking Prologues
 # Sector Size = 512
 Prologue Stats: (data bucketsize = 64) (maxrecsize = 400)
  # Prologue Updates:        2  # Record Deletes  :        0
  # Prologue Fixes  :        0  #Plg Reccnt < Orig:        0
  #Recycle moves    :        0  #Extent Seg Delete:        0
  #Extent Bkt Delete:        0
 Original Code version: 7.01 - Current code version 7.01
 Checking ID Block
 Checking RECord UPDate Block
 Checking KEY DEFinition Blocks
 Key Definition #1 Stats: (bucketsize = 12)
  Key Read Access count :        0  Key Bucket Split count:        0
  Empty Buckets Deleted :        0  Fix First Key Deletes :        6
  Duplicate add failures:        0
 Key Definition #2 Stats: (bucketsize = 18)
  Key Read Access count :        0  Key Bucket Split count:        0
  Empty Buckets Deleted :        0  Fix First Key Deletes :        0
Records: 15
ARSCHK finished:  SUCCESSFUL Validation of client.ars

c:\Sheerpower\Samples>
```

ARSRESTORE & ARS2ARS Enhancement

ARSRESTORE and ARS2ARS utilities have been enhanced to write out records that were rejected due to duplicate keys.

INDEX